30th International Conference on
Automated Planning and Scheduling

October 19–30, 2020, Virtual Nancy, France

PRL 2020

Proceedings of the 1st Workshop on

Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)

Edited by:

Alan Fern, Vicenç Gómez, Anders Jonsson, Michael Katz,
Hector Palacios, and Scott Sanner

Organization

Alan Fern
Oregon State University, USA

Vicenç Gómez
Universitat Pompeu Fabra, Barcelona, Spain

Anders Jonsson
Universitat Pompeu Fabra, Barcelona, Spain

Michael Katz
IBM Research AI, NY, USA

Hector Palacios
Element AI, Montreal, Canada

Scott Sanner
University of Toronto, Toronto, Canada

Foreword

While AI Planning and Reinforcement Learning communities focus on similar sequential decision-making problems, these communities remain somewhat unaware of each other on specific problems, techniques, methodologies, and evaluation.

This workshop aimed to encourage discussion and collaboration between the researchers in the fields of AI Planning and Reinforcement Learning. We aimed to bridge the gap between the two communities, facilitate the discussion of differences and similarities in existing techniques, and encourage collaboration across the fields. We solicited interest from AI researchers that work in the intersection of Planning and Reinforcement Learning, in particular, those that focus on intelligent decision making. As such, the joint workshop program was an excellent opportunity to gather a large and diverse group of interested researchers.

Alan Fern, Vicenç Gómez, Anders Jonsson, Michael Katz, Hector Palacios, and Scott Sanner
October 2020

Contents

PDDLGym: Gym Environments from PDDL Problems

Tom Silver and Rohan Chitnis
MIT Computer Science and Artificial Intelligence Laboratory
tslvr@mit.edu, ronuchit@mit.edu

Abstract

We present PDDLGym, a framework that automatically constructs OpenAI Gym environments from PDDL domains and problems. Observations and actions in PDDLGym are relational, making the framework particularly well-suited for research in relational reinforcement learning and relational sequential decision-making. PDDLGym is also useful as a generic framework for rapidly building numerous, diverse benchmarks from a concise and familiar specification language. We discuss design decisions and implementation details, and also illustrate empirical variations between the 20 built-in environments in terms of planning and model-learning difficulty. We hope that PDDLGym will facilitate bridge-building between the reinforcement learning community (from which Gym emerged) and the AI planning community (which produced PDDL). We look forward to gathering feedback from all those interested and expanding the set of available environments and features accordingly.

1 Introduction

The creation of benchmarks has often accelerated research progress in various subdomains of artificial intelligence (Deng et al. 2009; Wang et al. 2018; Wu et al. 2018). In sequential decision-making tasks, tremendous progress has been catalyzed by benchmarks such as the environments in OpenAI Gym (Brockman et al. 2016) and the planning tasks in the International Planning Competition (IPC) (Vallati et al. 2015). Gym defines a standardized way for an agent to interact with an environment, allowing easy comparison of various reinforcement learning algorithms. IPC provides a set of planning domains and problems written in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998), allowing easy comparison of symbolic planners.

In this work, we present PDDLGym, an open-source framework that combines elements of Gym and PDDL. **PDDLGym is a Python library that automatically creates Gym environments *from* PDDL domain and problem files.** The library is available at: https://github.com/tomsilver/pddlgym. Pull requests are welcome!

As with Gym, PDDLGym allows for episodic, closed-loop interaction between the agent and the environment; the agent receives an observation from the environment and gives back an action, repeating this loop until the end of an episode. As in PDDL, PDDLGym is fundamentally relational: observations are sets of ground relations over objects (e.g. `on(plate, table)`), and actions are templates ground with objects (e.g. `pick(plate)`). PDDLGym is therefore particularly well-suited for relational learning and sequential decision-making research. See Figure 1 for renderings of some environments currently implemented in PDDLGym, and Figure 2 for code examples.

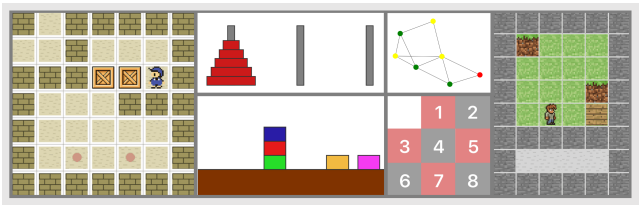


Figure 1: **Some examples of environments implemented in PDDLGym.** From top left: Sokoban, Hanoi, Blocks, Travelling Salesman (TSP), Slide Tile, and Crafting.

The Gym API used in reinforcement learning defines a hard boundary between the agent and the environment. In particular, the agent *only* interacts with the environment by taking actions and receiving observations. The environment implements a function `step` that advances the state given an action by the agent; `step` defines the transition model of the environment. Likewise, a PDDL domain encodes a transition model via its operators. However, in typical usage, PDDL is understood to exist entirely in the "mind" of the agent. A separate process is then responsible for turning plans into actions that the agent can execute in the world.

PDDLGym defies this convention: in PDDLGym, PDDL domains and problems lie firmly on the environment side of the agent-environment boundary. The environment uses the PDDL files to implement the `step` function that advances the state given an action. PDDLGym is thus perhaps best understood as a *repurposing* of PDDL. Implementation-wise, this repurposing has subtle but important implications, discussed in (§2.2) and Appendix A.

ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL), 2020.

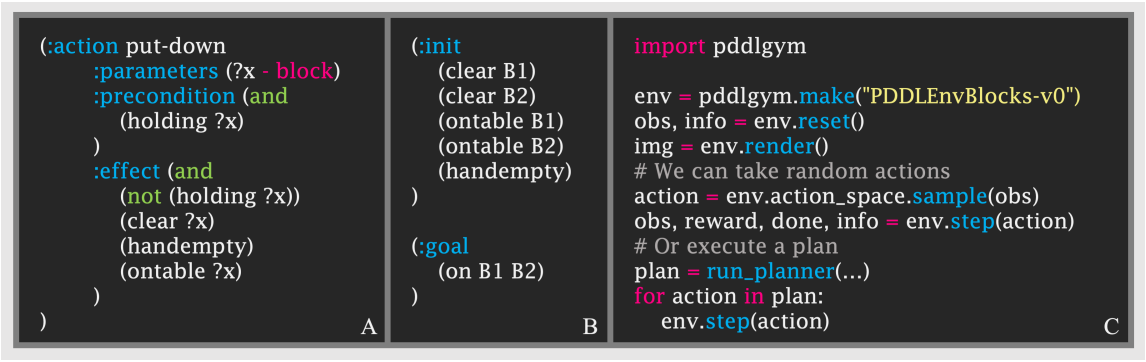


Figure 2: **PDDLGym code examples.** A PDDLGym environment is characterized by a PDDL domain file and a list of PDDL problem files. (A) One operator in the PDDL domain file for Blocks. (B) An excerpt of a simple PDDL problem file for Blocks. (C) After the PDDL domain and problem files have been used to register an environment with name "PDDLEnvBlocks-v0," we can interact with this PDDLGym environment in just a few lines of Python.

PDDLGym serves three main purposes:

(1) *Facilitate the creation of numerous, diverse benchmarks for sequential decision-making in relational domains*. PDDLGym allows tasks to be defined in PDDL, automatically building a Gym environment from PDDL files. PDDL offers a compact symbolic language for describing domains, which might otherwise be cumbersome and repetitive to define directly via the Gym API.

(2) *Bridge reinforcement learning and planning research.* PDDLGym makes it easy for planning researchers and machine learning researchers to test their methods on the exact same set of benchmarks, and to develop techniques that draw on the strengths of both families of approaches. Furthermore, since PDDLGym includes built-in domains and problems, it is straightforward to perform apples-to-apples comparisons without having to collect third-party code from disparate sources (see also (Muise 2016)).

(3) *Catalyze research on sequential decision-making in relational domains.* In our own research, we have found PDDLGym to be very useful while studying exploration for lifted operator learning (Chitnis et al. 2020), hierarchical goal-conditioned policy learning (Silver et al. 2020a), and state abstraction (Silver et al. 2020b). Other open research problems that may benefit from using PDDLGym include relational reinforcement learning (Lang, Toussaint, and Kersting 2012; Džeroski, De Raedt, and Driessens 2001; Tadepalli, Givan, and Driessens 2004), learning symbolic descriptions of operators (Lang, Toussaint, and Kersting 2012; Amir and Chang 2008; Pasula, Zettlemoyer, and Kaelbling 2007), discovering relational transition rules for efficient planning (Xia et al. 2019; Lang and Toussaint 2010), and learning lifted options (Konidaris, Kaelbling, and Lozano-Perez 2014; Stolle and Precup 2002; Precup, Sutton, and Singh 1998; Chentanez, Barto, and Singh 2005).

2 Design and Implementation

The Gym API defines environments as Python classes with three essential methods: `__init__`, which initializes the environment; `reset`, which starts a new episode and returns an observation; and `step`, which takes an action from the agent, advances the current state, and returns an observation, reward, a Boolean indicating whether the episode is complete, and optional debugging information. The API also includes other minor methods, e.g., to handle rendering and random seeding. Finally, Gym environments are required to implement an `action_space`, which represents the space of possible actions, and an `observation_space`, which represents the space of possible observations. We next give a brief overview of PDDL files, and then we describe how action and observation spaces are defined in PDDLGym. Subsequently, we move to a discussion of our implementation of the three essential methods. For implementation details regarding the main data structures used in PDDLGym, see `structs.py` in the code.

2.1 Background: Domain and Problem Files

There are two types of PDDL files: domain files and problem files. A single *benchmark* is characterized by one domain file and multiple problem files.

A PDDL domain file includes *predicates* — named relations with placeholder variables such as (`on ?x ?y`) — and *operators*. An operator is composed of a name, a list of parameters, a first-order logic formula over the parameters describing the operator's preconditions, and a first-order logic formula over the parameters describing the operator's effects. The forms of the precondition and effect formulas are typically restricted depending on the version of PDDL. Early versions of PDDL only permit conjunctions of ground predicates (Fikes and Nilsson 1971); later versions also allow disjunctions and quantifiers (Pednault 1989). See Figure 2A for an example of a PDDL operator.

A PDDL problem file includes a set of *objects* (named entities), an *initial state*, and a *goal*. The initial state is a set of predicates ground with the objects. Any ground predicates not in the state are assumed to be false, following the closed-world assumption. The goal is a first-order logical formula over the objects (the form of the goal is limited by the PDDL version, like for operators' preconditions and effects). Note

that PDDL (and PDDLGym) also allows objects and variables to be *typed*. See Figure 2B for a partial example.

2.2 Observation and Action Spaces

Each observation `obs` in PDDLGym has three components, mirroring the components of a PDDL problem file: `obs.objects` is a set containing all objects present in the problem; `obs.goal` contains the problem goal; and `obs.literals` is a set of all ground predicates that are true in the current state. These observations fully encapsulate the state of the environment, i.e., PDDLGym environments are fully-observed. The observation space is the powerset of all possible ground predicates, together with the objects and goal, which are static. This powerset is typically enormous; fortunately, it usually does not need to be explicitly computed. We expect that most algorithms for solving PDDLGym tasks will not be sensitive to its size.

The action space for a PDDLGym environment is one of the more subtle aspects of the overall framework, and there are two possible avenues to take. Instructions for taking both avenues are provided in the repository's README, in the "Step 3: Register Gym environment" section. We include a detailed discussion of the action spaces and the related design choices in Appendix A.

2.3 Initializing and Resetting an Environment

A PDDLGym environment is parameterized by a PDDL domain file and a list of PDDL problem files. For research convenience, each PDDLGym environment is associated with a *test* version of that environment, where the domain file is identical but the problem files are different (for instance, they could encode more complicated planning tasks, to measure generalizability). During environment initialization, all of the PDDL files are parsed into Python objects; we use a custom PDDL parser for this purpose. When `reset` is called, a single problem instance is randomly selected.[1] The initial state of that problem instance is the state of the environment. For convenience, `reset` also returns (in the debugging information) paths to the PDDL domain and problem file of the current episode. This makes it easy to run to a symbolic planner and execute resulting plans in the environment; see the repository's README for an example that uses Fast-Forward (Hoffmann 2001).

2.4 Implementing `step`

The `step` method of a PDDLGym environment takes in an action, updates the environment state, and returns an observation, reward, done Boolean, and debugging information. To determine the state update, PDDLGym checks whether any PDDL operator's preconditions are satisfied given the current state and action. Note that it is impossible to "accidentally match" to an undesired operator: each operator has a unique precondition as illustrated in Figure 4C, which is generated automatically based on the passed-in action. Since actions are distinct from operators (Appendix A), this precondition satisfaction check is non-trivial; non-free parameters must be bound. We have implemented two inference back-ends to perform this check. The first is a Python implementation of typed SLD resolution, which is the default choice when the query involves only conjunctions. The second is a wrapper around SWI Prolog (Wielemaker et al. 2012), which permits us to handle more sophisticated preconditions involving disjunctions and quantifiers. The latter is slower, but more general, than the former. When no preconditions hold for a given action, the state remains unchanged by default. In some applications, it may be preferable to raise an error if no preconditions hold; the optional initialization parameter `raise_error_on_invalid_action` permits this.

Rewards in PDDLGym are sparse and binary. In particular, the reward is 1.0 when the problem goal is satisfied and 0.0 otherwise. Similarly, the done Boolean is True when the goal is reached and False otherwise. (In practice, a maximum episode length is often used.)

If the underlying PDDL domain has probabilistic effects, as in PPDDL (Bryce and Buffet 2008), the `step` method will parse this appropriately and choose an effect based on the given probability distribution. If the given probabilities do not sum to 1, a default trivial effect is added in.

2.5 Development Status

In terms of lines of code, the bulk of PDDLGym is dedicated to PDDL file parsing and inference (used in `step`). We are continuing to develop both of these features so that a wider range of PDDL domains are supported. Aspects of PDDL 1.2 that are supported by PDDLGym include STRIPS, hierarchical typing, equality, quantifiers, constants, and derived predicates. Notable features that are not supported include conditional effects and action costs. Aspects of later PDDL versions, such as numeric fluents, are not supported. Our short-term objective is to provide full support for PDDL 1.2. We have found that a wide range of standard PDDL domains are already well-supported by PDDLGym; see (§3) for an overview. We welcome requests for features and extensions, via either issues created on the Github page or email. The authors' emails are provided at the top of this document.

3 PDDLGym by the Numbers

In this section, we start with an overview of the domains built into PDDLGym, as of the last date this report was updated (September 22, 2020). We then provide some experimental results that give insight into the variation between these domains, in terms of planning and model-learning difficulty. All experiments are performed on a single laptop with 32GB RAM and a 2.9GHz Intel Core i9 processor.

3.1 Overview of Environments

There are currently 20 domains built into PDDLGym. Most of the domains are adapted from existing PDDL repositories; the remainder are ones we found to be useful benchmarks in our own research. We have implemented custom rendering for 11 of the domains (see Figure 1 for examples). Table 1 gives a list of all environments, their sources, and

[1] Problem selection when resetting an episode is the only use of randomness in PDDLGym (aside from stochastic transitions).

Domain Name	Source	Rendering Included	Probabilistic	Average FPS
Baking	Ours	No	No	5897
Blocks	(Helmert 2011)	Yes	No	7064
Casino	Ours	No	No	7747
Crafting	Ours	Yes	No	4568
Depot	(Helmert 2011)	No	No	97
Doors	(Konidaris and Barto 2007)	Yes	No	917
Elevator	(Helmert 2011)	No	No	3501
Exploding Blocks	(Bryce and Buffet 2008)	Yes	Yes	6260
Ferry	(CSU 2002)	No	No	1679
Gripper	(Soar 2020)	Yes	No	319
Hanoi	(Soar 2020)	Yes	No	4580
Meet-Pass	(CSU 2002)	No	No	7380
Rearrangement	Ours	Yes	No	3808
River	(Bryce and Buffet 2008)	No	Yes	18632
Search and Rescue	Ours	Yes	No	3223
Slide Tile	(Soar 2020)	Yes	No	3401
Sokoban	(Helmert 2011)	Yes	No	155
Triangle Tireworld	(Bryce and Buffet 2008)	No	Yes	6491
TSP	(Soar 2020)	Yes	No	1688
USA Travel	Ours	No	No	1251

Table 1: **List of the 20 domains currently included in PDDLGym, as of the last date this report was updated (September 22, 2020).** For each environment, we report the original source of the PDDL files, whether or not we have implemented custom rendering, whether or not the domain has probabilistic effects, and the average frames per second (FPS). The FPS is calculated by executing a random policy for 100 episodes of 10 timesteps each, with no rendering.

their average frames per second (FPS) calculated by executing a random policy for 100 episodes of 10 timesteps each.

3.2 Variation in Environment Difficulty

We now provide some results illustrating the variation between the domains built into PDDLGym. We examine two axes of variation: planning difficulty and difficulty of learning the transition model.

Figure 3 (left) illustrates the average time taken by FastForward (Hoffmann 2001) to find a plan in each of the deterministic environments, averaged across all problem instances. The results reveal a considerable range in planning time, with the most difficult domain (Depot, omitted from the plot for visual clarity) requiring two orders of magnitude more time than the simplest one (TSP). The results also indicate that many included domains are relatively "easy" from a modern planning perspective. However, even in these simple domains, there are many interesting challenges to be tackled, such as learning the true PDDL operators from interaction data, or defining good state abstractions amenable to learning. One can always make larger problem instances if desired, to push the limits of modern planners.

Figure 3 (right) provides insight into the difficulty of learning transition models in some of the environments. For each environment, an agent executes a random policy for episodes of horizon 25. The observed transitions are used to learn transition models, which are then used for planning on a suite of test problems. The fraction of test problems solved is reported as an indicator of the learned transition model. To learn the transition models, we use first-order logic decision tree (FOLDT) learning (Blockeel and De Raedt 1998).

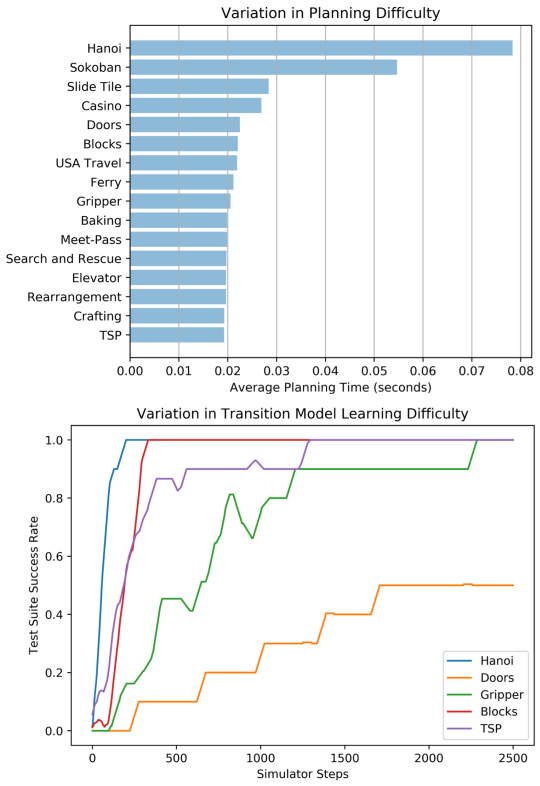


Figure 3: **Variation among PDDLGym environments.** The PDDL domains and problems built into PDDLGym vary considerably in terms of planning difficulty (top) and model learning difficulty (bottom). See text for details.

Five domains are visualized for clarity; among the remaining ones, several are comparable to the ones shown, but others, including Baking, Depot, and Sokoban, are difficult for our learning method: FOLDT learning is unable to find a model that fits the data in a reasonable amount of time. Of course, model-learning difficulty varies considerably with the learning method and the exploration strategy. We have implemented simple strategies here to show these results, but these avenues for future research are exactly the kind that we hope to enable with PDDLGym.

4 Conclusion and Future Work

We have presented PDDLGym, an open-source Python framework that automatically creates OpenAI Gym environments from PDDL domain and problem files. Our empirical results demonstrate considerable diversity among the built-in environments. We have been using PDDLGym actively in our own research on relational sequential decision-making and reinforcement learning. We also hope to interface PDDLGym with other related open-source frameworks, particularly the PDDL collection in planning.domains (Muise 2016), so that one can use PDDLGym simply by specifying a URL (along with information about free parameters).

We look forward to gathering feedback from the community and expanding the set of environments and features.

References

Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.

Blockeel, H., and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial intelligence* 101(1-2):285–297.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540*.

Bryce, D., and Buffet, O. 2008. International planning competition uncertainty part: Benchmarks and results. In *In Proceedings of IPC*. Citeseer.

Chentanez, N.; Barto, A. G.; and Singh, S. P. 2005. Intrinsically motivated reinforcement learning. In *Advances in neural information processing systems*, 1281–1288.

Chitnis, R.; Silver, T.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Perez, T. 2020. GLIB: Exploration via goal-literal babbling for lifted operator learning. *arXiv preprint arXiv:2001.08299*.

CSU. 2002. Colorado state university PDDL repository. https://www.cs.colostate.edu/meps/repository.html.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, 248–255. Ieee.

Džeroski, S.; De Raedt, L.; and Driessens, K. 2001. Relational reinforcement learning. *Machine learning* 43(1-2):7–52.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2011. Pyperplan. https://bitbucket.org/malte/pyperplan.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI magazine* 22(3):57–57.

Konidaris, G., and Barto, A. G. 2007. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, 895–900.

Konidaris, G.; Kaelbling, L.; and Lozano-Perez, T. 2014. Constructing symbolic representations for high-level planning. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Lang, T., and Toussaint, M. 2010. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research* 39:1–49.

Lang, T.; Toussaint, M.; and Kersting, K. 2012. Exploration in relational domains for model-based reinforcement learning. *Journal of Machine Learning Research* 13(Dec):3725–3768.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Muise, C. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research* 29:309–352.

Pednault, E. P. 1989. ADL: Exploring the middle ground between strips and the situation calculus. *Kr* 89:324–332.

Precup, D.; Sutton, R. S.; and Singh, S. 1998. Theoretical results on reinforcement learning with temporally abstract options. In *European conference on machine learning*, 382–393. Springer.

Silver, T.; Chitnis, R.; Ajay, A.; Tenenbaum, J.; and Kaelbling, L. P. 2020a. Learning skill hierarchies from predicate descriptions and self-supervision. In *AAAI GenPlan Workshop*.

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J.; Lozano-Perez, T.; and Kaelbling, L. P. 2020b. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*.

Soar. 2020. Soar group PDDL repository. https://github.com/SoarGroup.

Stolle, M., and Precup, D. 2002. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, 212–223. Springer.

Tadepalli, P.; Givan, R.; and Driessens, K. 2004. Relational reinforcement learning: An overview. In *Proceedings of the ICML-2004 workshop on relational reinforcement learning*, 1–9.

Vallati, M.; Chrpa, L.; Grześ, M.; McCluskey, T. L.; Roberts, M.; Sanner, S.; et al. 2015. The 2014 international planning competition: Progress and trends. *Ai Magazine* 36(3):90–98.

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. R. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wielemaker, J.; Schrijvers, T.; Triska, M.; and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12(1-2):67–96.

Wu, Z.; Ramsundar, B.; Feinberg, E. N.; Gomes, J.; Geniesse, C.; Pappu, A. S.; Leswing, K.; and Pande, V. 2018. MoleculeNet: a benchmark for molecular machine learning. *Chemical science* 9(2):513–530.

Xia, V.; Wang, Z.; Allen, K.; Silver, T.; and Kaelbling, L. P. 2019. Learning sparse relational transition models. *International Conference on Learning Representations*.

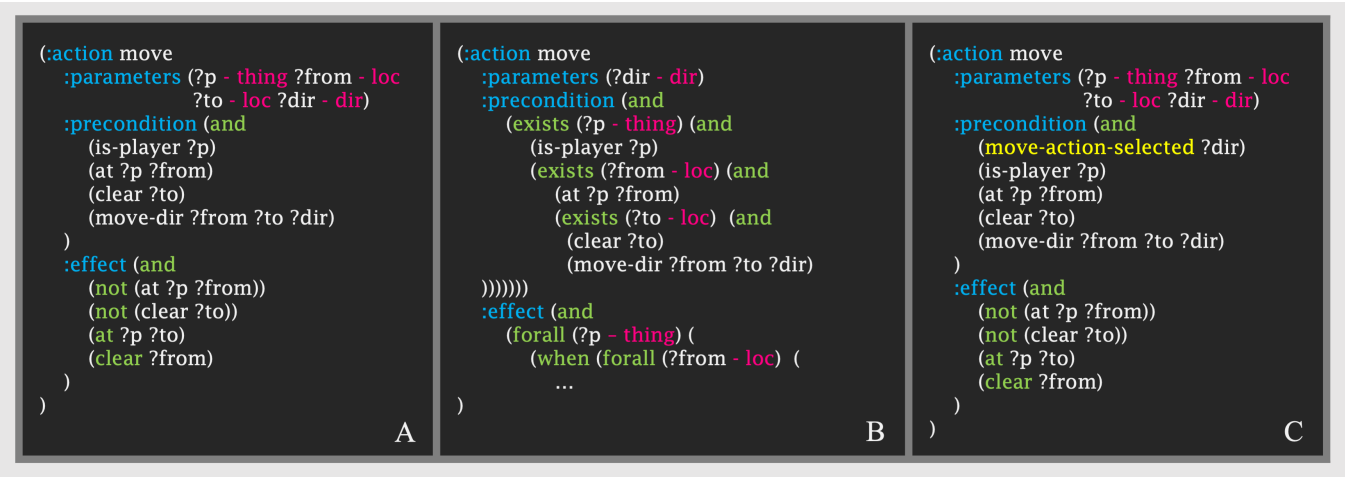


Figure 4: **Explicating free parameters in PDDL operators.** PDDL operators traditionally conflate free and non-free parameters. For example, in a typical move operator for Sokoban (A), the free parameter `?dir` is included alongside non-free parameters. PDDLGym must distinguish free parameters to properly define the action space. One option would be to require that all operator parameters are free, and introduce quantifiers in the operator body accordingly (B); however, this is cumbersome and leads to clunky, deeply nested operators, so we do *not* do this. Instead, we opt to introduce new predicates that are tied to operators, and whose parameters are just the operators' free parameters (C). An example of such a new predicate is shown in yellow (`move-action-selected`).

A Action Space Details

The action space for a PDDLGym environment is one of the more subtle aspects of the overall framework, and there are two possible avenues to take. Instructions for taking both avenues are provided in the repository's README, in the "Step 3: Register Gym environment" section.

The first avenue is appropriate if one wants to simply use off-the-shelf PDDL files with PDDLGym. One can do so by setting `operators_as_actions` to `True` in the environment registration, which tells PDDLGym that the operators present in the PDDL domain file should themselves be treated as the actions in the environment, parameterized by those operators' parameters.

The second avenue is recommended for more serious research, and stems from the semantic difference between "operators" in classical AI planning and "actions" in reinforcement learning. In AI planning, actions are typically equated with ground operators — operators whose parameters are bound to objects. However, in most PDDL domains, only some operator parameters are *free* (in terms of controlling the agent); the remaining parameters are included in the operator because they are part of the precondition/effect expressions, but can be derived from the current state or the choice of free parameters. PDDL itself makes no distinction between free and non-free parameters. For example, consider the operator for Sokoban shown in Figure 4A. This operator represents the rules for a player (`?p`) moving in some direction (`?dir`) from one cell (`?from`) to another cell (`?to`). In a real game of Sokoban, the only choice that an agent makes is what direction to move — only the `?dir` parameter is free. The player `?p` is always the same, `?from` is defined by the agent's location in the current state, and `?to` can be derived from `?from` and the agent's choice of `?dir`. To properly define the action space for a PDDLGym environment, we must explicitly distinguish free parameters from non-free ones. One option is to require that operator parameters are all free. Non-free parameters could then be folded into the preconditions and effects using quantifiers (Pednault 1989); see Figure 4B for an example. However, this is cumbersome and leads to clunky, deeply nested operators. Instead, we opt to introduce new predicates that represent operators, and whose variables are these operators' free parameters. We then include these predicates in the preconditions of the respective operators; see Figure 4C for an example. Doing so requires only minimal changes to existing PDDL files and does not affect readability, but requires adding in domain knowledge about the agent-environment boundary. Note that this domain knowledge is equivalent to defining an action space, which is very commonly done in reinforcement learning and is not a strong assumption. In this case, the action space of a PDDLGym environment is a discrete space over all possible groundings of the newly introduced predicates.

When sampling from the action space of a PDDLGym environment, PDDLGym will automatically only sample *valid* actions, i.e., actions that satisfy the preconditions of some operator. This check for validity is done using Fast Downward's translator (Helmert 2006), which can add non-negligible overhead in large problem instances.

Model-free Automated Planning Using Neural Networks

Michaela Urbanovská and **Jan Bím** and **Leah Chrestien** and **Antonín Komenda** and **Tomáš Pevný**
Department of Computer Science, Faculty of Electrical Engineering
Czech Technical University in Prague
{michaela.urbanovska,chreslea,antonin.komenda,pevnytom}@fel.cvut.cz,
jan.bim@datamole.cz

Abstract

Automated planning for problems without an explicit model is an elusive research challenge, which, however, if tackled, could provide general approach to problems in real-world unstructured environments. There are currently two strong research directions in the area of Artificial Intelligence (AI), namely, machine learning and symbolic AI. The former provides techniques to learn models of unstructured data but does not provide further problem solving capabilities on such models. The latter provides efficient algorithms for general problem solving, but requires a model to work with. In this paper, we propose a combination of these two areas, namely deep learning and classical planning, to form a planning system that works without a human-encoded model. The deep learning part extracts the model in a form of a transition system and a goal-distance heuristic estimator; the classical planning part uses such a model to efficiently solve the planning problem. Besides the design of such planning systems, we provide experimental evaluations comparing the implemented technique to classical model-based methods.

Introduction

The main focus of this work is to analyze the possibilities and limitations of using deep learning in combination with classical planning. Instead of replacing the planning process as a whole and trying to make the network learn a search algorithm, we decided to focus on partial replacement of two components involved in many standard planning algorithms, namely, the transition system and heuristic functions.

Classical planning provides great methods for general problem solving. Unfortunately, these methods can struggle in large unstructured domains. On the other hand, deep learning methods have been demonstrated to work well on many domains without a clear structure. Therefore, combining both of these methods may remove the need for an explicit planning model.

Asai and Fukunaga in (Asai and Fukunaga 2017) and (Asai and Fukunaga 2018) connected deep learning and classical planning by creating LatPlan, which is a system that takes in an initial and a goal state of a problem instance and returns a visualised plan execution. The image on the input is transformed and processed in order to generate a standardized problem representation, which can then be solved by classical planning methods. In (Garrett, Kaelbling, and Lozano-Pérez 2016), Garret et al. uses machine learning techniques to create heuristic functions that improve a search algorithm. Finally, in (Gomoluch et al. 2019), learning policies for search algorithms is provided.

This work can be understand as a follow-up work of Asai and Fukunaga's architecture. In contrast to their work, we use maze-like problems and images of these mazes are the input to our algorithm. The solution is then produced as a sequential plan navigating through our designed transition system, which in turn is generated by our learned model. To increase efficiency of the search, we use heuristic principle proposed by Garret et al.

Background

Classical Planning

Let's first focus on STRIPS, which provides a symbolic representation to a model-based planning problem instance.

Definition 1 (STRIPS Planning Task) *A STRIPS planning task* Π *is a tuple*

$$\Pi = \langle F, O, s_i, s_g, c \rangle$$

where $F = \{f_1, f_2, ..., f_n\}$ *is a set of facts, which can hold in the world. A state of the world is defined by facts which hold in the state* $s \subseteq F$ *.* $O = \{o_1, o_2, ..., o_m\}$ *is a set of operators transforming the world,* s_i *is the initial state, which consists of facts that hold in the initial state,* s_g *is a goal state condition, which contains facts that hold in every goal state and* c *is a cost function* $c(o) : o \rightarrow R^+$ *which gives each operator a positive cost.*

Every operator $o \in O$ *is a tuple where* $o = \langle pre(o), add(o), del(o) \rangle$*,* $pre(o) \subseteq F$ *is a set of preconditions, which are facts, that have to hold in a state for the operator* o *to be applicable in that state,* $add(o) \subseteq F$ *is a set of facts, which are added to the state after applying the operator* o *in* s *and* $del(o) \subseteq F$ *are delete effects, which are facts that are no longer true after using the operator* o*.*

To create a STRIPS representation of a problem, we have to be able to construct the facts and the operators from the problem definition. To find a solution of such a problem, we construct a state-transition system, in which we look for a path to the goal state from the initial state.

Definition 2 (Transition System) *A transition system is a tuple* $\Sigma = \langle S, A, \gamma, c \rangle$*, where*

- S *is a finite set of states*
- A *is a finite set of actions*

- $\gamma : S \times A \rightarrow S$ *is a state-transition function.* $\gamma(s, a)$ *is defined iff* a *is applicable in* s*, with* $\gamma(s, a)$ *being the predicted outcome.*
- $cost : A \rightarrow [0, \infty)$ *is a cost function assigning a value to each action. The cost value can have various meanings, for example time, price or anything we want to optimize.*

Any problem Π defined as STRIPS by Definition 1 can be translated to a transition system Σ and solved by the means of path-searching algorithms.

In order to find a solution to a planning problem, we need to find a path through the induced transition system, which is typically done by one of the heuristic state-space search algorithms.

Definition 3 (State-Space Search) *State-space search algorithm performs search over a graph* $G = (N, E)$*, where* N *is a set of nodes and* E *is a set of edges. Having a planning problem* $\Pi = \langle F, O, s_i, s_g, c \rangle$ *and its induced transition system* $\Sigma = \langle S, A, \gamma, c \rangle$*,* N *corresponds to* S *and* E *corresponds to* A*. The search starts in* s_i*, expanding each found state with* γ*, until a goal state is reached. In that case, plan* π *can be returned as a sequence of actions applied at each expansion of the search in order to reach the goal state.*

State space of many problems can be very large and exhaustive search may not be the most efficient way to look for the solution. Generally speaking, state-space search can be done blindly, however, additional information can greatly improve its performance. This additional information is added in the form of heuristic functions.

Heuristic function $h(s) : s \rightarrow R^+$ maps any state $s \in S$ to a positive value. Heuristic function gives us an estimate of path length from the current state s to a goal state. A function, which always maps $h(s)$ to the length of shortest possible path is called perfect or optimal heuristic and is denoted as h^*.

Neural Networks

Neural networks have proved to be a very powerful tool in many different domains. Here, we use the current state of the art approach which uses feed-forward neural networks that learn through back-propagation using Stochastic Gradient Descent (Ruder 2016).

Our primary aim lies in creating two networks, each one to substitute a different part of the state-space search algorithm. First network is used to replace the state-transition function γ in order to generate possible successor states in the state-transition system as defined in Definition 2. The second network is used to replace a heuristic function $h(s)$, which returns a number representing an estimate of the distance from state s to a goal state.

These two networks are implemented using convolutional neural networks (CNNs) as described in (LeCun, Bengio, and Hinton 2015).

The problem domains, in our case, have image-like grid structures which makes CNNs a viable choice in trying to extract information from the representations.

By replacing these parts of the state-space search, we avoid the need of creating a symbolic representation of the states which would have been necessary if we were to adhere to classical architectures. Thus, in this manner, we make our problem domains model free.

Attention for Neural Networks

In the recent past, by introducing the Transformer architecture, attention networks have proven to be a great success as shown in (Vaswani et al. 2017). In general, attention allows the network to focus only on subsets of inputs and requires the creation of attention masks.

In this work, we use soft attention in which the network focuses on input values that are between 0 and 1 as opposed to hard attention where the network focuses on either zeroes or ones.

Masks are generated using convolutional layer and a softmax layer of the same width and heigth as the input, with all values summing up to one. On having such a mask, we can then multiply the input features, resulting in a modified input with some of the features "emphasized" by the attention mask. The layers, which generate the attention masks are also trained with the whole architecture.

Data Domains

One of the key challenges of implementing the proposed approach is obtaining a good quality data set in order to train the networks. As CNNs are well suited to process data sets organized in grids, we consider problem domains which can be represented by a grid.

We use four problem domains; for each one of them, we create generators in order to obtain enough data. Examples of all four domains are shown in Figure 1.

First domain is a maze with one agent and one goal. The agent can move only in its 4-neighborhood and every free cell in the maze is accessible by the agent.

The second domain is the same, except we have multiple goals in the map. Therefore, we call this one the multi-goal maze. The goal in this domain is reached when the agent arrives in one of the goals.

The third domain is a multi-agent maze where the same rules apply, but we have the number of agents greater than one and the same number of goals in the maze. All the agents have to move at the same time and the goals are not assigned to a specific agent. The goal is reached when every agent in the map stands on a goal.

The last domain we use is the Sokoban puzzle which is similar to the maze domains in terms of movement, but it is more complicated because of the added box entity. Agent can push a box if there's a free space behind it and it's not possible to push multiple boxes at the same time. The main goal is to move every box to an arbitrary goal position in the map. Once every box is on a goal position, we reach the goal state.

Expansion Network

In the state-space search (Definition 3), transition function takes in a current state of the search and returns all its suc-

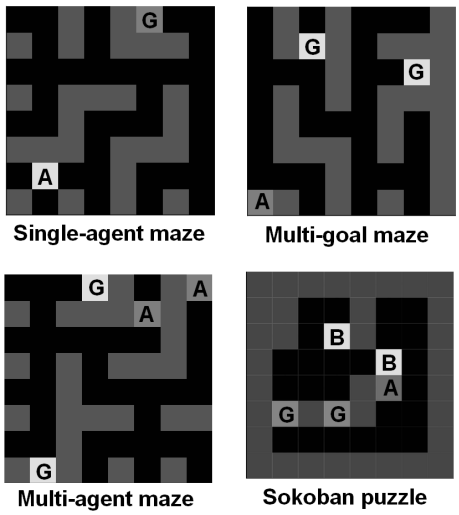


Figure 1: Examples of all four problem domains - **A** denotes agent, **G** denotes goal, **B** denotes box (in Sokoban domain)

cessors. The expansion network is used to generate these successors.

By observing pairs of states (in form of images of the mazes with the agent) without knowledge about actions that connect them, we want to learn possible actions for the problem. Even for the most simple maze domain, the size of the data set is important in order to train the network. The task does not only lay in the locating the free spaces around the agent. We need to make sure that the maze structure remains the same and no rules are broken on performing the learned actions.

We work with mazes represented as images, therefore, as mentioned earlier, it is convenient to use a CNN for this task. As we want to focus on the 4-neighborhood of the agent, we choose the kernel as a 3x3 window. Since we want to preserve the size of the input through the whole network, the padding for our CNN is equal to one.

To additionally improve the network, we use residual connections. A residual connection is an architecture modification, which is often used in deep learning and has achieved great results in learning an identity function, for example, in ResNet image classification network (He et al. 2016). Furthermore, residual connections resulted in a reduction in the complexity of the network and an improvement in the results.

Our expansion network has one residual connection which connects the input data with the output of the three chained convolutional layers. After concatenating these two parts of data, we process them through a 1x1 convolution to adjust the number of channels to match the input. In order to obtain even better results, we added normalization in the form of dropout between the first three convolutional layers. We can see the whole architecture in Figure 2.

Input and Output

The input of this network is the visual representation of the problem encoded in one-hot representation. This gives us a one-hot encoded vector for each cell in the problem's image, which tells us, which entity is in the cell based on the placement of the ones and zeros in the vector.

The output of this network is exactly the same in size, and it is similar to the one-hot encoding, however, the values on the output give us a distribution of the reachable next states. We can then use a threshold in order to extract the possible future states from this output.

Loss Function

In this network, we want to learn probabilistic distributions of the possible successor states of a current state. Our data is one-hot encoded, which means that there is a vector for each cell in the map or maze. According to the entity type placed on that cell, we put 1 to the corresponding index in the vector.

To train the network properly, we have to use a loss function suited for measuring the accuracy between true distribution and learned distribution of the possible successor states. Therefore, we use logistic cross entropy as our loss function for this network.

Heuristic Network

Another function in the state-space search (Definition 3) is the heuristic function which aids the search process. Computation of a heuristic is a non-trivial problem, especially in the case of non-simplified visual representation.

Since the input data is still based on the visual representation of the problem, convolutional networks are a good direction to explore. Inspired by the classical planning approaches for computing heuristics, we use attention to simplify the problem. Such simplification is usually used in classical planning heuristics in the form of relaxations or abstractions (Ghallab, Nau, and Traverso 2016).

Input and Output

The heuristic network receives a one-hot encoded visual representation of an input state. The label is the value of the h^* heuristic which is the length of the shortest path from the input state to the closest goal state. Therefore, we want the network to produce a single value for each input as well. This generated value is the heuristic in the planning algorithm.

Although learning a heuristic estimator from optimal plans sounds rather nonsensical (why to learn something we know the ground truth for), it acts in this work more as a proof of concept. The overall idea is to learn the heuristic from the best information about the distances to the goal. Since the learning process generalizes, it should be able to provide heuristic estimates even for cases in which it did not learn.

Loss Function and Data Set Structure

Training a network to return heuristic values requires optimal plan lengths as labels for all the data. For each maze

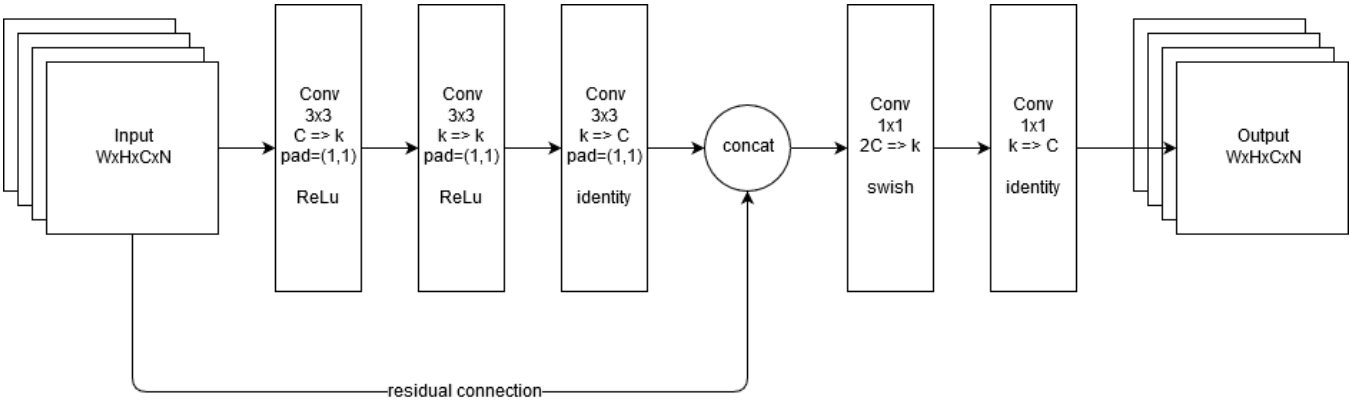


Figure 2: Expansion network architecture. Size of the input is width (W), heigth (H), number of channels (C) and number of samples in a batch. Each convolutional layer **Conv** has size of its kernel, number of input channels and output channels of the layer (C $\Rightarrow$ k means **C** input channels and **k** output channels), padding and activation function. In this network, we use rectified linear unit (**ReLu**), **swish** (modified ReLu) or no activation (**identity**).

instance in the original data set, we randomly picked multiple positions from all possible agent placements and added those randomly picked samples with their computed labels (h^* values) into the data set. To train the network, we created the batches by taking a selected number of positions from each maze instance so that there were always multiple different agent placements in the maze at one batch. This is important, because loss function in this case has to be focused on the relation between the values in the same problem instance and not just the pair (state, value).

Since neural networks represent a black-box approximation scheme, we cannot expect to ensure any properties of the generated values. Therefore, we used satisficing planning for our experiments. In our case, the additional heuristic information was provided by the trained heuristic network.

Another property which can influence the performance, also held by the labels is monotonicity. Having monotonic heuristic values for all the states provides us with a possibility of selecting the best states on just following the descent of the state heuristic values. To get as close as possible to this property, we implemented a custom loss function, which measures how far off is the monotonicity of the learned values when compared to the h^* values.

```
function loss(mx,y)
partial_losses = []
for every maze instance in batch
    ixs = all indexes of the instance
    data_diffs = mx[ixs] - transpose(mx[ixs])
    labels_diffs = y[ixs] - transpose(y[ixs])
    tmp = -data_diffs * sign(labels_diffs)+1
    l = sum(max(0, tmp)
    partial_losses.add(l)
end
return sum(partial_losses)
end
```

Listing 1: Pseudocode of loss for the heuristic network

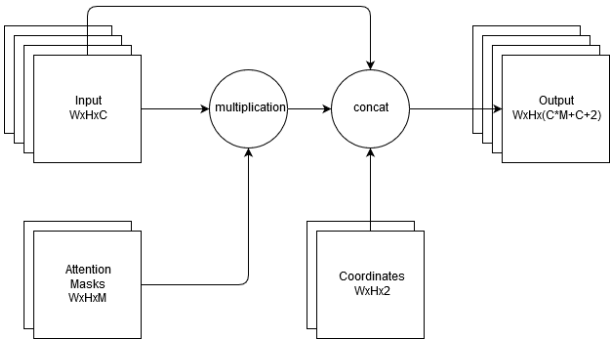


Figure 3: Attention block used in the heuristic network architecture - **W** is width of the input, **H** is height of the input, **C** is number of channels of the input structure

Architecture

One of the most notable features in this architecture is the usage of attention as described in section *Attention for Neural Networks*. The is analogous to the relaxation technique (Ghallab, Nau, and Traverso 2016) used in planning. If we imagine looking at a maze and identifying interesting parts of it, such as crossroads or long straight paths, we might simplify the problem enough to obtain a distance estimate from the agent to the goal.

Implementation of attention was done by using convolutional layers and using softmax over the first two dimensions of the input. Meaning, at the end, we received the attention mask, which has the same width and height as the input and all its values sum up to one (soft attention). One problem which comes up on using attention is deciding on the number of attention masks required to find enough attention-worthy places in the data. We experimented with different numbers of attentions in the architecture while selecting the networks that are further used in the planning experiments.

The input of this network is of the size $W \times H \times C \times N$; first two dimensions are width and height of the data; the third is the number of channels and the last one is number of samples in the processed mini-batch. Size of the mini-batch has been deliberately omitted in the architecture diagrams for simplification.

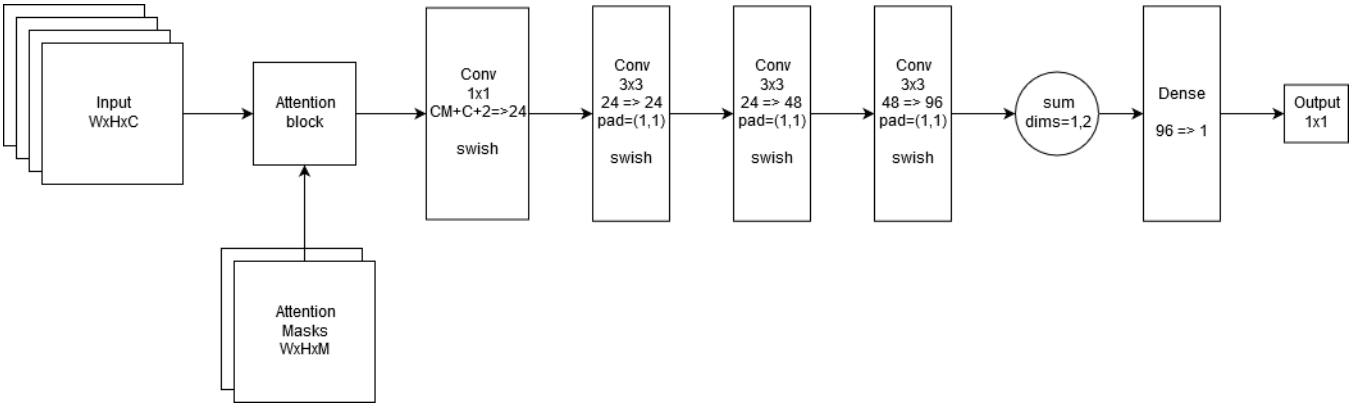


Figure 4: Heuristic network architecture

After creating M attention masks, we multiply the input data by each of the masks, thereby concatenating the results. This results in a data tensor of size $M \times C + C + 2$. We multiply each channel of the data by each attention and after concatenating the results with the original data, this multiplication is denoted in Figure 3 as "multiplication". Then, we add two last channels which are x and y coordinates for the mazes. It was shown in the (Wei et al. 2019) that for learning spacial information, it can be highly beneficial to provide coordinates for the data. Therefore, we added coordinates at the end of our data tensor. This whole computation happens in the "attention block" displayed in the Figure 3.

This created data tensor is then processed by multiple convolutional layers with same padding that keeps the width and height of the data the same throughout the whole network. After processing through the convolutional layers, aggregation in the form of a sum is performed over the first two dimensions, creating a vector of the same size as the number of channels in the last convolutional layer. Then, it is processed through a dense layer, returning one value, which is our final heuristic value for the input.

That is the top level description of the architecture. We experimented it with multiple small modifications to see if they influence the results. One modification is the number of attention layers which we mentioned earlier. The second modification which we denoted as an "attention block" states, how many times we repeat creating the attention masks and the large multiplication of these with the input data. One case is using is only once, as described above, at the beginning of the network. The other case is using five of them, one between each of the two convolutional layers. This case is displayed in Figure 5.

Experiments

Experiments in this work were conducted by training all described networks and then comparing their performance with techniques used in classical planning. To obtain each of the networks used in the planning experiments, we had to train dozens of its versions with different hyper-parameters to obtain the best possible one. Comparison of these trained networks was performed by evaluation functions.

In case of the expansion network, the adjusted hyper-parameters were number of channels in the convolutional layers, size of the convolutional kernel (in case of Sokoban), padding and number of epochs.

In case of the heuristic network, adjusted hyper-parameters were padding, number of channels in the convolutional layers, number of attention masks, number of used attention blocks and number of epochs.

Expansion Network Evaluation

Evaluation function for the expansion network is mostly used to check how accurately it can generate the possible successor states while also checking whether the network structure stays the same during the process. All the successor states in the output distribution also have to be valid and actually reachable. Same goes the other way - it is not desired to obtain any unreachable states in the output distribution. Based on these factors, we tested the trained expansion networks.

Since we use three domains which are very similar, we trained one expansion network for maze, multi-goal maze and multi-agent maze domains. Since the input dimension is different for Sokoban, we trained a separate expansion network for the Sokoban domain.

In Table 1, we can see results of the evaluation. **Wall difference** denotes the largest value assigned to a cell which is not supposed to be a wall. This means that we aim for the lowest possible values. In the evaluation table, it is obvious that wall placement is not a big problem for any of the networks.

Minimal correct step denotes the smallest value assigned to a cell which should contain an agent. The smaller the values, the less probable it is that agent can be located on the cell. In case of maze-exp-net, we can see that the value is 0.23101, which means that the least probable correct agent placement has this value. In the case of sokoban-exp-net, this value is very small, which means, that some of the possible successor states will not be discovered at all.

Maximal wrong step is analogical to the previous case. Here, we look for very small values because it denotes the highest possible wrong agent placement. In case of maze-exp-net, the value is very small, so there are no invalid successor states generated. In case of sokoban-exp-net, the value is very high, which means that invalid successor states can be generated during the search.

Therefore, the expansion network for Sokoban has not been successfully implemented and the complexity of the problem is probably higher then we first anticipated. This results in bad results for the planning experiments that use this expansion network as a state-transition function.

Heuristic Network Evaluation

Evaluation of the heuristic network is a more complicated problem since we're trying to train a monotonic function. Therefore, we took a set of optimal plans and each step of the plan was assigned a value by the heuristic network. Then we ordered the steps

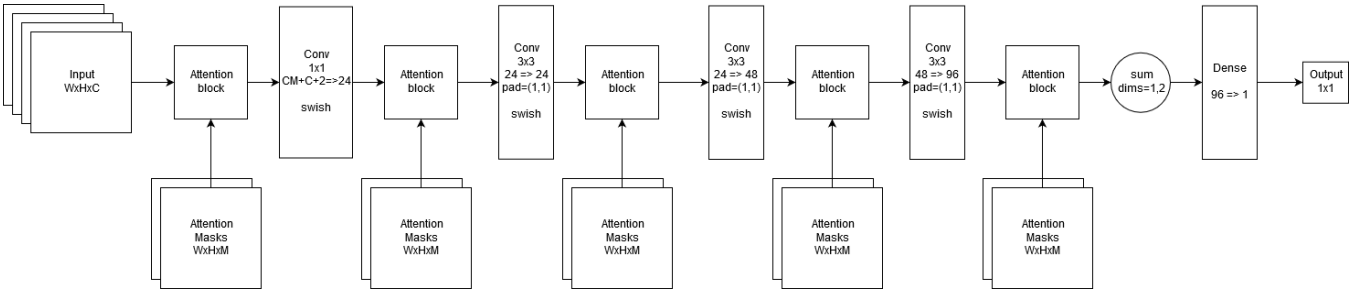


Figure 5: Heuristic network architecture using 5 attention blocks

Model	Wall difference	Minimal correct step	Maximal wrong step
maze-exp-net	1.85346e-5	**0.23101**	1.5056f-7
sokoban-exp-net	0.00241	1.33777e-11	0.96589

Table 1: Evaluation of expansion networks

by the heuristic network generated values and measured how far is the ordering from the ordering in the original optimal plan.

Planning Experiments

Based on the evaluation functions, we chose both expansion and heuristic networks which are used in the planning experiments. There are two expansion networks, one for Sokoban and one for all the three maze-related domains. There are also four heuristic networks, one for each problem domain.

For every domain, we implemented a planner with the same algorithms and the same heuristics. There are three state-space search algorithms. The first one is **greedy best-first search** (GBFS) which is guided by the computed heuristic values. State with the lowest value is always expanded first because the heuristic suggests it is closest to the goal.

Next one is the **best-first search algorithm** (BFS) with the same h value computation as in A*. This means, we don't expand the states based only on the heuristic value but we use sum of the heuristic with length of the path from the initial state to the current state.

The last one is **multi-heuristic search** (MH-GBFS) using tie-breaking (Röger and Helmert 2010). It uses a main heuristic and a second heuristic in case there's a tie between the main heuristic values. Since the order of heuristics is important in this case, we computed the experiments with all possible combinations of pairs of heuristics.

For heuristic, we can select a blind heuristic, Euclidean distance, H^{FF} heuristic (Hoffmann 2001), LM-cut (Pommerening and Helmert 2013) and the heuristic neural network. For use in multi-heuristic search, they can be arbitrarily combined.

Since we're in the field of satisficing planning, we want to compare our proposed methods against the state of the art in the field. One of the most successful planners in the community is LAMA (Richter and Westphal 2010), which uses H^{FF} heuristic together with landmarks. Therefore, we decided to implement both H^{FF} and LM-cut (which uses landmarks) and compare the approach with our heuristic network.

For each problem domain, we created 50 new problem instances which are not in any of the training data sets we used earlier. To compare the performance of all the planner configurations, we measured length of the output plans and number of expanded states during the search. There is also coverage denoted as **cvg**, which states the percentage of solved problems from the set of 50. All the experiments were executed with time limit set to 10 minutes per one problem instance on the same hardware.

Coverage Comparison We decided to compare the regular state-transition function and the expansion network by measuring coverage of the planners. Coverage tells us how many of the given problems were solved by the planner in a given time limit per problem.

In Tables 2, 3 and 4, we can see coverage for each of the state-space search algorithms used in the planning experiments. One big difference between the regular state-transition function and the expansion network is the zero percent coverage on Sokoban. The expansion network for Sokoban was not a success and we did not manage to train a well functioning expansion network for this domain as we described earlier. On the other hand, for the rest of the domains, the coverage is the same as in the case of the regular state-transition function.

Heuristic Performance Comparison In order to compare performances of all heuristics provided in the planner, including the heuristic network, we measured length of the resulting plan and number of expanded states during the search.

In the maze domain, both GBFS and BFS results show that the performance of the heuristic network is comparable to the other heuristics. Length of the resulting plans is the same in most cases and the only heuristic outperforming the heuristic network in terms of expanded states is the LM-cut heuristic. In MH-GBFS, using the heuristic network as a primary heuristic function provides us with results comparable to other heuristics. LM-cut seems to be the only heuristic which is giving a lot better results in terms of the expanded states. Results of all three search algorithms are in Figure 6.

] In the multi-goal maze domain, the results are very similar. GBFS performance is the best possible, same as H^{FF} and LM-cut. In BFS, heuristic network even outperforms both of these in the average number of expanded states. The MH-GBFS results show that heuristic network as a primary heuristic gives great results. All results are in Figure 7.

In the multi-agent domain, the number of expanded states are a lot higher for the heuristic network, as we can see in Figure 8 for all three search algorithms. This can be caused by the complexity of this problem. Simultaneous movement of multiple agents in the maze is a lot more complicated than movement of just one. Also,

	State-transition function				Expansion Network			
	maze	mg-maze	ma-maze	sokoban	maze	mg-maze	ma-maze	sokoban
blind	1	1	1	1	1	1	1	0
euclid	1	1	1	1	1	1	1	0
hff	1	1	1	0	1	1	1	0
lmcut	1	1	0.94	0	1	1	0.94	0
nn	1	1	1	1	1	1	1	0

Table 2: Coverage comparison for GBFS

	State-transition function				Expansion Network			
	maze	mg-maze	ma-maze	sokoban	maze	mg-maze	ma-maze	sokoban
blind	1	1	1	1	1	1	1	0
euclid	1	1	1	1	1	1	1	0
hff	1	1	1	0	1	1	1	0
lmcut	1	1	0.94	0	1	1	0.94	0
nn	1	1	1	1	1	1	1	0

Table 3: Coverage comparison for BFS

	State-transition function				Expansion Network			
	maze	mg-maze	ma-maze	sokoban	maze	mg-maze	ma-maze	sokoban
euclid, hff	1	1	1	0	1	1	1	0
euclid, lmcut	1	1	0.74	0	1	1	0.78	0
euclid, nn	1	1	1	1	1	1	1	0
hff, eucl	1	1	1	0	1	1	1	0
hff, lmcut	1	1	0.98	0	1	1	0.96	0
hff, nn	1	1	1	0	1	1	1	0
lmcut, eucl	1	1	0.98	0	1	1	0.96	0
lmcut, hff	1	1	0.98	0	1	1	0.96	0
lmcut, nn	1	1	0.98	0	1	1	0.96	0
nn, eucl	1	1	1	1	1	1	1	0
nn, hff	1	1	1	0	1	1	1	0
nn, lmcut	1	1	0.82	0	1	1	0.82	0

Table 4: Coverage comparison for MH-GBFS

the goals are not assigned, therefore it becomes even more complicated in terms of computing the distance estimate to a goal state.

In Sokoban domain, H^{FF} and LM-cut were not capable of solving the problems in the given time limit. The coverage for both of these heuristics is equal to zero. In the GBFS, heuristic network found the shortest plans on average. Also, in BFS, the average number of expanded states was also the lowest when using the heuristic network. We can see the results in Figure 9.

This shows us that the heuristic network is performing better in domains with one agent. Another advantage of the heuristic network is its computation time. Compared to H^{FF} and LM-cut, the computation is much faster, especially on a complicated domain like Sokoban, as reflected in the coverage comparison.

Conclusions

In this work, we have proposed replacement of two key parts of search-based automated planning algorithm by deep neural networks. One network learns the planning model from image representation of state transitions. The other network learns heuristic function from image representations of states and their distances to a goal. Such architecture allows for use of automated planning for model-free problems. Experimentally, we have shown the efficiency of such search is on par with the classical planning heuristics and therefore a viable direction for future research.

Although the work provides promising results, it is preliminary in several aspects. First of all, the heuristic function is learned from optimal plans, which makes sense only as an optimistic placeholder for cleverly generated (sub-)sequences of actions towards goals (e.g. by action backward chaining). Other future work is to design a neural network usable for variable sizes of the inputs, i.e. one neural network for different maze/Sokoban puzzle sizes.

Acknowledgements The work was supported by the Czech Science Foundation (grant no. 18-24965Y).

References

Asai, M., and Fukunaga, A. 2017. Classical planning in deep latent space: From unlabeled images to pddl (and back). In *NeSy*.

Asai, M., and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to rank for synthesizing planning heuristics. *arXiv preprint arXiv:1608.01302*.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge University Press.

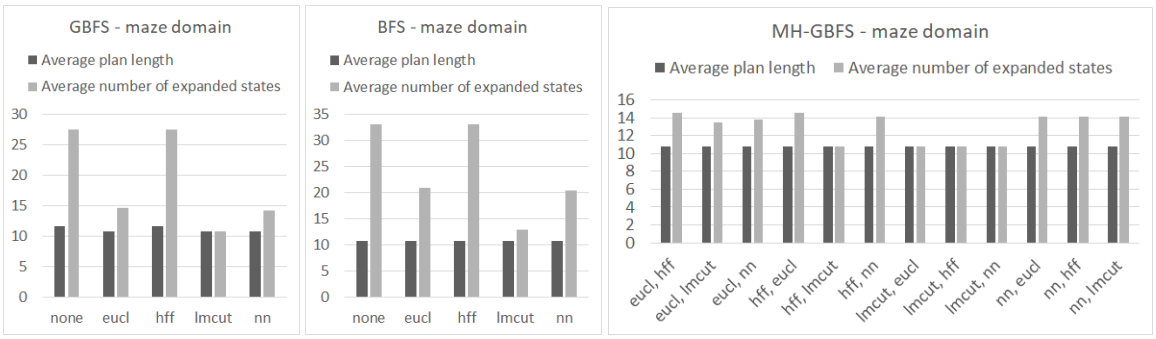


Figure 6: Performance of heuristics for maze domain in all search algorithms

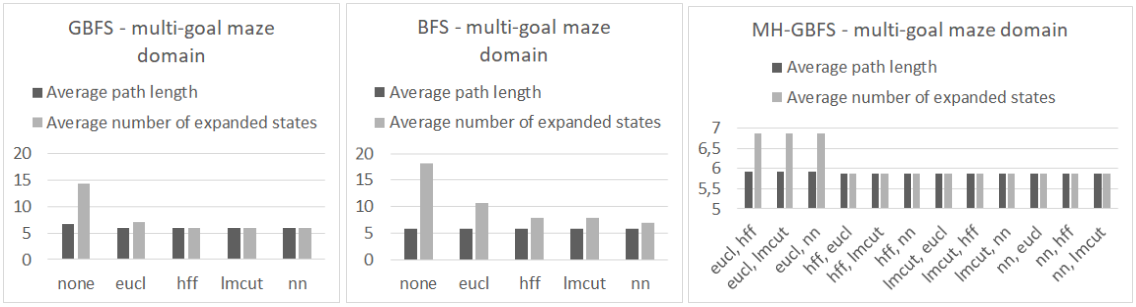


Figure 7: Performance of heuristics for multi-goal maze domain in all search algorithms

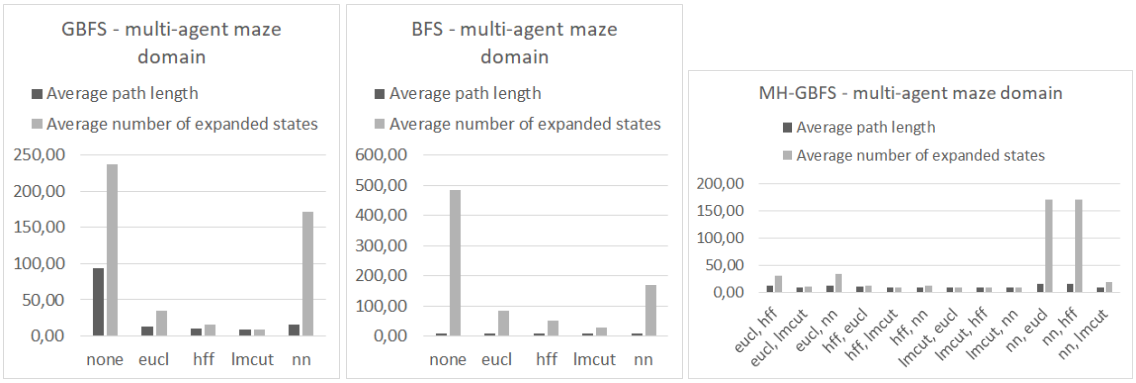


Figure 8: Performance of heuristics for multi-agent maze domain in all search algorithms

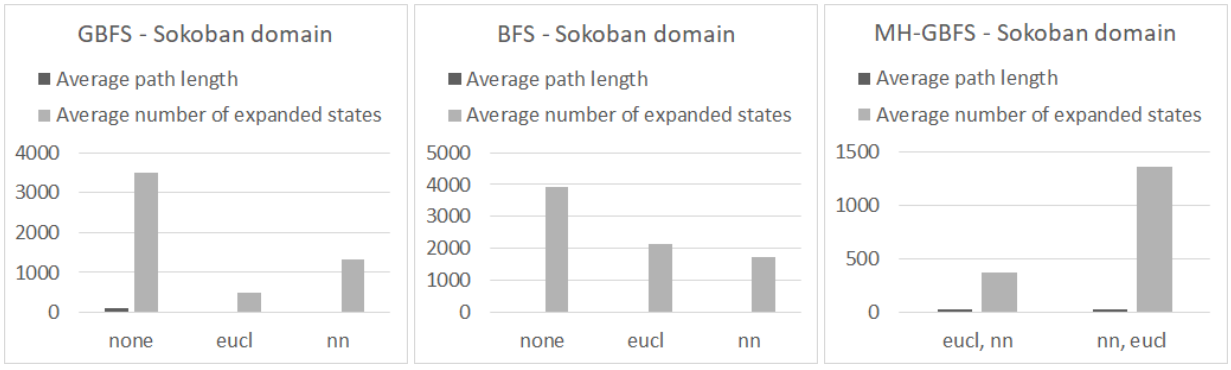


Figure 9: Performance of heuristics for Sokoban domain in all search algorithms

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2019. Learning neural search policies for classical planning. *arXiv preprint arXiv:1911.12200*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hoffmann, J. 2001. Ff: The fast-forward planning system. *AI magazine* 22(3):57–57.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature* 521(7553):436–444.

Pommerening, F., and Helmert, M. 2013. Incremental lm-cut. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Twentieth International Conference on Automated Planning and Scheduling*.

Ruder, S. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Wei, X.; Bârsan, I. A.; Wang, S.; Martinez, J.; and Urtasun, R. 2019. Learning to localize through compressed binary maps. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 10316–10324.

Generalized Planning With Deep Reinforcement Learning

Or Rivlin[1], **Tamir Hazan**[2], **Erez Karpas**[2]
[1]Technion Autonomous Systems and Robotics Program, [2]Faculty of Industrial Engineering and Management
srivlin@campus.technion.ac.il, [tamir.hazan, karpase] @technion.ac.il
Technion Israel Institute of Technology

Abstract

A hallmark of intelligence is the ability to deduce general principles from examples, which are correct beyond the range of those observed. Generalized Planning deals with finding such principles for a class of planning problems, so that principles discovered using small instances of a domain can be used to solve much larger instances of the same domain. In this work we study the use of Deep Reinforcement Learning and Graph Neural Networks to learn such generalized policies and demonstrate that they can generalize to instances that are orders of magnitude larger than those they were trained on.

Introduction

Classical Planning is concerned with finding plans, or sequences of actions, that when applied to some initial condition specified by a set of logical predicates, will bring the environment to a state that satisfies a set of goal predicates. This is usually performed by some heuristic search procedure, and the resulting plan is applicable only to the specific instance that was solved. However, a possibly stronger outcome would be to find some sort of higher level plan that can solve many instances that belong to the same domain, and thus share an underlying structure. The study of methods that can discover such higher level plans is called Generalized Planning. Generalized plans do not necessarily exist for all classical planning domains, but finding such solutions for domains in which it is possible could obviate the need to perform compute intensive search in cases where we only wish to find a goal satisfying solution. To give an example of such a generalized plan, let us consider a simplified Blocksworld domain. In this domain there are unique blocks that can be either stacked on each other or strewn about the floor, and the goal is to stack and unstack blocks such that we arrive at a goal configuration from an initial configuration. Finding a plan that does so in an optimal number of steps is generally NP-hard (Gupta and Nau 1992), but finding a plan that satisfies the goal regardless of cost can be done in polynomial time in the following manner:

1. Unstack all the blocks so that they are scattered on the floor
2. stack the block according to the goal configuration, beginning with the lower blocks

This strategy is not optimal since we might unstack blocks that are already in their proper place according to the goal specification, but it will yield a goal satisfying plan for every instance in this simplified Blocksworld domain. Such a generalized strategy can also be thought of as a policy, which raises the possibility of learning it through reinforcement learning. Machine learning theory often assumes that our training data distribution is representative of the test data distribution, thus justifying our expectation that our models generalize well to the test data. In generalized planning this is not the case, as our test instances could be much larger than the training instances, and thus far out of the training distribution. In this work we show that having the **right** inductive bias in the form of a neural network architecture could lead to models that effectively learn policies that are akin to general principles, and can solve problems that are orders of magnitude larger than those encountered during training.

Background

Classical Planning

Classical planning uses a formal description language called Planning Domain Definition Language (PDDL) (McDermott et al. 1998), derived from the STRIPS modeling language (Fikes and Nilsson 1971) to define problem domains and their corresponding states and goals. we are concerned with satisficing planning tasks, which can be defined by a set (F, O, I, G) where F is a set of propositions (or predicates) that describe the properties of the objects present in task instance and their relations, O is a set of operators (or actions types), $I \subseteq F$ is the initial state and $G \subseteq F$ is a set of goal states. each action type $o \in O$ is defined by a triple $(Pre(o), Add(o), Del(o))$, where the preconditions $Pre(o)$ is a set of predicates that must have a true value for the action to be applicable, $Add(o)$ is a set of predicates which the action turns to true upon application and $Del(o)$ is a set of predicates which the action turns false upon application. We seek to find a plan, or a sequence of actions that when applied will lead to a state s for which $G \subseteq s$, within some time limit or a predefined number of steps. Finding plans for planning tasks is often accomplished by heuristic search

methods, however in this work we focus on learning reactive planning policies that can train on instances of a specific domain and then generalize to new, unseen instances in that same domain.

Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that deals with learning policies for sequential decision making problems. RL algorithms most often assume the problem can be modelled as a Markov Decision Process (MDP), which in the finite horizon case is defined by a tuple (S, A, r, P, T, ρ), where S is the set of states, A is the set of actions, r is a reward function that maps states or state-actions to some scalar reward, P is the transition probability function such that $p(s'|s,a) = P(s',s,a)$, T is the task horizon and ρ is the distribution over initial states. The value of a policy π in the finite horizon RL problem is:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} r(s_t, a_t)\right] \tag{1}$$

Where τ are trajectories sampled from the distribution induced by the policy π, initial state distribution ρ and transition function P, and $r(s_t, a_t)$ is the reward received after taking action a_t at state s_t. The learning problem can thus be formalized as an optimization problem, in which we wish to find a policy that maximizes the objective:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2}$$

We focus on stochastic policies with parameters θ, which map states and actions to probabilities, such that $p(a|s) = \pi_\theta(a|s)$, and use proximal policy optimization as the learning algorithm.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) (Schulman et al. 2017) is a policy gradient based algorithm that seeks to better exploit the data gathered during the learning process, by performing several gradient updates on the collected data before discarding it to collect more. In order to avoid stability issues that could arise from large policy updates, PPO uses a special clipped objective to discourage divergence between the current policy and the data collection policy, by defining the following optimization problem:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|} \sum_{\tau\in D_k} \sum_{t=0}^{T} min\left(g_1, g_2\right) \tag{3}$$

$$g_1 = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A(a_t|s_t)^{\pi_{\theta_k}} \tag{4}$$

$$g_2 = clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right) A(a_t|s_t)^{\pi_{\theta_k}} \tag{5}$$

Where $clip$ is a function that clips the values of its input to be between the specified minimum and maximum values, π_θ is the policy we are currently optimizing, π_{θ_k} is the policy used to collect the data (before updating) and $A(a_t|s_t)^{\pi_{\theta_k}}$ is the advantage of the action given the current state and parameters:

$$\hat{A}(a_t|s_t)^{\pi_{\theta_k}} = R_t - \hat{V}_{\phi_k}(s) \tag{6}$$

Where the dependency on the action a comes from the empirical return to-go $R(s_t)$, which depends on the specific actions that were taken by the policy. $V_{\phi_k}(s)$ is a state value predicted by some function approximator with parameters ϕ_k, obtained at each iteration by solving:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|} \sum_{\tau\in D_k} \sum_{t=0}^{T} \left|\hat{V}_\phi(s_t) - R_t)\right|^2 \tag{7}$$

Learning Generalized Policies

State Representation

We chose to represent the states in our framework as graphs, with features encoding the properties and relations between the objects in a given state. Our framework operates on problem domains specified by the PDDL modeling language, in which problem instances are defined by a list of objects and a list of predicates that describe the properties of these objects and the relations between them at the current state. We limit ourselves to domains for which predicates have an arity of no more than two, which is not a significant limitation since higher arity predicates can in many cases be decomposed to several lower arity predicates. Our graphs are composed of global features, node features and edge features, as in (Battaglia et al. 2018). We denote our global features U, our nodes V and our edges E. Global features represent properties of the problem instance or entities that are unique for the domain, such as the hand in the Blocksworld domain, and are determined by the 0-arity predicates of the domain. Node features represent properties of the objects in the domain, such as their type, and are determined by the 1-arity predicates. Lastly, edge features represent relations between the objects and are determined by the 2-arity predicates.

When producing a graph representation of a PDDL instance state, a complete graph is produced with a node for each object in the state. For each predicate in the state, the corresponding feature is assigned a binary value of 1, and all other features are assumed to be false with a value of 0. In order to include the goal configuration in the input to the neural network, the goal predicates are treated almost as if they were another state-graph, and the two graphs are concatenated together to form a single representation for the state-goal. The difference between state graphs and goal graphs, is that in the goal graphs a 0 valued feature means that it contributes no goal, and in the state graph a 0 valued feature means that the predicate is assigned a false value.

The classical planning domains used throughout this work are deterministic and Markovian, meaning that the current state holds all the required information to solve the problem optimally. Despite this property, we found that adding past states in addition to the current one helps the learning process and improves the generalization capability to larger instances. While this is not strictly essential, our experiments suggest that this step helps the policy mitigate "back-and-forth" behavior to some extent, and this is especially helpful on the

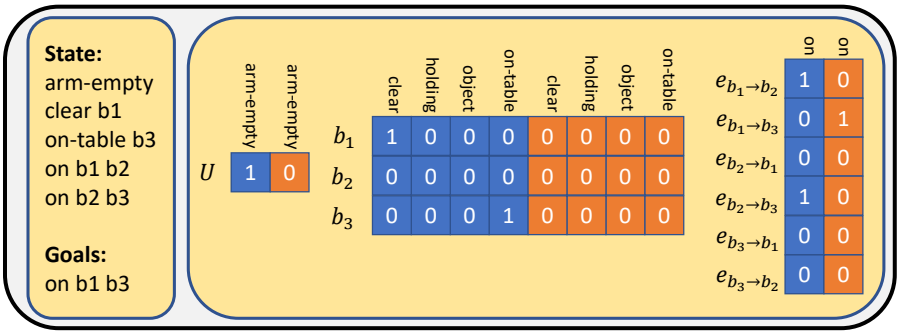


Figure 1: Example state-goal graph from the Blocksworld domain. The left side shows the PDDL description of the state-goal, and the right side shows the graph representation of the same state-goal. Blue represents state information and orange represents goal information

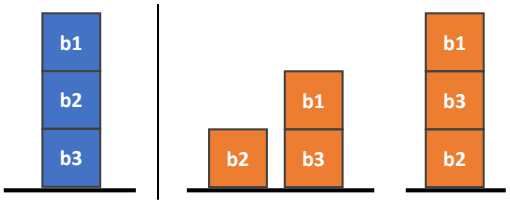


Figure 2: Visualization of the state from the previous figure (blue) and possible goal-satisfying configurations (orange)

larger instances where the policy is more prone to make mistakes and then attempt to correct them. Adding this history is straight-forward; we simply concatenate the graphs for the K previous states and current state, and then concatenate the goal graph as mentioned previously. We tested several such history horizons, and found that adding only the last state results in overall best performance and generalization. An example of a state-goal graph from the Blocksworld domain can be seen in figures 1 and 2, showing an instance with 3 blocks.

Graph Embedding

In order to learn good policies using the graph representations of state-goals we first use a Graph Neural Network (GNN) to embedd the node, edge and global features of the graph in respective latent spaces. The GNN performs message passing between the different components of the graph, allowing useful information to flow. We use two different types of GNN blocks, each enforcing a different style of information flow within the graph and thus more suited to certain problem domains than others. In both of these types the update order is similar and takes the following common form:

1. Edges are updated using the previous edges and the "origin" nodes of those edges.
2. Nodes are updated using the previous nodes, the incoming updated edges and the global features.
3. Globals are updated using the previous globals and the aggregation of the updated nodes.

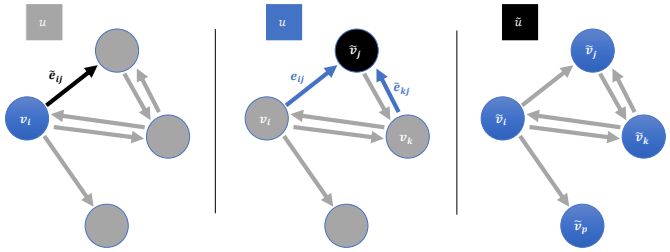


Figure 3: Flow of information within the graph network block. The black piece is the component being updated at each step, the additional information used to update that component is in blue color, and the gray information is not used. On the left: updating the edges. In the middle: updating the nodes. On the right: updating the global features

The first block type we used is similar to the one described in (Battaglia et al. 2018) which we name accordingly **Graph Network block** (GN block). Mathematically, this block performs the following operations:

$$\tilde{e}_{ij} = \phi(W^e[e_{ij}, v_i] + b^e) \tag{8}$$

$$h_{ij} = \phi(W_1^v[V_i, \tilde{e}_{ij}] + b_1^v) \tag{9}$$

$$m_i = \psi(h_{ij}) \tag{10}$$

$$\tilde{v}_i = \phi(W_2^v[m_i, u] + b_2^v) \tag{11}$$

$$\tilde{u} = \phi(W^u \frac{1}{|v|} \sum_{i \in v} \tilde{v}_i + b^u) \tag{12}$$

In the above notation, ϕ is a nonlinearity such as Rectified Linear Unit, ψ is a node-wise max-pooling operation over messages h and W, b are respective weight matrices and biases. In the GN block, nodes receive messages from their neighbouring nodes indiscriminately, which works well to

propagate general information across the graph but makes it harder to transfer specific bits of information when needed.

The second type of block was designed to address that shortcoming of the GN block, and for that purpose was endowed with an attention mechanism. We named the second block **Graph Network Attention block** (GNAT block), and unlike the Graph Attention Network of (Veličković et al. 2017), it uses an attention mechanism similar to the Transformer model of (Vaswani et al. 2017). This block performs the following operations:

$$\tilde{e}_{ij} = \phi(W^e[e_{ij}, v_i] + b^e) \tag{13}$$

$$h_i = \phi(W_1^v v_i + b_1^v) \tag{14}$$

$$k_i = \phi(W^k v_i + b^k) \tag{15}$$

$$q_{ij} = \phi(W^q \tilde{e}_{ij} + b^q) \tag{16}$$

$$\alpha_{ij} = \frac{\exp(k_i^T q_{ij})}{\sum_p \exp(k_i^T q_{ip})} \tag{17}$$

$$m_i = \varphi(\alpha_{ij} \odot \tilde{e}_{ij}) \tag{18}$$

$$\tilde{v}_i = \phi(W_2^v[h_i, m_i, u] + b_2^v) \tag{19}$$

$$\tilde{u} = \phi(W^u \frac{1}{|v|} \sum_{i \in v} \tilde{v}_i + b^u) \tag{20}$$

In the above notation, φ is a node-wise summation operation, $\odot$ is the Hadamard product and W, b are respective weight matrices and biases. As mentioned above, this type of block allows certain bits of information to travel in the graph in a more deliberate manner, by endowing the nodes with the ability to focus on specific messages. When constructing our GNN model, we can stack several blocks of these types (and combinations of them) to attain a deeper graph embedding capacity. In most of our experiments we used two blocks, either two successive GN blocks, or a GNAT block followed by a GN block. Each configuration excelled at a different group of problems as we will show in the experiments section.

Policy Representation

Unlike common reinforcement learning benchmarks where the set of actions is fixed and can be conveniently handled by standard neural network architectures, in classical planning problems the set of actions is state dependent and varies in size between states. In PDDL, each domain description defines a set of action types that can be instantiated by grounding those action types to the state. Each action type receives a set of arguments, and in order to be applicable the arguments of the action must conform to a set of pre-conditions. For example, the Blocksworld domain has an action type called "pick-up" which gets a single block object as an argument. This block must be "clear", "on-table" and the "arm-empty" property must be true for the action to be applicable. All blocks that comply with these pre-conditions can be picked up, and represent a unique action. In addition to pre-conditions, each action type also has effects which are caused to the states upon application of the action. Some of these effects could be positive (certain predicates of the state will take a true value) and some negative (predicates will assume a false value).

At each step of planning, the successor-state generator gives the current state and a list of applicable actions. In order to represent the actions in a meaningful way that enables learning a policy over them, we chose to describe the actions in terms of their effects, since these are the essential components needed to make decisions. Since the successor-state generator provides the agent with all the legal actions at each step, we ignored the preconditions (all legal actions satisfy the pre-conditions). Each action is composed of several effects, each concerning a different aspect of the state, and are either positive or negative. The effects are clustered together based on their type (global effect, node effect or edge effect), and are represented as a concatenation of the embedding of the respective component and a one-hot vector describing which predicate is changed and if it is positive or negative. This one-hot vector is in the dimension of corresponding input component (d_v for node effects for example) and contains either 1 for positive effects or -1 for negative effects at the appropriate predicate location. Each effect is transformed by a multi layered perceptron (MLP) according the its type and then the transformed effects are scattered back to their origin actions. The effects of each action are aggregated together to form a single vector representation of that action, which is fed eventually to the policy neural network. Figure 4 illustrates the process of action representation.

The final policy is a MLP that outputs a single scalar for each action, and these scalars are then normalized by a softmax operation to get a discrete distribution over the actions. In addition, another MLP takes the final global feature embeddings of the graph and outputs the predicted value of the state, to be used for advantage estimation in the RL algorithm.

Training Procedure

Since the focus of this work was finding feasible plans, we chose to model our problem as a sparse reward problem with a binary reward. If the agent satisfies all the goals within a predefined horizon length, it gets a reward of 1, and if not it gets no reward. To determine an appropriate time limit we used the commonly used **hff** heuristic (Hoffmann and Nebel 2001), which solves a relaxed version of the problem in linear time (the relaxed problem has no negative effects). We take the length of the relaxed plan and multiply it by a constant factor of 5 to get the horizon length.

To train our policy we chose to use Proximal Policy Optimization (PPO) (Schulman et al. 2017) for its simplicity and good performance. To handle the problem of sparse rewards we initially experimented with using Hindsight Experience Replay DQN (Andrychowicz et al. 2017) due to its demonstrated ability to tackle sparse goal reaching problems, but found that it introduced a lot of bias and resulted in unsatisfactory performance. To allow our policy to learn from a sparse binary reward, we resorted to a simpler method; we generated each training episode from a distribution over instance sizes, which includes sizes small enough to be occasionally solved by a randomly initialized policy. Doing this allows the policy to progress to eventually solve all the instance sizes in the distribution, without the need for a manual tuning of a curriculum. Although setting this distribution needs to be done manually, we found it very easy and quick to do by simple

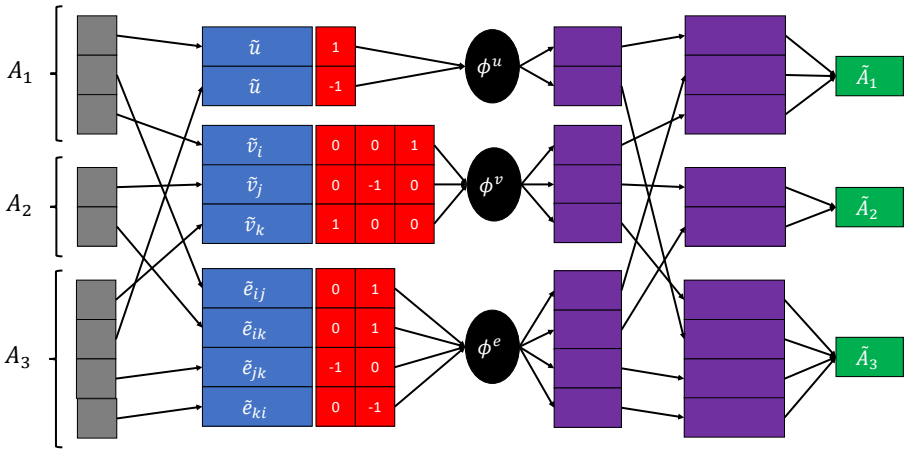


Figure 4: Illustration of the action embedding process. From left to right: gray blocks represent original action effects, blue blocks represent embedding of appropriate graph elements clustered (according to globals, nodes and edges), red blocks represent the one-hot effect type vectors, black ellipses represent MLPs, purple blocks represent effect embeddings clustered and then scattered back to their origin actions, and green blocks represent final action embeddings

trial and error with a random untrained neural network.

We made several small adjustments to the standard PPO algorithm which improved performance in our case. Many RL algorithms implementations roll out the policy for a fixed number of steps before updating the model parameters, often terminating episodes before completion in the process and using methods such as Generalized Advantage Estimation (Schulman et al. 2015) and bootstrapping value estimations to estimate the returns as in (Mnih et al. 2016). We found these elements to add unwanted bias to our learning process, and instead rolled out each episode until termination, using empirical returns instead of bootstrapped value estimates to compute advantages. We also found that using many roll-outs and large batch sizes helped stabilize the learning process and resulted in better final performance, and so we performed 100 episode roll-outs and used the resulting data to update the model parameters at each iteration of the learning algorithm.

Planning During Inference

To improve the ability of our generalized policies to use additional time during test, we use them within a search algorithm, as was done in many other works such as (Silver et al. 2016), (Anthony, Tian, and Barber 2017). This type of synthesis gained great success in zero sum games such as Go and Chess (Silver et al. 2017), where a deep neural network policy was used in conjunction with a Monte Carlo Tree Search algorithm, which prompted other authors to do the same even for non-game problems (Abe et al. 2019). We take a different approach and design our search algorithm specifically for the case of deterministic planning problems with a strong reactive policy. Our algorithm is based on the classic Greedy Best First Search (GBFS) algorithm, but augments it in several key ways. In standard GBFS, a search tree is constructed from the root node, and at each iteration, the node with the best heuristic estimate is extracted from the open list, expanded and its child nodes added to the open list, and this procedure is repeated until a goal node is found or until time is out. Our algorithm, which we name GBFS-GNN, performs a similar procedure, but uses the policy and value functions to compute a heuristic value for each node, and performs a full roll-out for each expanded node. The offspring of the expanded node are added to the open list, but the rest of the nodes encountered during the roll-out are not, to avoid rapid memory consumption growth in large problems. Each node in our search tree represents a state-action pair, and we use the following heuristic estimate for each node:

$$g(s,a) = \frac{\pi(a|s) \cdot V(s)}{1 + H(\pi(\cdot|s))} \tag{21}$$

In this equation, $g(s,a)$ is the heuristic estimate of the state-action, $\pi(a|s)$ is the probability of action a under our policy π, $V(s)$ is the estimated state value according the the critic part of our neural network policy, and $H(\pi(\cdot|s))$ is the entropy of the policy's distribution over actions at state s. Figure 5 illustrates our search algorithm.

Related Work

Learning to plan has been an active topic of research for many years, with different methods attempting to learn different aspects of a complete solver. Some works attempted to learn heuristic values of states for specific domains using features generated by other domain independent heuristics, such as (Yoon, Fern, and Givan 2006), which learns heuristic values by regression. more recent works such as (Garrett, Kaelbling, and Lozano-Pérez 2016) learn to rank successor states by using RankSVM (Joachims 2002). These types of methods do not explicitly use the state or goal information from the problem description, but rather learn using hand crafted features, and in addition do not learn an explicit planning policy over the available actions. Contrary to this, our methods learns

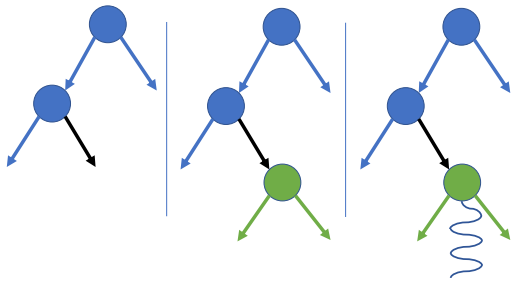


Figure 5: Illustration of the search procedure. From left to right: a state-action edge with the highest heuristic estimate is chosen (black), the action is applied to generate the successor state with its child edges (green) and finally a full roll-out is performed from the successor state using the policy (purple spiral)

planning policies over explicit states and goals, that directly choose actions to take.

Other works such as (Tamar et al. 2016), (Groshev et al. 2018) and (Guez et al. 2019) learn an explicit planning policy over actions, using the actual state of the problem as input and a deep convolutional neural network, but rely on having a visual representation of the problem. This limits their usage to domains where a visual representation is available. Another limitation is that (Tamar et al. 2016) and (Groshev et al. 2018) rely in addition on successful plans generated by a planning algorithm and learn policies using imitation learning, while (Guez et al. 2019) use reinforcement learning for this purpose. Our work does not rely on visual representations or successful plans generated by planning algorithms, but learns directly from a PDDL representation of the problem by trial and error via deep reinforcement learning.

Some works have begun to study the use of graph representations of states and the use of different kinds of graph neural networks for the task of learning policies or heuristics. In (Toyer et al. 2018) the authors proposed a unique kind of neural network called Action Schema Network (ASNet) which consists of alternating action layers and proposition layers to learn planning policies. They represent their state as a graph in which objects and actions are connected and propagate information back and forth to finally output a probability over actions. They train their ASNets by imitating plans generated by other planners, and augment the input with domain independent heuristic values to improve performance. In their experiments, they focus mainly on stochastic planning problems and demonstrate that their trained policies can generalize to larger instances than trained on. A limitation of ASNet is their fixed receptive field, that limits their ability to reason over long chains, which our work does not share.

In a recent paper, (Shen, Trevizan, and Thiébaux 2019) propose an extension of (Battaglia et al. 2018) to hypergraphs, and use it to learn heuristics over hypergraphs that represent the delete relaxation states of planning problems. They use supervised learning of optimal heuristic values generated by a planning algorithm, and then use the resulting neural network as a heuristic function within a search algorithm. In contrast to this, our method focuses on learning policies, which can be more time-efficient during evaluation since a single forward pass over the neural network is needed to make a decision at each state. Using heuristic estimates requires estimating all the successor states of a state in order to choose the best action, which could potentially increase run-time. Another difference is that our work operates directly on states instead of delete relaxations, which might limit the power of heuristics since some information is omitted. Overviews of older methods of learning to plan can be found in (Minton 2014), (Zimmerman and Kambhampati 2003) and (Fern, Khardon, and Tadepalli 2011).

Experiments

Domains

We evaluate our approach on five common classical planning domains, Chosen from the IPC planning competition collection of domain generators that have predicates of arity no larger than 2:

- Blocksworld (4 op): A robotic arm must move blocks from an initial configuration in order to arrange them according to a goal configuration.
- Satellite: A fleet of satellites must take images of locations, each with a specified type of sensor.
- Logistics: Packages must be delivered to target locations, using airplanes and trucks to move them between cities and locations.
- Gripper: A twin-armed robot must deliver balls from room A to room B.
- Ferry: A ferry must transport cars from initial locations to designated target locations.

What these five domains have in common is that simple generalized plans can be formulated for them, which are capable of solving arbitrarily large instances. We wish to demonstrate that our method is capable of producing policies that solve much larger instances than those they were trained on, thus automatically discovering such generalized plans. Some domains are easier than others, and in cases where the generalized plan is very easy to describe we often witnessed that the policy generalizes very successfully. For example, the Gripper domain has a very simple strategy (Grab 2 balls with each trip to room B) and indeed our neural network learns the optimal strategy and usually still performs optimally even for instances with hundreds of balls. To demonstrate that our policies indeed generalize well, we trained them on small instances and used both small and large instances for evaluation.

- For the Blocksworld domain we trained our policy on instances with 4 blocks, and evaluated on instances with 5-100 blocks.
- For the Satellite domain we trained our policy on instances with 1-3 satellites, 1-3 instruments per satellite, 1-3 types of instruments, 2-3 targets, and evaluated on instances with

1-14 satellites, 2-11 instruments per satellite, 1-6 types of instruments and 2-42 targets.

- For the Logistics domain we trained our policy on instances with 2-3 airplanes, 2-3 cities, 2-3 locations per city, 1-2 packages, and evaluated on instances with 4-12 airplanes, 4-15 cities, 1-6 locations per city and 8-40 packages.
- For the Gripper domain we trained our policy on instances with 3 balls, and evaluated on instances with 5-200 balls.
- For the Ferry domain we trained our policy on instances with 3-4 locations, 2-3 cars and evaluated on instances with 4-40 locations and 2-120 cars.

Experimental setting

For training our policies, we rely on having instance generators to produce random training instances, since our method requires large amounts of training data. All policies are trained for 1000 iterations, each with 100 training episodes and up to 20 gradient update steps. Experiments are performed on a single machine with a i7-8700K processor and a single NVIDIA GTX 1070 GPU. We used the same training hyperparameters for all five domains, but slightly varying neural network models. We used a hidden representation size of 256 and ReLU activations, a learning rate of 0.0001, a discount factor of 0.99, an entropy bonus of 0.01, a clipping ratio of 0.2 and a KL divergence cutoff parameter of 0.01. For the Blocksworld and Gripper domains we used a two layer GNN with both layers of the GN block type, and for the Satellite, Ferry and Logistics domains we used a two layer GNN with a GNAT block followed by a GN block. Our code was implemented in Python and our neural networks and learning algorithm were implemented using PyTorch (Paszke et al. 2019).

Baseline

We focus in our evaluation on solving large instances of generalized planning domains and compare our method with a classical planner. Other learning based methods either had no available code by the time this work was written (such as (Shen, Trevizan, and Thiébaux 2019)) or were inherently limited in scaling to the large problems (for example (Toyer et al. 2018)), so we opted for a more general baseline in the form of a classical planner, which can scale to large problems given enough time and memory. We compare against fast-downward (Helmert 2006), which is a state of the art framework. Our approach uses Pyperplan as the model and successor state generator, which is a Python based framework. We use the LAMA-first configuration as the setup for fast-downward, as it is a top performing competitive satisficing planning algorithm.

Evaluation Metrics

Since our work is focused on satisficing planning, we use success rate as our main metric. We run both our GBFS-GNN and fast-downward on a set of 50 held out evaluation instances per domain, and run each method for a fixed time limit of 600 seconds per instance, we then plot the success rate of each method against the time limit and against the number of expanded states to see how each method scales with given computation. The evaluation instances are generated according to a wide distribution such that both small and large instances are sampled. During GBFS-GNN planning we count all the states visited including during rollouts.

Results

We now present our results. Figure 6 shows a comparison between our method and fast-downward for the five domains we used in our experiments. The plots show success rate as a function of number of expanded states, and demonstrate that our method indeed scales favourably compared to the classical planner on 4 of the 5 domains. In fact, on the 4 domains where our policies generalized well, GBFS-GNN required very little to no search. In these domains, a solution can be found by just greedily following the policy in all but the hardest instances. Our search algorithm builds on this generalization capability and uses a small number of full policy roll-outs while searching.

In figure 7 we present a comparison between our method and fast-downward, plotting success rate against given runtime. We can see that even though fast-downward has a highly optimized C++ implementation and uses sophisticated modeling tools to efficiently solve planning problems, our method overcomes it in one domain (Blocksworld) and closely matches it on three others. Despite GBFS-GNN using a successor state and legal action generator that is orders of magnitude slower than that of fast-downward, our method's generalization capability makes it competitive with state of the art implementations of classical planners.

An obvious exception concerning the generalization performance of our method is the Logistics domain. Our policy successfully achieved good performance on the training instances but failed to generalize to much larger instance sizes, and consequently was vastly outperformed by fast-downward on that domain. We hypothesize that unlike the other domains, the Logistics domain contains a tighter coupling between the different objects in each instance. In the Satellite domain for example, calibrating an instrument or imaging a target does not interfere with other satellites, in the sense that the policy can have multiple "half-baked" goals and switch between them without interference. This is not possible in the Logistics domain, as all the packages share the trucks and airplanes, and moving a specific truck to pick up a package might interfere with another package that was meant to be picked up in another location. Different graph neural network architectures could perhaps encourage the policy to remain "fixed" on a single goal until its satisfaction before moving to another, thus possibly overcoming the issue with the Logistics domain and other similar types of problems.

Conclusion and Future Work

In this work we studied the ability of graph neural networks and deep reinforcement learning algorithms to learn generalized planning policies that can solve instances much larger than those encountered during training, in effect learning principles that generalize well. Unlike some other approaches,

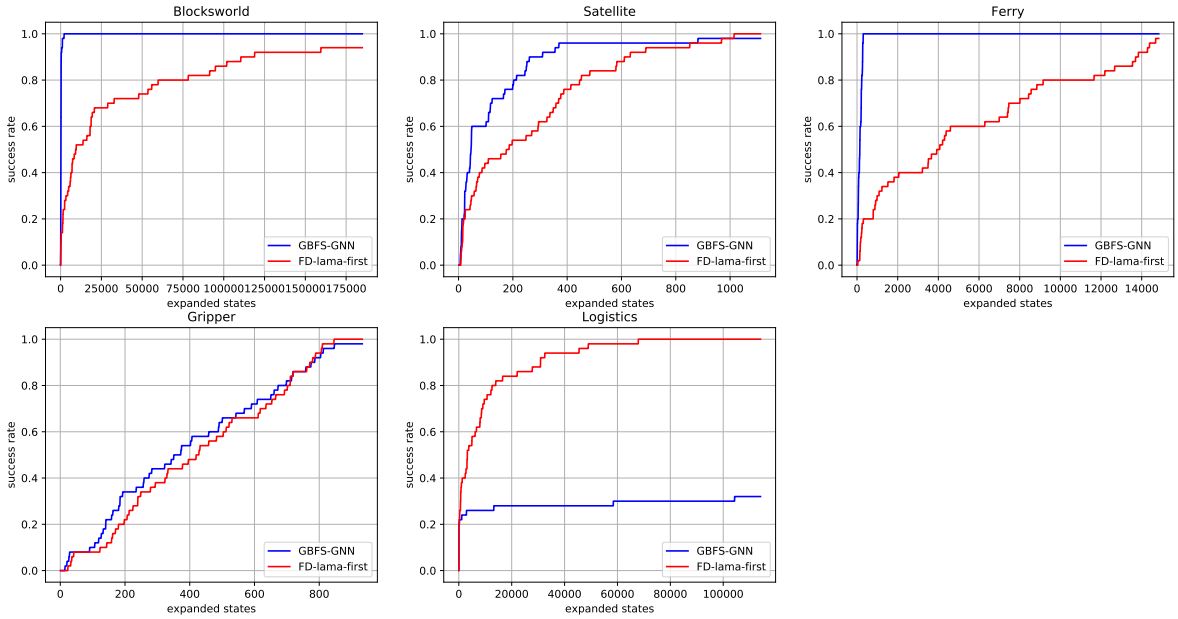


Figure 6: These plots compare success rate against number of expanded states for the various domains used in the evaluation

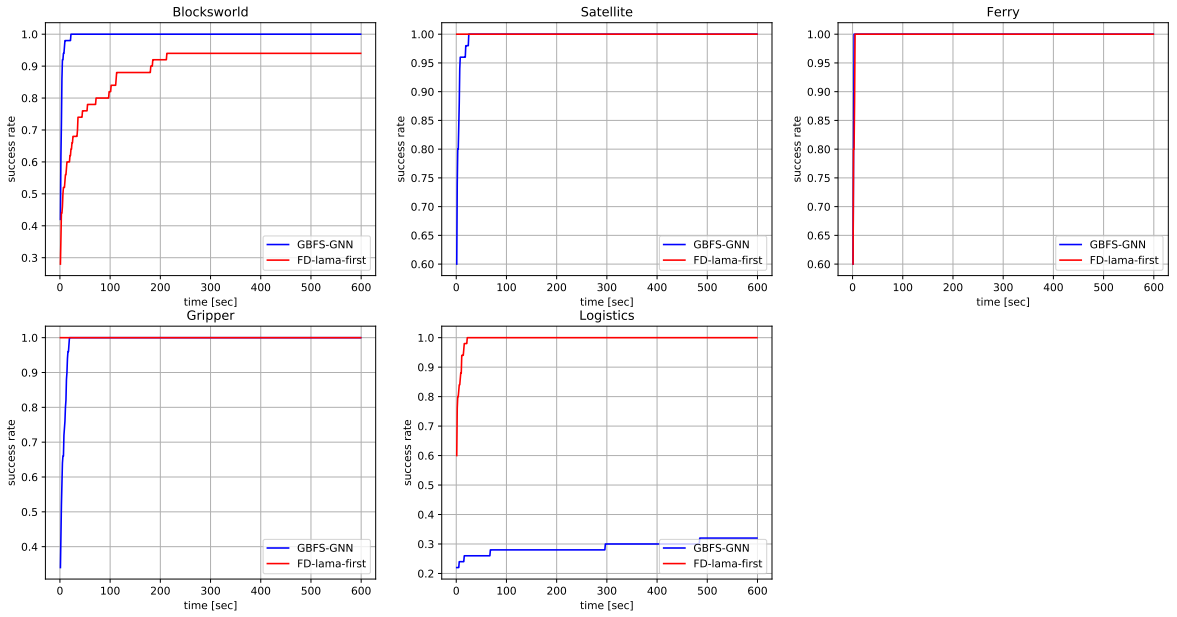


Figure 7: These plots compare success rate against run time for the various domains used in the evaluation

our method does not rely on optimal solutions provided by existing planners, nor on heuristics to boost performance. We further introduce GBFS-GNN, a search algorithm that exploits the availability of high performing reactive policies to quickly find solutions to very large instances. Our policies are learned from scratch via reinforcement learning, and combined with GBFS-GNN achieve performance that surpasses highly optimized implementations of state of the art planners in terms of expanded states, and is on par in terms of run-time. Directions for future work include studying how specific mechanisms in graph neural networks architectures relate to the emergent generalization behaviour on different domains, studying the effect of different reinforcement learning algorithms on generalization and perhaps exploring

regularization schemes on the policy training procedure that might encourage better generalization.

References

Abe, K.; Xu, Z.; Sato, I.; and Sugiyama, M. 2019. Solving np-hard problems on graphs by reinforcement learning without domain knowledge. *arXiv preprint arXiv:1905.11623*.

Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Abbeel, O. P.; and Zaremba, W. 2017. Hindsight experience replay. In *Advances in neural information processing systems*, 5048–5058.

Anthony, T.; Tian, Z.; and Barber, D. 2017. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, 5360–5370.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* 84(1-2):81–107.

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to rank for synthesizing planning heuristics. *arXiv preprint arXiv:1608.01302*.

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *2018 AAAI Spring Symposium Series*.

Guez, A.; Mirza, M.; Gregor, K.; Kabra, R.; Racanière, S.; Weber, T.; Raposo, D.; Santoro, A.; Orseau, L.; Eccles, T.; et al. 2019. An investigation of model-free planning. *arXiv preprint arXiv:1901.03559*.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2-3):223–254.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 133–142.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Minton, S. 2014. *Machine learning methods for planning*. Morgan Kaufmann.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 1928–1937.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 8024–8035.

Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2019. Learning domain-independent planning heuristics with hypergraph networks. *arXiv preprint arXiv:1911.13101*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354–359.

Tamar, A.; Wu, Y.; Thomas, G.; Levine, S.; and Abbeel, P. 2016. Value iteration networks. In *Advances in Neural Information Processing Systems*, 2154–2162.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.

Yoon, S. W.; Fern, A.; and Givan, R. 2006. Learning heuristic functions from relaxed plans. In *ICAPS*, volume 2, 3.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine* 24(2):73–73.

Reinforcement Learning of Risk-Constrained Policies in Markov Decision Processes (Extended Abstract)

Tomáš Brázdil[1], **Krishnendu Chatterjee**[2], **Petr Novotný**[1], **Jiří Vahala**[1]
[1]Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbrazdil, petr.novotny, xvahala1}@fi.muni.cz
[2]Institute of Science and Technology Austria, Klosterneuburg, Austria
Krishnendu.Chatterjee@ist.ac.at

Abstract

Markov decision processes (MDPs) are the standard model of sequential decision making under stochastic uncertainty. A classical optimization criterion for MDPs is to maximize the expected discounted-sum payoff, which ignores low probability catastrophic events with highly negative impact on the system. On the other hand, risk-averse policies require the probability of undesirable events to be below a given threshold, but they do not account for optimization of the expected payoff. We consider MDPs with discounted-sum payoff and with failure states which represent catastrophic outcomes. The objective of *risk-constrained* planning is to maximize the expected discounted-sum payoff among risk-averse policies that ensure the probability to encounter a failure state is below a desired threshold. Our main contribution is an efficient risk-constrained planning algorithm that combines UCT-like search with a predictor learned through interaction with the MDP (in the style of AlphaZero) and with a risk-constrained action selection via linear programming. We demonstrate the effectiveness of our approach with experiments on classical MDPs from the literature, including benchmarks with an order of 10^6 states.

This extended abstract summarizes results presented in the paper *Reinforcement Learning of Risk-Constrained Policies in Markov Decision Processes* published at AAAI'20.

1 Introduction & Problem Statement

MDPs. We consider *Markov decision processes* (MDPs) with discounted-sum payoff, a standard model of probabilistic decision-making (Puterman 1994). Formally and MDP consists of: a finite set $\mathcal{S}$ of *states;* a finite alphabet $\mathcal{A}$ of *actions;* a *probabilistic transition function* δ that given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$ returns the probability distribution $\delta(s,a)$ over the successor states; a *reward function* $rew : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$; and a *discount factor* γ.

Fixing some initial state s_0, the interaction with an MDP starts in s_0 and proceeds sequentially through a *policy* π, a function which acts as a blueprint for selecting actions, producing longer and longer *history* of actions and observations. We denote by $\mathbb{P}^\pi(E)$ the probability of an event E under policy π, and by $\mathbb{E}^\pi[X]$ the expected value of a random variable X under π.

Expectation optimization and risk. In the classical studies of MDPs with discounted-sum payoff, the objective is to obtain policies that maximize the *expected* payoff. Formally, the expected payoff of a policy π is the number $Payoff(\pi) = \mathbb{E}^\pi[\sum_{i=0}^{\infty} \gamma^i \cdot rew(s_i, a_i)]$, where s_i, a_i are the current state and the action selected in step i. However, the expected payoff criterion ignores the possible presence of low probability failure events that can have a highly negative impact.

Motivating scenarios. In scenarios such as a robot exploring an unknown environment, a significant damage of the robot ends the mission. Of course, the policy which never moves the robot would be likely safe in such a scenario, but goes against the robot's primary objective of an effective exploration. Instead, the operator can set a *risk threshold,* i.e. the probability of the robot's destruction that is acceptable under given operational parameters. The goal is to ensure effective exploration while keeping the risk of destruction below the threshold, which naturally gives rise to the risk-constrained planning problem we consider.

In the pure expectation-optimizing framework, we might attempt to encode the risk-taking aspect directly into the reward function, e.g. by stipulating that entering a failure state incurs a large negative penalty. However, the risk-taking aspect of the resulting policy is then very sensitive to the penalty variations, demanding an elaborate tuning of the penalties to achieve a desired behaviour (see also (Undurti and How 2010) for a critical discussion of reward-function engineering in risk-constrained scenarios). In contrast, we aim to achieve an explicit control over the risk taken by a policy, decoupling the risk-taking aspect from the expected payoff optimization.

Problem statement. Given an MDP, we fix a set $F \subseteq \mathcal{S}$ of *failure states*. A *risk* of a policy π is then the probability that a failure state is encountered:

$$Risk(\pi) = \mathbb{P}^\pi\Big(\bigcup_{i=0}^{\infty} \{s_i \in F\}\Big).$$

We assume that each $s \in F$ is a *sink,* i.e. $\delta(s,a)(s) = 1$ and $rew(s,a) = 0$ for all $a \in \mathcal{A}$. Hence, F models failures after which the agent has to cease interacting with the environment (e.g. due to being destroyed).

The risk-constrained planning problem is defined as follows: given an MDP $\mathcal{M}$ and a *risk threshold* $\Delta \in [0,1]$, find

a policy π which maximizes $Payoff(\pi)$ subject to the constraint that $Risk(\pi) \leq \Delta$. If there is no *feasible* policy, i.e. a policy s.t. $Risk(\pi) \leq \Delta$, then we want to find a policy that minimizes the risk and among all such policies optimizes the expected payoff.

The risk-constrained planning problem can be formulated as a special case of constrained MDPs (Altman 1999) (see (Brázdil et al. 2020) for an overview of related work). Our main contribution is a new reinforcement-learning algorithm for risk-constrained planning.

2 Our Contribution

We present RAlph (a portmanteau of "Risk" and "Alpha"), an online algorithm for risk-constrained planning. Inspired by the successful approach of AlphaZero (Silver et al. 2017; 2018), RAlph combines a UCT-like exploration of the *history tree* $\mathcal{T}$ of the MDP with evaluation of the leaf nodes via a suitable *predictor* learned through a repeated interaction with the system. On top of this, we augment the algorithm's action-selection phase with a risk-constrained mechanism based on evaluation of a linear program over the constructed tree. The algorithm starts with a risk threshold $\Delta_0 = \Delta$, which is updated in each decision step to take into account the risk already taken by the agent. We denote by Δ_i the threshold in step i.

The main novel features of RAlph (in comparison with the AlphaZero framework) are the following:

Risk predictor. In our algorithm, the predictor is extended with risk prediction.

Risk-constrained action selection. When selecting an action a_i to be played at step i, we solve a linear program (LP) over $\mathcal{T}$, yielding a local policy that maximizes the estimated payoff while keeping the estimated risk below the current threshold Δ_t. The distribution ξ_i used by the local policy in the first step is then used to sample a_i. The LP is such that if the predictor was replaced with a perfect oracle, the LP solution would give the optimal risk-constrained policy.

Risk-constrained exploration. Some variants of AlphaZero enable additional exploration by selecting each action with a probability proportional to its exponentiated visit count (Silver et al. 2017). Our algorithm uses a technique which perturbs the distribution computed by the LP while keeping the risk estimate of the perturbed distribution below the required threshold

Estimation of alternative risk. The risk threshold must be updated after playing an action, since each possible outcome of the action has a potential contribution towards the global risk. We use linear programming and the risk predictor to obtain an estimate of these contributions.

3 Implementation & Evaluation

Predictor. In principle any predictor (e.g. a neural net) can be used with RAlph. In our implementation, as a proof of concept, we use just a simple table predictor, directly storing the estimated payoff and risk for each state s.

Benchmarks. We implemented RAlph and evaluated it on two types of benchmarks. The first is a perfectly observable version of Hallway (Pineau et al. 2003; Smith and Simmons 2004) where we control a robot in a grid maze. In each move, there is a chance of the robot being shifted sideways of the intended move direction, possibly into a trap where there is a chance of destruction. As a second type, we consider a controllable 1-dimensional *random walk,* modeling an investor in a financial market.

```
1 1 1 1 1 1
1 A B x   1
1 D C E g 1
1 1 1 1 1 1
```

Figure 1: Example of a Hallway MDP. Symbols '1', 'x', 'g' represent wall/trap/goal cell respectively. The agent starts in B facing east, and obtains a reward for reaching the goal cell.

Evaluation. We evaluate RAlph on four instances of the Hallway benchmark. The corresponding MDPs have state-spaces of sizes equal to 20, 44, 1136, and 6553600, respectively. For the random walk, we consider benchmarks with 50 and 200 states. We compared RAlph with the RAMCP algorithm from (Chatterjee et al. 2018). RAMCP can also perform risk-averse planning via the use of heuristic search and linear programming, but does not use any predictor. The outcome of our experiments is reported in (Brázdil et al. 2020). RAlph was shown to be much faster than RAMCP, making up to two orders of magnitude less node expansions when building $\mathcal{T}$. RAlph also consistently finds solutions that satisfy the risk threshold, which is not always the case for RAMCP, whose expected payoff and risk tended to deteriorate quite fast with increasing number of states. In contrast, RAlph was able to learn a well-behaving risk-averse policy in less than 15 minutes even in the largest benchmark.

Discussion. RAlph exhibited interesting behavior on several benchmarks. An an example, consider the Hallway instance shown in Figure 1. For $\Delta = 0$, the only way to reach the gold is by exploiting the move perturbations: since the robot cannot to move east from C without risking a shift to the trap, it must keep circling through A, B, C, D until it is randomly shifted to E. RAlph is able, with some parameter tuning, to find this policy.

4 Conclusions & Future Work

Our experiments show that even with a simple predictor, RAlph performs and scales significantly better than a state-of-the-art algorithm. As an interesting future work we see extension of the method to POMDPs and incorporation of more sophisticated predictors.

Acknowledgements

Krishnendu Chatterjee is supported by the Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE), and COST Action GAMENET. Tomáš Brázdil is supported by the Grant Agency of Masaryk University grant no. MUNI/G/0739/2017 and by the Czech Science Foundation grant No. 18-11193S. Petr Novotný and Jiří Vahala are supported by the Czech Science Foundation grant No. GJ19-15134Y.

References

Altman, E. 1999. *Constrained Markov decision processes*, volume 7. CRC Press.

Brázdil, T.; Chatterjee, K.; Novotný, P.; and Vahala, J. 2020. Reinforcement learning of risk-constrained policies in Markov decision processes. *CoRR* abs/2002.12086.

Chatterjee, K.; Elgyütt, A.; Novotný, P.; and Rouillé, O. 2018. Expectation optimization with probabilistic guarantees in pomdps with discounted-sum objectives. In *IJCAI 2018*, 4692–4699.

Pineau, J.; Gordon, G.; Thrun, S.; et al. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*, volume 3, 1025–1032.

Puterman, M. L. 1994. *Markov Decision Processes*. J. Wiley and Sons.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Smith, T., and Simmons, R. 2004. Heuristic search value iteration for POMDPs. In *UAI*, 520–527. AUAI Press.

Undurti, A., and How, J. P. 2010. An online algorithm for constrained POMDPs. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, 3966–3973. IEEE.

Time-based Dynamic Controllability of Disjunctive Temporal Networks with Uncertainty: A Tree Search Approach with Graph Neural Network Guidance

Kevin Osanlou[1,2,3,4], Jeremy Frank[1], J. Benton[1], Andrei Bursuc[5], Christophe Guettier[2], Eric Jacopin[6] and Tristan Cazenave[3]

[1] NASA Ames Research Center [2] Safran Electronics & Defense [3]LAMSADE, Paris-Dauphine
[4] Universities Space Research Association [5]valeo.ai [6]CREC Saint-Cyr Coetquidan
{kevin.osanlou, jeremy.d.frank, j.benton}@nasa.gov
{kevin.osanlou, christophe.guettier}@safrangroup.com andrei.bursuc@valeo.com
eric.jacopin@st-cyr.terre-net.defense.gouv.fr tristan.cazenave@lamsade.dauphine.fr

Abstract

Scheduling in the presence of uncertainty is an area of interest in artificial intelligence due to the large number of applications. We study the problem of dynamic controllability (DC) of disjunctive temporal networks with uncertainty (DTNU), which seeks a strategy to satisfy all constraints in response to uncontrollable action durations. We introduce a more restricted, stronger form of controllability than DC for DTNUs, time-based dynamic controllability (TDC), and present a tree search approach to determine whether or not a DTNU is TDC. Moreover, we leverage the learning capability of a message passing neural network (MPNN) as a heuristic for tree search guidance. Finally, we conduct experiments for which the tree search shows superior results to state-of-the-art timed-game automata (TGA) based approaches, effectively solving fifty percent more DTNU problems on a known benchmark. We also observe that MPNN tree search guidance leads to substantial performance gains on benchmarks of more complex DTNUs, with up to eleven times more problems solved than the baseline with the same time budget.

1 Introduction

Temporal Networks (TN) are a common formalism to represent temporal constraints over a set of time points (*e.g.* start/end of activities in a scheduling problem). The Simple Temporal Networks with Uncertainty (STNUs) (Tsamardinos 2002) (Vidal and Fargier 1999) explicitly incorporate qualitative uncertainty into temporal networks. Considerable work has resulted in algorithms to determine whether or not all timepoints can be scheduled, either up-front or reactively, in order to account for uncertainty (*e.g.* (Morris and Muscettola 2005), (Morris 2014)). In particular, an STNU is *dynamically controllable* (DC) if there is a reactive strategy in which controllable timepoints can be executed either at a specific time, or after observing the occurrence of an uncontrollable timepoint. Cimatti et al. (Cimatti, Micheli, and Roveri 2016) investigate the problem of DC for Disjunctive Temporal Networks with Uncertainty (DTNUs), which generalize STNUs. Figure 1a shows two DTNUs γ and γ' on the left side; a_i are controllable timepoints, u_j are uncontrollable timepoints. Timepoints are variables which can take on any value in $\mathbb{R}$. Constraints between timepoints characterize a minimum and maximum time distance separating them, likewise valued in $\mathbb{R}$. The key difference between STNUs and DTNUs lies in the *disjunctions* that yield more choice points for consistent scheduling, especially reactively.

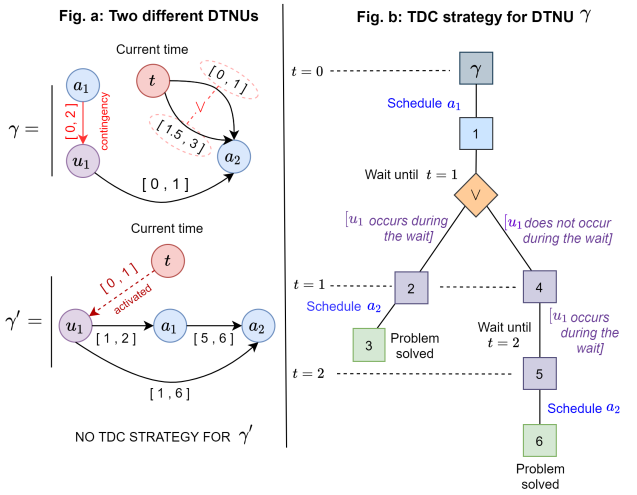


Figure 1: **Two example DTNUs γ and γ'.** In both examples, timepoints a_1 and a_2 are controllable; u_1 is uncontrollable. Black arrows and their intervals represent time constraints between timepoints; the light red arrow and its interval contingency links. The dashed dark red arrow in γ' implies u_1 has already been activated and will occur in the specified interval. A TDC strategy is displayed for γ. Nodes below γ are sub-DTNUs except the $\vee$ node which lists transitional possibilities. DTNU γ', on the other hand, is an example of a DTNU which is DC but not TDC.

The complexity of DC checking for DTNUs is $PSPACE$-complete (Bhargava and Williams 2019), making this a highly challenging problem. The difficulty in proving or disproving DC arises from the need to check all possible combinations of disjuncts in order to handle all possible occurrence outcomes of the uncontrollable timepoints. The best previous approaches for this problem use timed-game automata (TGAs) and Satisfiability Modulo Theories (SMTs), described in (Cimatti, Micheli, and Roveri 2016).

A new emerging trend of neural networks, graph-based neural networks (GNNs), have been proposed as an exten-

sion of convolutional neural networks (CNNs) (Krizhevsky, Sutskever, and Hinton 2012) to graph-structured data. Recent variants based on spectral graph theory include (Defferrard, Bresson, and Vandergheynst 2016), (Li et al. 2016), (Kipf and Welling 2017). They take advantage of relational properties between nodes for classification, but do not take into account potential edge weights. In newer approaches, Message Passing Neural Networks (MPNNs) with architectures such as in (Battaglia et al. 2016), (Gilmer et al. 2017) and (Kipf et al. 2018) use embeddings comprising edge weights within each computational layer. We focus our interest on these architecture types as DTNUs can be formalized as graphs with edge distances representing time constraints.

In this work, we study DC checking of DTNUs as a search problem, express states as graphs, and use MPNNs to learn heuristics based on previously solved DTNUs to guide search. The key contributions of our approach are the following. **(1)** We introduce a time-based form of dynamic controllability (TDC) and a tree search approach to identify TDC strategies. We informally show that TDC implies DC, but the opposite is not generally true. **(2)** We describe an MPNN architecture for handling DTNU scheduling problems and use it as heuristic for guidance in the tree search. Moreover, we define a self-supervised learning scheme to train the MPNN to solve randomly generated DTNUs with short timeouts to limit search duration. **(3)** We introduce constraint propagation rules which enable us to enforce time domain restrictions for variables in order to ensure soundness of strategies found. We carry out experiments showing that the tree search algorithm improves performance and scalability over the best previous DC-solving approach in (Cimatti, Micheli, and Roveri 2016), PYDC-SMT, with 50% more DTNU instances solved. Moreover, we expose that the learned MPNN heuristic considerably improves the tree search on harder DTNUs: performance gains go up to 11 times more instances solved within the same time frame. Results also highlight that the MPNN, which is trained on a set of solved DTNUs, is able to generalize to larger DTNUs.

2 Time-based Dynamic Controllability

A DC *strategy* for a DTNU either executes controllable timepoints at a specific time, or reacts to the occurrence of an uncontrollable timepoint. We present our TDC formalism here. A TDC strategy executes controllable timepoints at specific times under the assumption that some uncontrollable timepoints may occur or not in a given time interval. Each interval in a TDC strategy can have an arbitrary duration. Controllable timepoints are usually executed at the start or the end of an interval, while uncontrollable timepoints may occur inside the interval. TDC also makes it possible to execute a controllable timepoint at the exact same time as the occurrence time of an uncontrollable timepoint inside the interval, with a *reactive execution*.

TDC is less flexible than a DC strategy which can wait for an uncontrollable timepoint to occur before making a new decision. Conversely TDC does not allow, for instance, a delayed reactive execution of a controllable timepoint in response to an uncontrollable one. TDC is a subset of DC, and a stronger form of controllability: TDC implies DC.

DTNU γ' in Figure 1a shows an example of an STNU which is not TDC but DC. In this example, uncontrollable timepoint u_1 is activated, *i.e.* the controllable timepoint associated to u_1 in the contingency links has been executed. Moreover, it is known that u_1 occurs between t and $t+1$, where t is the current time. The interval $[t, t+1]$ is referred to as the *activation time interval* for u_1. Controllable timepoint a_1 must be executed at least 1 time unit after u_1, and controllable timepoint a_2 at least 5 time units after a_1. However, controllable timepoint a_2 cannot be executed later than 6 time units after u_1. A valid DC strategy waits for u_1 to occur, then schedules a_1 exactly 1 time unit later, and a_2 5 time units after a_1. However, for any TDC strategy, there is no wait duration small enough while waiting for u_1 to happen that does not violate these constraints. There will always be some strictly positive lapse of time between the moment u_1 occurs and the end of the wait. The exact execution time of u_1 during the wait is unknown: a TDC strategy therefore assumes u_1 happened at the end of the wait when trying to schedule a_1 at the earliest. Therefore, the earliest time a_1 can be scheduled in a TDC strategy is 1 time unit after the end of the wait, which is too late.

3 Tree Search Preliminaries

We introduce here the tree search algorithm. The approach discretizes uncontrollable durations, *i.e.* durations when one or several uncontrollable timepoints can occur, into reduced intervals. These are in turn used to account for possible outcomes of uncontrollable timepoints and adapt the scheduling strategy accordingly. The root of the search tree built by the algorithm is a DTNU, and other tree nodes are either sub-DTNUs or logical nodes (*OR, AND*) which respectively represent decisions that can be made and how uncontrollable timepoints can unfold. At a given DTNU tree node, decisions such as executing a controllable timepoint or waiting for a period of time develop children DTNU nodes for which these decisions are propagated to constraints. The TDC controllability of a *leaf* DTNU, *i.e.* a sub-DTNU for which all controllable timepoints have been executed and uncontrollable timepoints are assumed to have occurred in specific intervals, indicates whether or not this sub-DTNU has been solved at the end of the scheduling process. We also refer to the TDC controllability of a DTNU node in the search tree as its *truth attribute*. Lastly, the search logically combines TDC controllability of children DTNUs to determine TDC controllability for parent nodes. We give a simple example of a TDC strategy for a DTNU γ in Figure 1.

Let $\Gamma = \{A, U, C, L\}$ be a DTNU. A is the list of controllable timepoints, U the list of uncontrollable timepoints, C the list of constraints and L the list of contingency links. The root node of the search tree is Γ. There are four different types of nodes in the tree and each node has a *truth* attribute (see §4.4) which is initialized to *unknown* and can be set to either *true* or *false*. The different types of tree nodes are listed below and shown in Figure 2.

***DTNU* nodes.** Any DTNU node other than the original problem Γ corresponds to a sub-problem of Γ at a given

point in time t, for which some controllable timepoints may have already been scheduled in upper branches of the tree, some amount of time may have passed, and some uncontrollable timepoints are assumed to have occurred. A DTNU node is made of the same timepoints A and U, constraints C and contingency links L as DTNU Γ. It also carries a schedule memory S of what exact time, or during what time interval, scheduled timepoints were executed during previous decisions in the tree. Lastly, the node also keeps track of the activation time intervals of activated uncontrollable timepoints B. The schedule memory S is used to create an updated list of constraints C' resulting from the propagation of the execution time or execution time interval of timepoints in constraints C as described in §4.5. A non-terminal DTNU node, *i.e.* a DTNU node for which all timepoints have not been scheduled, has exactly one child node: a *d-OR* node.

***OR* nodes.** When a choice can be made at time t, this transition control is represented by an *OR* node. We distinguish two types of such nodes, *d-OR* and *w-OR* . For *d-OR* nodes, the first type of choice available is which controllable timepoint a_i to execute. This leads to a DTNU node. The other type of choice is to wait a period of time (§4.1) which leads to a *WAIT* node. *w-OR* nodes can be used for *reactive wait strategies*, *i.e.* to stipulate that some controllable timepoints will be scheduled reactively during waits (§4.3). The parent of a *w-OR* node is therefore a *WAIT* node and its children are *AND* nodes, described below.

***WAIT* nodes.** These nodes are used after a decision to wait a certain period of time Δ_t. The parent of a *WAIT* node is a *d-OR* node. A *WAIT* node has exactly one child: a *w-OR* node, which has the purpose of exploring different reactive wait strategies. The uncertainty management related to uncontrollable timepoints is handled by *AND* nodes.

***AND* nodes.** Such nodes are used after a wait decision is taken and a reactive wait strategy is decided, represented respectively by a *WAIT* and *w-OR* node. Each child node of the *AND* node is a DTNU node at time $t + \Delta_t$, t being the time before the wait and Δ_t the wait duration. Each child node represents an outcome of how uncontrollable timepoints may unfold and is built from the set of *activated* uncontrollable timepoints (uncontrollable timepoints that have been triggered by the execution of their controllable timepoint) whose occurrence time interval overlaps the wait. If there are l activated uncontrollable timepoints, then there are at most 2^l *AND* node children, representing each element of the power set of activated uncontrollable timepoints (§4.1).

Figure 2 illustrates how a sub-problem of Γ, referred to as $DTNU_{O,P,t}$, is developed. Here, $O \subset A$ is the set of controllable timepoints that have already been executed, $P \subset U$ the set of uncontrollable timepoints which have occurred, and t the time. This root node transitions into a *d-OR* node. The *d-OR* node in turn is developed into several children nodes $DTNU_{O\cup\{a_i\},P,t}$ and a *WAIT* node. Each node $DTNU_{O\cup\{a_i\},P,t}$ corresponds to a sub-problem which is obtained from the execution of controllable timepoint a_i at time t. The *WAIT* node refers to the process of waiting a given period of time, Δ_t in the figure, before making the next decision. The *WAIT* node leads directly to a

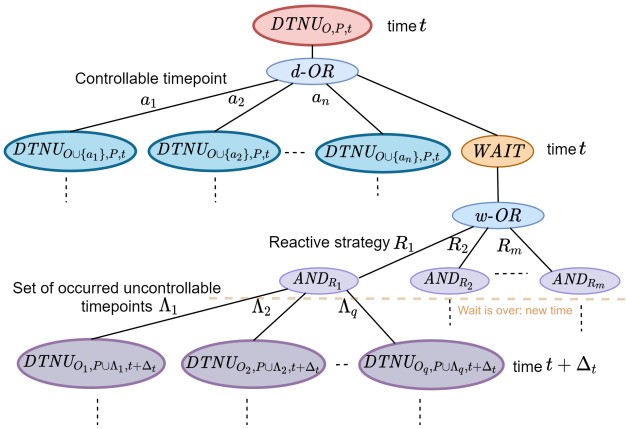


Figure 2: **Basic structure of the search tree describing how a DTNU node $DTNU_{O,P,t}$ is developed.** $DTNU_{O,P,t}$ (placed at the root of the tree) refers to a DTNU where O is the set of controllable timepoints that have already been executed, P the set of uncontrollable timepoints that have occurred, and t the time. Each branch a_i refers to a controllable timepoint a_i, R_i to a reactive strategy during the wait, and Λ_i to a combination of uncontrollable timepoints which can occur during the wait.

w-OR node which lists different wait strategies R_i. If there are l activated uncontrollable timepoints, there are 2^l subsets of uncontrollable timepoints Λ_i that could occur. Each AND_{R_j} node has one sub-problem DTNU for each Λ_i. Each sub-problem $DTNU_{O_i,P\cup\Lambda_i,t+\Delta_t}$ of the node AND_{R_j} is a DTNU at time $t+\Delta_t$ for which all uncontrollable timepoints in Λ_i are assumed to have happened during the wait period, *i.e.* in the time interval $[t, t + \Delta_t]$. Additionally, some controllable timepoints may have been reactively executed during the wait and may now be included in the set of scheduled controllable timepoints O_i. Otherwise, $O_i = O$.

Two types of leaf nodes exist in the tree. The first type is a node $DTNU_{A,U,t}$ for which all controllable timepoints $a_i \in A$ have been scheduled and all uncontrollable timepoints $u_i \in U$ have occurred. The second type is a node $DTNU_{A\setminus A',U,t}$ for which all uncontrollable timepoints $u_i \in U$ have occurred, but some controllable timepoints $a_i \in A'$ have not been executed. The constraint satisfiability test of the former type of leaf node is straightforward: all execution times of all timepoints are propagated to constraints in the same fashion as in §4.5. The leaf node's truth attribute is set to *true* if all constraints are satisfied, *false* otherwise. For the latter type, we propagate the execution times of all uncontrollable timepoints as well as all scheduled controllable timepoints in the same way, and obtain an updated set of constraint C'. This leaf node, $DTNU_{A\setminus A',U,t}$, is therefore characterized as $\{A', \emptyset, C', \emptyset\}$ and is a DTN. We add the constraints $a'_i \geq t, \forall a'_i \in A'$ and use a mixed integer linear programming solver (Cplex 2009) to solve the DTN. If a solution is found, the execution time values for each $a'_i \in A'$ are stored and the leaf node's truth value is set to *true.* Otherwise, it is set to *false.* After a truth value is assigned to the leaf node, the truth propagation function defined in §4.4 is called to logically infer truth value properties for parent nodes. Lastly, the search algorithm ex-

plores the tree in a depth-first manner. At each *d-OR* , *w-OR* and *AND* node, children nodes are visited in the order they are created. Once a child node is selected, its entire subtree will be processed by the algorithm before the other children are explored. Some simplifications made in the exploration are detailed in §11.6 in the appendix.

4 Tree Search Characteristics

4.1 Wait action

When a wait decision of duration Δ_t is taken at time t for a DTNU node, two categories of uncontrollable timepoints are considered to account for all transitional possibilities:

- $Z = \{\zeta_1, \zeta_2, ..., \zeta_l\}$ is a set of timepoints that could either happen during the wait, or afterwards, *i.e.* the end of the activation time interval for each ζ_i is greater than $t + \Delta_t$.
- $H = \{\eta_1, \eta_2, ..., \eta_m\}$ is a set of timepoints that are certain to happen during the wait, *i.e.* the end of the activation time interval for each η_i is less than or equal to $t + \Delta_t$.

There are $q = 2^l$ number of different possible combinations (empty set included) $\Upsilon_1, \Upsilon_2, ..., \Upsilon_q$ of elements taken from Z. For each combination Υ_i, the set $\Lambda_i = H \cup \Upsilon_i$ is created. The union $\bigcup_{i=1}^{q} \Lambda_i$ refers to all possible combinations of uncontrollable timepoints which can occur by $t + \Delta_t$. In Figure 2, for each *AND* node, the combination Λ_i leads to a DTNU sub-problem $DTNU_{O_i, P \cup \Lambda_i, t+\Delta_t}$ for which the uncontrollable timepoints in Λ_i are considered to have occurred between t and $t + \Delta_t$ in the schedule memory S. In addition, any potential controllable timepoint ϕ planned to be instantly scheduled in a reactive wait strategy R_i in response to an uncontrollable timepoint u in Λ_i will also be considered to have been scheduled between t and $t + \Delta_t$ in S. The only exception is when checking constraint satisfiability for the conjunct $u - \phi \in [0, y]$ which required the reactive scheduling, for which we assume ϕ executed at the same time as u, thus the conjunct is considered satisfied.

4.2 Wait Eligibility and Period

The way time is discretized holds direct implications on the search space explored and the capability of the algorithm to find TDC strategies. Longer waits make the search space smaller, but carry the risk of missing key moments where a decision is needed. On the other hand, smaller waits can make the search space too large to explore. We explain when the wait action is eligible, and how its duration is computed.

Eligibility At least one of these two criteria has to be met for a *WAIT* node to be added as child of a *d-OR* node. **(1)** There is at least one activated uncontrollable timepoint for the parent DTNU node. **(2)** There is at least one conjunct of the form $v \in [x, y]$, where v is a timepoint, in the constraints of the parent DTNU node. These criteria ensure that the search tree will not develop branches below *WAIT* nodes when waiting is not relevant, *i.e.* when a controllable timepoint necessarily needs to be scheduled. It also prevents the tree search from getting stuck in infinite *WAIT* loop cycles.

Wait Period We define the wait duration Δ_t at a given *d-OR* node eligible for a wait dynamically by examining the updated constraint list C' of the parent DTNU and the activation time intervals B of its activated uncontrollable timepoints. Let t be the current time for this DTNU node. The wait duration is defined by comparing t to elements in C' and B to look for a minimum positive value defined by the following three rules. **(1)** For each activated time interval $u \in [x, y]$ in B, we select $x - t$ or $y - t$, whichever is smaller and positive, and we keep the smallest value δ_1 found over all activated time intervals. **(2)** For each conjunct $v \in [x, y]$ in C', where v is a timepoint, we select $x - t$ or $y - t$, whichever is smaller and positive, and we keep the smallest value δ_2 found over all conjuncts. **(3)** We determine timepoints which need to be scheduled ahead of time by chaining constraints together. Intuitively, when a conjunct $v \in [x, y]$ is in C', it means v has to be executed when $t \in [x, y]$ to satisfy this conjunct. However, v could be linked to other timepoints by constraints which require them to happen before v. These timepoints could in turn be linked to yet other timepoints in the same way, and so on. The third rule consists in chaining backwards to identify potential timepoints which start this chain and potential time intervals in which they need to be executed. The following mechanism is used: for each conjunct $v \in [x, y]$ in C' found in (2), we apply a recursive backward chain function to both (v, x) and (v, y). We detail here how it is applied to (v, x), the process being the same for (v, y). Conjuncts of the form $v - v' \in [x', y'], x' \geq 0$ in C' are searched for. For each conjunct found, we add to a list two elements, $(v', x - x')$ and $(v', x - y')$. We select $x - x' - t$ or $x - y' - t$, whichever is smaller and positive, as potential minimum candidate. The backward chain function is called recursively on each element of the list, proceeding the same way. We keep the smallest candidate δ_3. Figure 8 in the appendix illustrates an application of this process. Finally, we set $\Delta_t = \min(\delta_1, \delta_2, \delta_3)$ as the wait duration. This duration is stored inside the *WAIT* node.

4.3 Reactive scheduling during waits

Execution of a controllable timepoint may be necessary in some situations at the exact same time as when an uncontrollable timepoint occurs to satisfy a constraint. Therefore, different reactive wait strategies are considered and listed as children of a *w-OR* node after a wait decision, before the actual start of the wait itself. We designate as a *conjunct* a constraint relationship of the form $v_i - v_j \in [x, y]$ or $v_i \in [x, y]$, where v_i, v_j are timepoints and $x, y, \in \mathbb{R}$. We refer to a constraint where several conjuncts are linked by $\vee$ operators as a *disjunct*. If at any given DTNU node in the tree there is an activated uncontrollable timepoint u with the potential to occur during the next wait and there is at least one unscheduled controllable timepoint a such that a conjunct of the form $u - a \in [0, y], y \geq 0$ is present in the constraints, a reactive wait strategy is available that will schedule a as soon as u occurs.

If there are s controllable timepoints that may be reactively scheduled, there are 2^s different reactive wait strategies R_i, each of which is embedded in an *AND* child of the

w-OR node. Let $\Phi = \{\phi_1, \phi_2, ..., \phi_s\} \subset A$ be the complete set of unscheduled controllable timepoints for which there are conjunct clauses $u - \phi_i \in [0, y]$. We denote as $R_1, R_2, ..., R_m$ all possible combinations of elements taken from Φ, including the empty set. The child node AND_{R_i} of the *w-OR* node resulting from the combination R_i has a reactive wait strategy for which all controllable timepoints in R_i will be immediately executed at the moment u occurs during the wait, if it does. If u doesn't occur, no controllable timepoint is reactively scheduled during the wait.

4.4 Truth Value Propagation

In this section, we describe how truth attributes of nodes are related to each other. The truth attribute of a tree node represents its TDC controllability, and the relationships shared between nodes make it possible to define sound strategies. When a leaf node is assigned a truth attribute β, the tree search is momentarily stopped and β is propagated onto upper parent nodes. To this end, a parent node $\boldsymbol{\omega}$ is selected recursively and we distinguish the following cases:

- The parent $\boldsymbol{\omega}$ is a DTNU or *WAIT* node: $\boldsymbol{\omega}$ is assigned β.
- The parent $\boldsymbol{\omega}$ is a *d-OR* or *w-OR* node: If $\beta = true$, then $\boldsymbol{\omega}$ is assigned *true* . If $\beta = false$ and all children nodes of $\boldsymbol{\omega}$ have *false* attributes, $\boldsymbol{\omega}$ is assigned *false* . Otherwise, the propagation stops.
- The parent $\boldsymbol{\omega}$ is an *AND* node: If $\beta = false$, then $\boldsymbol{\omega}$ is assigned *false* . If $\beta = true$ and all children nodes of $\boldsymbol{\omega}$ have *true* attributes, $\boldsymbol{\omega}$ is assigned *true* . Otherwise, the propagation stops.

After the propagation finishes, the tree search algorithm resumes where it was stopped. A *true* attribute reaching the root node of the tree means a TDC strategy has been found. A *false* attribute means none could be found. The pseudocode for the propagation algorithm is given in Algorithm 1 in the appendix.

4.5 Constraint Propagation

Decisions taken in the tree define when controllable timepoints are executed and also bear consequences on the execution time of uncontrollable timepoints. We explain here how these decisions are propagated into constraints, as well as the concept of '*tight bound*'. Let C' be the list of updated constraints for a DTNU node $\boldsymbol{\psi}$ for which the parent node is $\boldsymbol{\omega}$. We distinguish two cases. Either $\boldsymbol{\omega}$ is a *d-OR* node and $\boldsymbol{\psi}$ results from the execution of a controllable timepoint a_i, or $\boldsymbol{\omega}$ is an *AND* node and $\boldsymbol{\psi}$ results from a wait of Δ_t time units. In the first case, let t be the execution time of a_i. The updated list C' is built from the constraints of the parent DTNU of $\boldsymbol{\psi}$ in the tree. If a conjunct contains a_i and is of the form $a_i \in [x, y]$, this conjunct is replaced with *true* if $t \in [x, y]$, *false* otherwise. If the conjunct is of the form $v_j - a_i \in [x, y]$, we replace the conjunct with $v_j \in [t + x, t + y]$. The other possibility is that $\boldsymbol{\psi}$ results from a wait of Δ_t time at time t, with a reactive wait strategy R_j. In this case, the new time is $t + \Delta_t$ for $\boldsymbol{\psi}$. As a result of the wait, some uncontrollable timepoints $u_i \in \Lambda_i$ are assumed to have occured, and some controllable timepoints $a_i \in R_j$ may be executed reactively during the wait. Let $v_i \in \Lambda_i \cup R_j$ be these timepoints occurring during the wait. The execution time of these timepoints is assumed to be in $[t, t + \Delta_t]$. For uncontrollable timepoints $u'_i \in \Lambda'_i \subset \Lambda_i$ for which the activation time ends at $t + \Delta'_{t_i} < t + \Delta_t$, and potential controllable timepoints a'_i instantly reacting to these uncontrollable timepoints, the execution time is further reduced and considered to be in $[t, t + \Delta'_{t_i}]$. We define a concept of *tight bound* to update constraints which restricts time intervals in order to account for all possible values v_i can take between t and $t + \Delta_t$. For all conjuncts $v_j - v_i \in [x, y]$, we replace the conjunct with $v_j \in [t + \Delta_t + x, t + y]$. Intuitively, this means that since v_i can happen at the latest at $t + \Delta_t$, v_j can not be allowed to happen before $t + \Delta_t + x$. Likewise, since v_i can happen at the earliest at t, v_j can not be allowed to happen after $t + y$. Finally, if $t + \Delta_t + x > t + y$, the conjunct is replaced with *false* . Also, the process can be applied recursively in the event that v_j is also a timepoint that occurred during the wait, in which case the conjunct would be replaced by *true* or *false*. In any case, any conjunct obtained of the form $a_j \in [x', y']$ is replaced with *false* if $t + \Delta_t > y'$. Finally, if all conjuncts inside a disjunct are set to *false* by this process, the constraint is violated and the DTNU is no longer satisfiable.

5 Learning-based Heuristic

We present here our learning model and explain how it provides tree search heuristic guidance. Our learning architecture originates from (Gilmer et al. 2017). It uses message passing rules allowing neural networks to process graph-structured inputs where both vertices and edges possess features. This architecture was originally designed for node classification in quantum chemistry and achieved state-of-the-art results on a molecular property prediction benchmark. Here, we first define a way of converting DTNUs into graph data. Then, we process the graph data with our MPNN and use the output to guide the tree search.

Let $\Gamma = \{A, U, C, L\}$ be a DTNU. We explain how we turn Γ into a graph $\mathcal{G} = (\mathcal{K}, \mathcal{E})$. First, we convert all time values from absolute to relative with the assumption the current time for Γ is $t = 0$. We search all converted time intervals $[x_i, y_i]$ in C and L for the highest interval bound value d_{max}, *i.e.* the farthest point in time. We proceed to normalize every time value in C and L by dividing them by d_{max}. As a result, every time value becomes a real number between 0 and 1. Next, we convert each controllable timepoint $a \in A$ and uncontrollable timepoint $u \in U$ into graph nodes with corresponding *controllable* or *uncontrollable* node features. The time constraints in C and contingency links in L are expressed as edges between nodes with 10 different edge distance classes ($0 : [0, 0.1)$, $1 : [0.1, 0.2)$, ..., $9 : [0.9, 1]$). We also use additional edge features to account for edge types (constraint, disjunction, contingency link, direction sign for lower and upper bounds). Moreover, intermediary nodes are used with a distinct node feature in order to map possible disjunctions in constraints and contingency links. We add a *WAIT* node with a distinct node feature which implicitly designates the act of waiting a period of time. The graph conversion of DTNU γ is characterized by three elements:

the matrix of all node features X_κ, the adjacency matrix of the graph X_ϵ and the matrix of all edge features X_ρ.

These features are processed by several consecutive *message passing* layers from (Gilmer et al. 2017). Each layer takes an input graph, consists of a phase during which messages are passed between nodes, and returns the same graph with new node features. The overall process for a layer is the following. For each node κ_i in the input graph, a *feature aggregation* phase is applied and creates new features for κ_i from current features of neighboring nodes and edges. In detail, for each neighbor node κ_j, a small neural network (termed multi-layer perceptron, or MLP) takes as input the features of the edge connecting κ_i and κ_j and returns a matrix which is then multiplied by the features of κ_j to obtain a feature vector. The sum of these vectors for the entire neighborhood defines the new features for κ_i. The output of the message passing layer consists of the graph updated with the new node features. The feature aggregation process being the same for any node, it can be applied to input graphs of any size, *i.e.* it enables our MPNN architecture to take as input DTNUs of any size. Moreover, each message passing layer contains a different MLP and can thus be trained to learn a different feature aggregation scheme.

Let f be the mathematical function for our MPNN and θ its set of parameters. Our function f stacks 5 message passing layers coupled with the $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$ piecewise activation function (Glorot, Bordes, and Bengio 2011). The `sigmoid` function $\sigma(\cdot) = \frac{1}{1+\exp(-\cdot)}$ is then used to obtain a list of probabilities π over all nodes in $\mathcal{G}$: $f_\theta(X_\kappa, X_\epsilon, X_\rho) = \pi$. The probability of each node κ in π corresponds to the likelihood of transitioning into a TDC DTNU from the original DTNU Γ by taking the action corresponding to κ. If κ represents a controllable timepoint a in Γ, its corresponding probability in π is the likelihood of the sub-DTNU resulting from the execution of a being TDC. If κ represents a *WAIT* decision, its probability refers to the likelihood of the *WAIT* node having a *true* attribute, *i.e.* the likelihood of all children DTNUs resulting from the wait being TDC (with the wait duration rules set in §4.2). We call these two types of nodes *active* nodes. Otherwise, if κ is another type of node, its probability is not relevant to the problem and ignored. Our MPNN is trained on DTNUs generated and solved in §6 only on active nodes by minimizing the cross-entropy loss:

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{q} -Y_{ij}\log(f_\theta(X_i)_j) - (1-Y_{ij})\log(1-f_\theta(X_i)_j)$$

Here $X_i = (X_{i_\kappa}, X_{i_\epsilon}, X_{i_\rho})$ is DTNU number i among a training set of m examples, Y_{ij} is the TDC controllability (1 or 0) of active node number j for DTNU number i.

Lastly, the MPNN heuristic is used in the following way in the tree search. Once a *d-OR* node is reached, the parent DTNU node is converted into a graph and the MPNN f is called upon the corresponding graph elements $X_\kappa, X_\epsilon, X_\rho$. Active nodes in output probabilities π are then ordered by highest values first, and the tree search visits the corresponding children tree nodes in the suggested order, preferring children with higher likelihood of being TDC first.

6 Randomized Simulations for Heuristic Training

We leverage a learning-based heuristic to guide the tree search. A key component in learning-based methods is the annotated training data. We generate such data in automatic manner by using a DTNU generator to create random DTNU problems and solving them with a modified version of the tree search. We store results and use them for training the MPNN. We detail now our data generation strategy.

We create DTNUs with a number of controllable timepoints ranging from 10 to 20 and uncontrollable timepoints ranging from 1 to 3. The generation process is the following. For interval bounds of constraint conjuncts or contingency links, we randomly generate real numbers within $[0, 100]$. We restrict the number of conjuncts inside a disjunct to 5 at most. A random number $n_1 \in [10, 20]$ of controllable timepoints and $n_2 \in [1, 3]$ of uncontrollable timepoints are selected. Each uncontrollable timepoint is randomly linked to a different controllable timepoint with a contingency link. Next, we iterate over the list of timepoints, and for each timepoint v_i not appearing in constraints or contingency links, we add in the constraints a disjunct for which at least one conjunct constrains v_i. The type of conjunct is selected randomly from either a *distance* conjunct $v_i - v_j \in [x, y]$ or a *bounded* conjunct $v_i \in [x, y]$. On the other hand, if v_i was already present in the constraints or contingency links, we add a disjunct constraining v_i with only a 20% probability.

In order to solve these DTNUs, we modify the tree search as follows. For a DTNU Γ, the first *d-OR* child node is developed as well as its children $\psi_1, \psi_2, ..., \psi_n \in \Psi$. The modified tree search explores each ψ_i multiple times (ν times at most), each time with a timeout of τ seconds. We set $\nu = 25$ and $\tau = 3$. For each exploration of ψ_i, children nodes of any *d-OR* node encountered in the corresponding subtree are explored randomly each time. If ψ_i is proved to be either TDC or non-TDC during an exploration, the next explorations of the same child ψ_i are called off and the truth attribute β_i of ψ_i is updated accordingly. The active node number k, corresponding to the decision leading to ψ_i from DTNU Γ's *d-OR* node, is updated with the same value, *i.e.* $Y_k = \beta_i$ (1 for *true*, 0 for *false*). If every exploration times out, ψ_i is assumed non-TDC and Y_k is set to *false*. Once each ψ_i has been explored, the pair $\langle G(\Gamma), (Y_1, Y_2, ..., Y_n)\rangle$ is stored in the training set, where $G(\Gamma)$ is the graph conversion of Γ described in §5. Data related to solved sub-DTNUs of Γ are not stored in the training set as it was found to cause bias issues and overall decrease generalization in MPNN predictions.

The assumption of non-TDC controllability for children nodes for which all explorations time out is acceptable in the sense that the heuristic used is not admissible and does not need to be. The output of the MPNN is a probability for each child node of the *d-OR* node, creating a preferential order of visit by highest probabilities first. Even in the event the suggested order first recommends visiting children nodes which will be found to be non-TDC, the algorithm will continue to explore the remaining children nodes until one is found to be TDC. Nevertheless, such a scenario will rarely occur as the trained MPNN will give higher probabilities for children

nodes for which explorations would tend to find a TDC strategy before timeout, and lower probabilities for ones where explorations would tend to result in a timeout.

7 Strategy Execution

A strategy found by the tree search for a DTNU Γ is sound and guarantees constraint satisfiability if executed in the following manner. Let $\mathcal{Q}$ be the system interacting with the environment, executing controllable timepoints and observing how uncontrollable timepoints unfold. At each DTNU node in the tree, $\mathcal{Q}$ moves on to the child *d-OR* node. The child node ψ_i of the *d-OR* node which was found by the strategy to have a *true* attribute is selected. If ψ_i is a DTNU node, $\mathcal{Q}$ executes the corresponding controllable timepoint a_i and moves on to ψ_i. If ψ_i is a *WAIT* node, $\mathcal{Q}$ moves on to ψ_i, reads the wait duration Δ_t stored in ψ_i and moves on to the child *w-OR* node. The child node AND_{R_j} of the *w-OR* node which has a *true* attribute is selected, and $\mathcal{Q}$ will wait Δ_t time units with the reactive wait strategy R_j. After the wait is over, $\mathcal{Q}$ observes the list of all uncontrollable timepoints Λ_i which occurred, deduces which DTNU child node of the AND_{R_j} node it transitioned into, and moves on to that node.

By following these guidelines, the final tree node $\mathcal{Q}$ transitions into is necessarily a leaf node with a *true* attribute, *i.e.* a node for which all constraints are satisfied. This is due to the fact that for *d-OR* and *w-OR* nodes $\mathcal{Q}$ visits, $\mathcal{Q}$ chooses to transition into a child node with a *true* attribute. For *AND* nodes $\mathcal{Q}$ visits, all children DTNU nodes have a *true* attribute, so $\mathcal{Q}$ transitions into a child node with a *true* attribute regardless of how uncontrollable timepoints unfold.

8 Experiments

We evaluate experimentally the efficiency of the tree search approach and the effect of the MPNN's guidance. We also compare them with a DC solver from (Cimatti, Micheli, and Roveri 2016). TDC is a subset of DC and a more restrictive form of controllability: non-TDC controllability does not imply non-DC controllability. A TDC solver can thus be expected to offer better performance than a DC one while potentially being unable to find a strategy when a DC algorithm would. In this section, we refer to the tree search algorithm as TS, the tree search algorithm guided by the trained MPNN up to the 15^{th} (respectively X^{th}) *d-OR* node depth-wise in the tree as MPNN-TS (respectively MPNN-TS-X) and the most efficient DC solver from (Cimatti, Micheli, and Roveri 2016) as PYDC-SMT ordered.

First, we use the benchmark in the experiments of (Cimatti, Micheli, and Roveri 2016) from which we remove DTNs and STNs. We compare TS, MPNN-TS and PYDC-SMT on the resulting benchmark which is comprised of 290 DTNUs and 1042 STNUs. Here, Limiting the maximum depth use of the MPNN to 15 offers a good trade off between guidance gain and cost of calling the heuristic. Results are given in Figure 3. We observe that TS solves roughly 50% more problem instances than PYDC-SMT within the allocated time (20 seconds). In addition, TS solves 56% of all instances while the remaining ones time out. Among solved instances, a strategy is found for 89% and the remaining

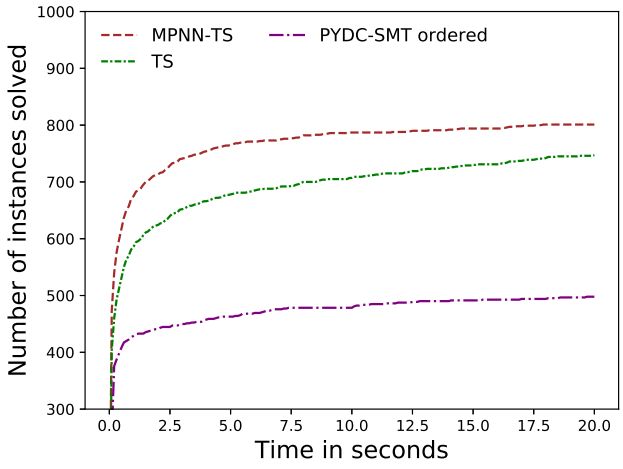


Figure 3: **Experiments on (Cimatti, Micheli, and Roveri 2016)'s benchmark from which DTNs and STNs have been removed.** The X-axis represents the allocated time in seconds and the Y-axis the number of instances in the benchmark each solver can solve within the corresponding allocated time. Timeout is set to 20 seconds per instance.

11% are proved non-TDC. On the other hand, PYDC-SMT solves 37% of all instances. A strategy is found for 85% of PYDC-SMT's solved instances while the remaining 15% are proved non-DC. Finally, out of all instances PYDC-SMT solves, TS solves 97% accurately with the same conclusion, *i.e.* TDC when DC and non-TDC when non-DC. The use of the heuristic leads to an additional $+6\%$ problems solved within the allocated time. We argue this small increase is essentially due to the fact that most problems solved in the benchmark are small-sized problems with few timepoints which are solved quickly. Despite this fact, the heuristic still provides performance boost on a benchmark generated with another DTNU generator, suggesting the bias introduced by our DTNU generator remains limited and the MPNN is able to generalize to DTNUs created with a different approach.

For further evaluation of the heuristic, we create new benchmarks using the DTNU generator from §6 with varying number of timepoints. These benchmarks have fewer quick to solve DTNUs and harder ones instead. Each benchmark contains 500 randomly generated DTNUs which have 1 to 3 uncontrollable timepoints. Moreover, each DTNU has 10 to 20 controllable timepoints in the first benchmark B_1, 20 to 25 in the second benchmark B_2 and 25 to 30 in the last benchmark B_3. Each disjunct in the constraints of any DTNU contains up to 5 conjuncts. Experiments on B_1, B_2 and B_3 are respectively shown in Figure 4, 6c (in the appendix) and 5. We note that for all three benchmarks no solver ever proves non-TDC or non-DC controllability before timing out due to the larger size of these problems.

PYDC-SMT performs poorly on B_1 and cannot solve any instance on B_2 and B_3. TS does not perform well on B_2 and only solves 2 instances on B_3. However, we see a significantly higher gain from the use of the MPNN, varying with the maximum depth use. At best depth use, the gain is

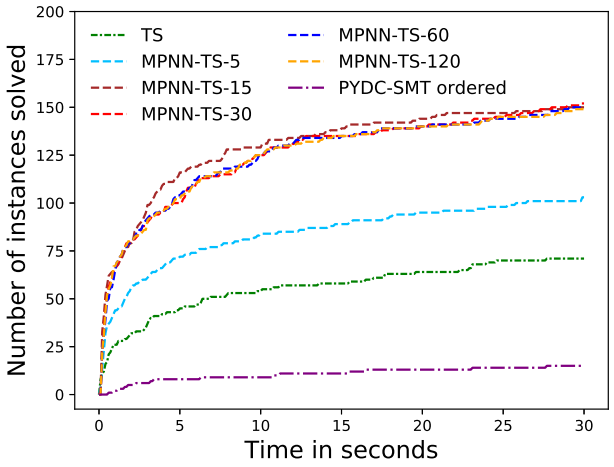


Figure 4: **Experiments on benchmark B_1.** Axes are the same as in Figure 3. Timeout is set to 30 seconds per instance.

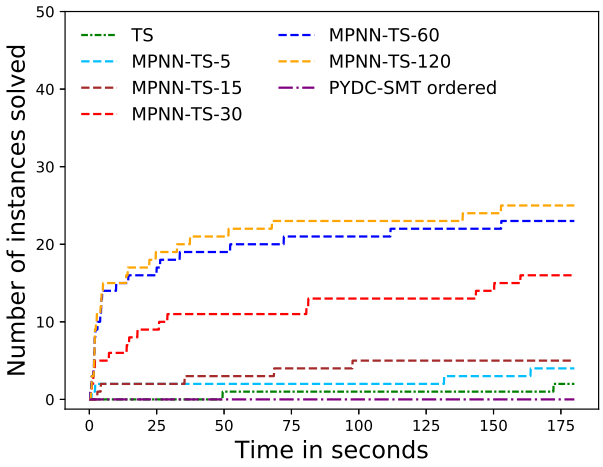


Figure 5: **Experiments on benchmark B_3.** Axes are the same as in Figure 3. Timeout is set to 180 seconds.

$+91\%$ instances solved for B_1, $+980\%$ for B_2 and $+1150\%$ for B_3. The more timepoints instances have, the more worthwhile heuristic guidance appears to be. Indeed, the optimal maximum depth use of the MPNN in the tree increases with the problem size: 15 for B_1, 60 for B_2 and 120 for B_3. We argue this is due to the fact that more timepoints results in a wider search tree overall, including in deeper sections where heuristic use was not necessarily worth its cost for smaller problems. Furthermore, the MPNN is trained on randomly generated DTNUs which have 10 to 20 controllable timepoints. The promising gains shown by experiments on B_2 and B_3 suggest generalization of the MPNN to bigger problems than it is trained on.

The proposed tree search approach presents a good trade off between search completeness and effectiveness: almost all examples solved by PYDC-SMT from (Cimatti, Micheli, and Roveri 2016)'s benchmark are solved with the same conclusion, and many more which could not be solved are. Moreover, the TDC approach scales up better to problems with more timepoints, and the tree structure allows the use of learning-based heuristics. Although these heuristics are not key to solving problems of big scales, our experiments suggest they can still provide a high increase in efficiency.

9 Related Work

Learning-based heuristics have become increasingly popular for planning, combinatorial and network modeling problems. Recent works applied to network modeling and routing problems include (Rusek et al. 2019), (Chen et al. 2018), (Xu et al. 2018), (Kool and Welling 2018). Recently, GNNs have become a popular extension of CNNs. Essentially, their ability to represent problems with a graph structure and the resulting node permutation invariance makes them convenient for some applications. We refer the reader to (Wu et al. 2019) for a complete survey on GNNs. In combinatorial optimization, GNNs can benefit both approximate and exact solvers. In (Li, Chen, and Koltun 2018), authors combine tree search, GNNs and a local search algorithm to achieve state-of-the-art results for approximate solving of NP-hard problems such as the maximum independent set problem. On the other hand, (Gasse et al. 2019) use a GNN for branch and bound variable selection for exact solving of NP-hard problems and achieve superior results to previous learning approaches. In path-planning problems with NP-hard constraints, (Osanlou et al. 2019) use a GNN to predict an upper bound for a branch and bound solver and outperform an A*-based planner coupled with a problem-suited handcrafted heuristic. (Ma et al. 2018) leverage a GNN for the selection of a planner inside a portfolio for STRIPS planning problems and outperform the previous leading learning-based approach based on a CNN (Sievers et al. 2019). In most works, GNNs seem to offer generalization to bigger problems than they are trained on. Results from our experiments are in line with this observation.

10 Conclusion

We introduced a new type of controllability, time-based dynamic controllability (TDC), and a tree search approach for solving disjunctive temporal networks with uncertainty (DTNU) in TDC. Strategies are built by discretizing time and exploring different decisions which can be taken at different key points, as well as anticipating how uncontrollable timepoints can unfold. We defined constraint propagation rules which ensure soundness of strategies found. We showed that the tree search approach is able to solve DTNUs in TDC more efficiently than the state-of-the-art dynamic controllability (DC) solver, PYDC-SMT, with almost always the same conclusion. Lastly, we created MPNN-TS, a solver which combines the tree search with a heuristic function based on message passing neural networks (MPNN) for guidance. The MPNN is trained with a self-supervised strategy and enables steady improvements of the tree search on harder DTNU problems, notably on DTNUs of bigger size than those used for training the MPNN.

References

Battaglia, P.; Pascanu, R.; Lai, M.; Rezende, D. J.; et al. 2016. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, 4502–4510.

Bhargava, N., and Williams, B. C. 2019. Complexity bounds for the controllability of temporal networks with conditions, disjunctions, and uncertainty. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 6353 – 6357.

Chen, X.; Guo, J.; Zhu, Z.; Proietti, R.; Castro, A.; and Yoo, S. 2018. Deep-rmsa: A deep-reinforcement-learning routing, modulation and spectrum assignment agent for elastic optical networks. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*, 1–3. IEEE.

Cimatti, A.; Micheli, A.; and Roveri, M. 2016. Dynamic controllability of disjunctive temporal networks: Validation and synthesis of executable strategies. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Cplex, I. I. 2009. V12. 1: User's manual for cplex. *International Business Machines Corporation* 46(53):157.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, 3844–3852.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12(Jul):2121–2159.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, 15554–15566.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1263–1272. JMLR. org.

Glorot, X.; Bordes, A.; and Bengio, Y. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 315–323.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

Kipf, T.; Fetaya, E.; Wang, K.-C.; Welling, M.; and Zemel, R. 2018. Neural relational inference for interacting systems. *arXiv preprint arXiv:1802.04687*.

Kool, W., and Welling, M. 2018. Attention solves your tsp. *arXiv preprint arXiv:1803.08475*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. S. 2016. Gated graph sequence neural networks. In *International Conference on Learning Representations*.

Li, Z.; Chen, Q.; and Koltun, V. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, 536—-545.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2018. Adaptive planner scheduling with graph neural networks. *CoRR* abs/1811.00210.

Morris, P., and Muscettola, N. 2005. Temporal dynamic controllability revisited. In *Proceedings of the* 22^{nd} *National Conference on Artificial Intelligence*.

Morris, P. 2014. Dynamic controllability and dispatchability relationships. In *Proceedings of the IInternational Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 464 – 479.

Osanlou, K.; Bursuc, A.; Guettier, C.; Cazenave, T.; and Jacopin, E. 2019. Optimal solving of constrained path-planning problems with graph convolutional networks and optimized tree search. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3519–3525. IEEE.

Rusek, K.; Suárez-Varela, J.; Mestres, A.; Barlet-Ros, P.; and Cabellos-Aparicio, A. 2019. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *Proceedings of the 2019 ACM Symposium on SDN Research*, 140–151.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7715–7723.

Tsamardinos, I. 2002. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence*, 97 – 108.

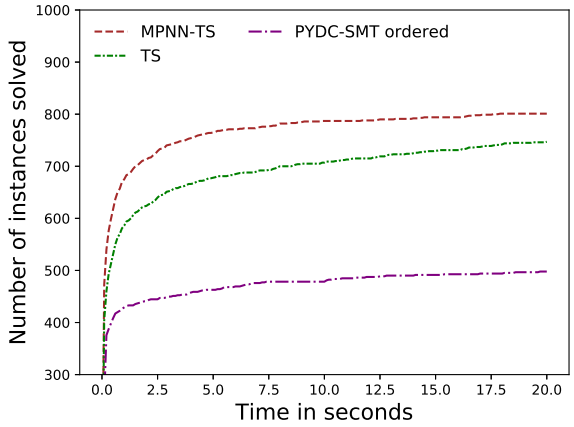
Vidal, T., and Fargier, H. 1999. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence* 11(1):23 – 45.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Yu, P. S. 2019. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*.

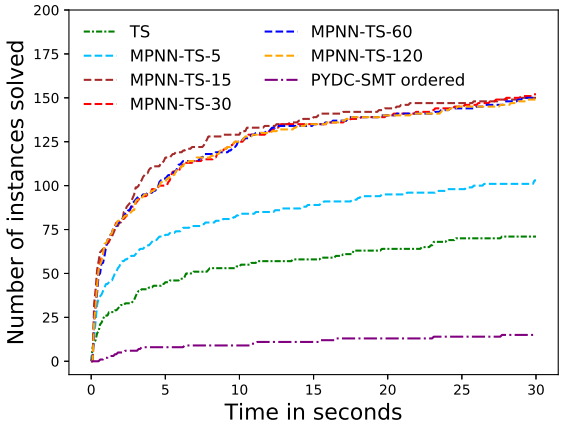
Xu, Z.; Tang, J.; Meng, J.; Zhang, W.; Wang, Y.; Liu, C. H.; and Yang, D. 2018. Experience-driven networking: A deep reinforcement learning based approach. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, 1871–1879. IEEE.

11 Appendix

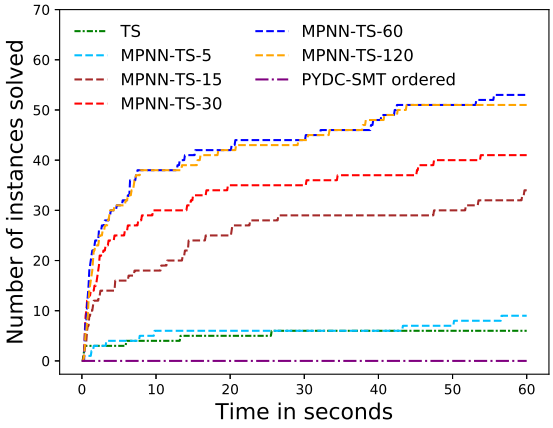
11.1 Plots



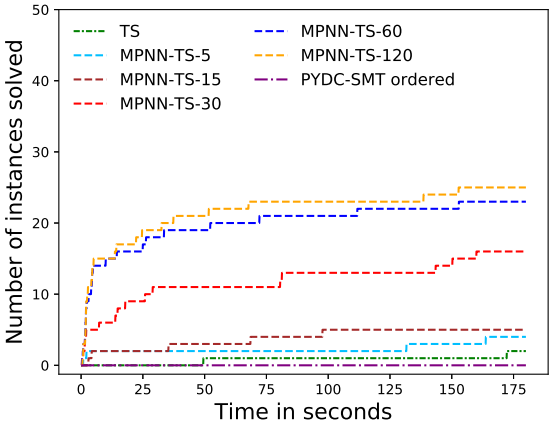
(a) Experiments on (Cimatti, Micheli, and Roveri 2016)'s benchmark from which the DTNs and STNs have been removed. The X-axis represents the allocated time in seconds and the Y-axis the total number of instances that each solver can solve within the corresponding allocated time. Timeout is set to 20 seconds per instance.



(b) Experiments on benchmark B_1. Axes are the same as in figure 6a. Timeout is set to 30 seconds per instance.



(c) Experiments on benchmark B_2. Axes are the same as in figure 6a. Timeout is set to 60 seconds per instance.



(d) Experiments on benchmark B_3. Axes are the same as in figure 6a. Timeout is set to 180 seconds per instance.

Figure 6: **Summary of experiments on benchmarks**

11.2 Simplified Example

Figure 7 is a simplified example of a TDC strategy of the example DTNU from (Cimatti, Micheli, and Roveri 2016).

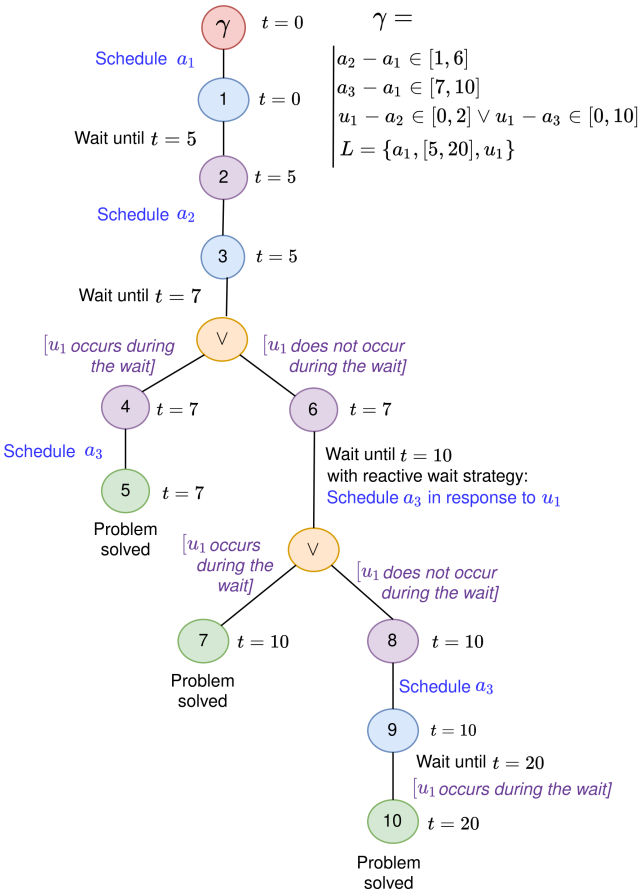


Figure 7: **Simplified TDC strategy of a DTNU** Γ**.** For space reasons, we only give a summarized copy of the strategy found. Branches leading to unsolved cases are excluded, and we do not include *d-OR* , *w-OR* , *AND* and *WAIT* nodes. The node γ is the original DTNU. Other nodes are sub-DTNUs, except the $\vee$ node which aims to list transitional possibilities, and should be interpreted in the figure as an *AND* node.

11.3 Truth Value Propagation Algorithm

We present in this section Algorithm 1. This algorithm is called to propagate a truth value in the tree. The propagation is done in an ascending way: truth values are inferred from the leaves of the tree towards the root.

Algorithm 1 Truth Value Propagation

```
1:  function PROPAGATETRUTH(TREENODE ψ)
2:      ω ← parent(ψ)                                          ▷ [1]*
3:      if ω = null then
4:          return
5:      if isDTNU(ω) or isWAIT(ω) then                         ▷ [2]*
6:          ω.truth ← ψ.truth
7:          propagateTruth(ω)
8:      else if isOR(ω) then                                   ▷ [3]*
9:          if ψ.truth = True then
10:             ω.truth ← True
11:             propagateTruth(ω)
12:         else
13:             if ∀σ_i, σ_i.truth = False then                ▷ [4]*
14:                 ω.truth ← False
15:                 propagateTruth(ω)
16:     else if isAND(ω) then                                  ▷ [5]*
17:         if ψ.truth = False then
18:             ω.truth ← False
19:             propagateTruth(ω)
20:         else
21:             if ∀σ_i, σ_i.truth = True then                 ▷ [4]*
22:                 ω.truth ← True
23:                 propagateTruth(ω)
```

[1]∗ parent(x): Returns the parent node of x, $null$ if none.
[2]∗ isDTNU(x): Returns *True* if x is a DTNU node, *False* otherwise; isWait(x): Returns *True* if x is a *WAIT* node, *False* otherwise.
[3]∗ isOR(x): Returns *True* if x is an *d-OR* or *w-OR* node, *False* otherwise.
[4]∗ σ_i: Child number i of ω. For a *d-OR* or *w-OR* node, in the case where ψ is *false* but not all other children of ω are *false* the propagation stops. Likewise, for an *AND* node and in the case where ψ is *true* but not all other children of ω are *true* , the propagation stops.
[5]∗ isAnd(x): Returns *True* if x is an *AND* node, *False* otherwise.

11.4 Tree Search Algorithm

We give the simplified pseudocode for the tree search in Algorithm 2.

Algorithm 2 Tree Search

function EXPLORE(TREENODE ψ)
  if $parent(\psi).truth \neq unknown$ **then**
    return
  if isDTNU(ψ) **then**
    updateConstraints(ψ) ▷ [6]*
    if IsLeaf(ψ) **then** ▷ [7]*
      propagateTruth(ψ)
      return
    Create *d-OR* child ψ'
    explore(ψ')
  if isOR(ψ) **then**
    Create list of all children Ψ' ▷ [8]*
    for $\psi' \in \Psi'$ **do**
      explore(ψ')
  if isAND(ψ) **then**
    Create list of all children Ψ' ▷ [9]*
    for $\psi' \in \Psi'$ **do**
      explore(ψ')
  if isWAIT(ψ) **then**
    create *w-OR* child ψ'
    explore (ψ')
function MAIN(DTNU γ)
  explore(γ)
  if $\gamma.truth = True$ **then**
    return $True$
  else
    return $False$

[6]* updateConstraints(x): Updates the constraints of DTNU node x.
[7]* isLeaf(x): Sets the truth value of x to *true* and returns *true* if all constraints are satisfied. Sets the truth value to *false* and returns *true* if a constraint is violated. If no truth value can be inferred at this stage with the updated constraints, a second check is run to determine if all uncontrollable timepoints have occurred. If so, the corresponding DTN is solved, the truth value of x is updated accordingly, and the function returns *true* . Otherwise, no logical outcome can be inferred for the current state of the constraints because there remains at least one uncontrollable timepoint and this function returns *false* .
[8]* If this is a *d-OR* node, the list Ψ' contains all the children DTNU nodes resulting from either the decision of scheduling a controllable timepoint, or the *WAIT* node resulting from a wait if available. If this is a *w-OR* node, Ψ' contains all AND_{R_j} nodes, each of which possess a reactive wait strategy R_j
[9]* Here, the list Ψ' contains all DTNUs resulting from all possible combinations $\Lambda_1, \Lambda_2, ..., \Lambda_q$ of uncontrollable timepoints which have the potential to occur during the current wait.

11.5 Wait Period

Figure 8 gives an example of the third rule used to compute a wait duration.

11.6 Optimization Rules

The following rules are added to make branch cuts when possible.

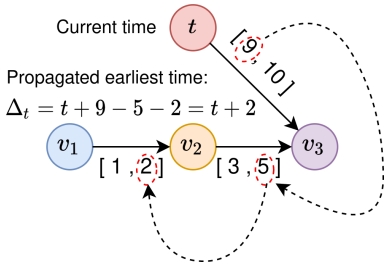


Figure 8: **Application of the 3^{rd} rule to determine a wait duration.** Current time is t. Variables v_1, v_2 and v_3 are timepoints. Here, v_2 is constrained to execute in the time interval $[1, 2]$ after v_1, v_3 in $[3, 5]$ after v_2 as well as in $[t + 9, t + 10]$. The rule suggests not to wait longer than 2 units of time at t: an execution of v_1 at $t + 2$, followed by an execution of v_2 at $t + 4$ opens a window of opportunity for v_3 to execute at $t + 9$.

Constraint Check. When a DTNU node is explored and the updated list of constraints C' is built according to §4.5, if a disjunct is found to be *false* , C' will no longer be satisfiable. All the subtree which can be developed from the DTNU will only have leaf nodes for which this is the case as well. Therefore, the search algorithm will not develop this subtree.

Symmetrical subtrees. Some situations can lead to the development of the exact same subtrees. A trivial example, for a given DTNU node at a time t, is the order in which a given combination of controllable timepoints $a_1, a_2, ..., a_k$ is taken before taking a wait decision. Regardless of what order these timepoints are explored in the tree before moving to a *WAIT* node, they will be considered executed at time t. Therefore, when taking a wait decision, it is checked that all preceding controllable timepoints executed before the previous wait are a combination of timepoints that has not been tested yet.

Truth Checks. Before exploring a new node for which the truth attribute is set to *unknown*, the truth attribute of the parent node is also checked. The node is only developed if the parent node's truth attribute is set to *unknown*. In this manner, when children of a tree node are being explored (depth-first) and the exploration of a child node leads to the assignment of a truth value to the tree node, the remaining unexplored children can be left unexplored.

11.7 Message Passing Layer

We use message passing layers that take as input a graph where nodes and edges possess features and return the graph with new node features. We detail pseudocode of a message passing layer applied to a graph $\mathcal{G} = (\mathcal{K}, \mathcal{E})$ in Algorithm 3.

11.8 Learning Implementation Details

Our MPNN architecture is made of 5 graph convolutional layers from (Gilmer et al. 2017). Each layer has a residual skip connection to the preceding layer (He et al. 2016), 32 abstract node features and a different two-layer MLP (multilayer perceptron) which has 128 neurons in its hidden layer. In addition, we use batch normalization after each graph

Algorithm 3 Message Passing Layer

function MSGPASS(GRAPH $\langle(\mathcal{K}, \mathcal{E}), (H_\kappa, X_\epsilon, X_\rho)\rangle$) ▷ [10]*
  $H'_\kappa(\cdot,\cdot) \leftarrow 0$ // *Initialize new node features matrix*
  for all $\kappa_i \in \mathcal{K}$ **do**
    $h'_i \leftarrow 0$ // *Initialize new features for* κ_i
    for all $\kappa_j \in \mathcal{K}$ **do**
      if $X_\epsilon(\kappa_i, \kappa_j) = 1$ **then**
        $\alpha \leftarrow X_\rho(\kappa_i, \kappa_j)$
        $h \leftarrow H_\kappa(\kappa_j, \cdot)$
        $h'_i \leftarrow h'_i + MLP(\alpha)h$ ▷ [11]*
    $H'_\kappa(\kappa_i, \cdot) \leftarrow h'_i$ // *Assign new features for* κ_i
  return $\langle(\mathcal{K}, \mathcal{E}), (H'_\kappa, X_\epsilon, X_\rho)\rangle$

[10]* $H_\kappa(\kappa_i, \cdot)$ returns a vector of current features for node κ_i; $X_\epsilon(\kappa_i, \kappa_j)$ returns 1 if $(\kappa_i, \kappa_j) \in \mathcal{E}$, 0 otherwise; $X_\rho(\kappa_i, \kappa_j)$ returns a vector of current features for edge (κ_i, κ_j).

[11]* MLP represents a multi-layer perceptron mapping input edge features to a matrix of dimension num-output-node-features x num-input-node-features. Moreover, h is of dimension num-input-node-features x 1. The matrix multiplication therefore results in a vector of size num-output-node-features.

layer and apply the $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$ activation function. The input of the MPNN is the graph conversion of a DTNU. Figure 9 illustrates an example of graph conversion. We use 10 different edge distance classes: $0 : [0, 0.1)$, $1 : [0.1, 0.2)$, ..., $9 : [0.9, 1]$. Training is done with the *adagrad* optimizer (Duchi, Hazan, and Singer 2011) and an initial learning rate 10^{-4} on a dataset comprised of $30K$ instances generated as described in §6. We split the data into a training set comprised of $25K$ instances and a cross-validation set comprised of $5K$ instances. We add a dropout regularization layer with a *keep rate* 0.9 before the output layer to reduce overfitting.

11.9 Architecture Comparison

We study the impact of the design choices of the MPNN architecture on performance. To this end we compare different architectures of MPNN by varying depth and width (number of abstract node features per layer) and train them on the training set created in §6. We also assess the added value of residual skip connections to preceding layers. We create a benchmark of 400 DTNU instances, each of which has 20 to 25 controllable timepoints and up to 3 uncontrollable timepoints. We solve them using the tree search guided by each of these MPNN architectures. We limit the use of the MPNN architectures to a maximal depth of 50 (*d-OR* node-wise). Results are shown in Figure 10. We note the smallest network is too small to learn efficiently and performs poorly. Three-layer networks perform better. Wider networks perform slightly better for the same depth, black network 32 vs. green network 16. Overall, medium-depth networks of 5 layers work best. Residual connections lead to slight but steady gains. Interestingly, deeper networks (8+ layers) display lower scores compared to more shallower variants (5 layers), suggesting depth performance saturation. The quantity of training data can however be a limiting factor: we assume the optimal architecture to be actually deeper.

$$\gamma = \begin{vmatrix} a_2 - u_1 \in [0, 1] \\ a_2 \in [0, 1] \vee a_2 \in [1.5, 3] \\ L = \{a_1, [0, 2], u_1\} \end{vmatrix}$$

$$\gamma' = \begin{vmatrix} a_2 - u_1 \in [0, 0.33] \\ a_2 \in [0, 0.33] \vee a_2 \in [0.5, 1] \\ L = \{a_1, [0, 0.66], u_1\} \end{vmatrix}$$

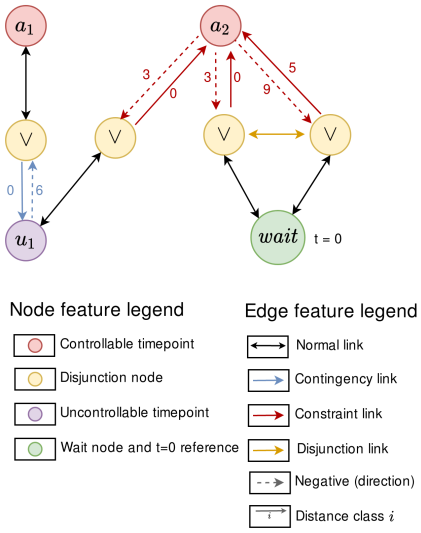


Figure 9: **Conversion of a DTNU γ into a graph.** γ' is the normalized DTNU. Edge distances are expressed as distance classes. To distinguish between lower and upper bounds in intervals, we introduce an additional *negative directional sign* feature.

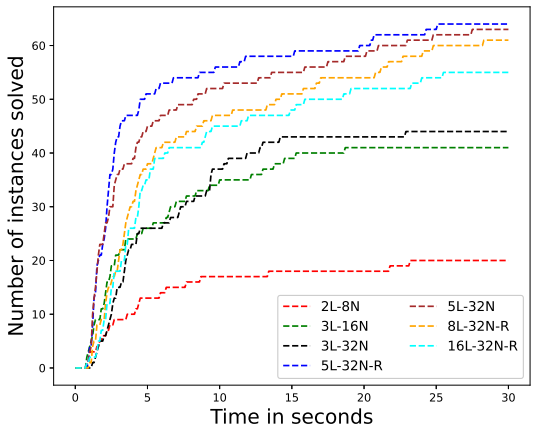


Figure 10: **Comparison of different MPNN architectures.** The notation XL-YN refers to an MPNN with X layers and Y abstract node features per layer. The "-R" tag refers to the presence of residual layers. Experiments are done on a DTNU benchmark containing 400 instances with 20 to 25 controllable timepoints and up to 3 uncontrollable timepoints per DTNU. Timeout is set to 30 seconds per DTNU instance.

Synthesis of Search Heuristics for Temporal Planning via Reinforcement Learning

Andrea Micheli and Alessandro Valentini
Fondazione Bruno Kessler, Trento, Italy
{amicheli,alvalentini}@fbk.eu

Abstract

Automated temporal planning is the problem of synthesizing, starting from a model of a system, a course of actions to achieve a desired goal when temporal constraints, such as deadlines, are present in the problem. Despite considerable successes in the literature, scalability is still a severe limitation for existing planners, especially when confronted with real-world, industrial scenarios.

In this paper, we aim at exploiting recent advances in reinforcement learning, for the synthesis of heuristics for temporal planning. Starting from a set of problems of interest for a specific domain, we use a customized reinforcement learning algorithm to construct a value function that is able to estimate the expected reward for as many problems as possible. We use a reward schema that captures the semantics of the temporal planning problem and we show how the value function can be transformed in a planning heuristic for a semi-symbolic heuristic search exploration of the planning model. We show on two case studies how this method can be used to extend the reach of current temporal planning technology with encouraging results.

1 Introduction

Automated temporal planning concerns the synthesis of strategies to reach a desired goal with a system that is formally specified by providing an initial condition together with the possible actions that can drive it in presence of temporal constraints. In this context, actions become intervals (instead of being instantaneous as in classical planning) that have a duration (possibly subject to metric constraints). Similarly, plans are no longer simple sequences of actions, but they are schedules. Automated temporal planning received considerable attention in the literature, and the definition of the standard PDDL 2.1 language (Fox and Long 2003) fueled the research of effective search-based techniques to solve the problem (Coles et al. 2010; Eyerich, Mattmüller, and Röger 2012; Rankooh and Ghassem-Sani 2015).

Despite considerable success stories, scalability is still a major hindrance for the adoption of automated temporal planning in real-world industrial scenarios. For example, the experiments reported in (Micheli and Scala 2019) and, more recently in (Valentini, Micheli, and Cimatti 2020) show how existing tools are unable to cope with very small and simple industrial problems when rich temporal constraints need to be modeled. From a practical standpoint, in many scenarios one wants to have a planner that is able to quickly solve problems on the same domain: for this reason, many practitioners resort to domain-dependent planners.

In order to mitigate this issue and retain a domain-independent framework, we propose to leverage recent advances in model-free reinforcement learning (RL), in particular Value Iteration using Neural Networks, to automatically construct temporal planning heuristics for a specific domain. Ideally, we want to take a temporal planning domain, analyze it off-line using RL and produce a heuristic function that allows a planning technique to extend the coverage of solved problems in that domain. To the best of our knowledge, no previous work addressed the problem of learning heuristics for temporal planning.

In this paper, we present a domain-independent learning and planning framework that, given a planning domain and a set of training problems (not solution plans), synthesizes a temporal planning heuristic for problems in the same domain. We empirically show how this method outperforms existing symbolic heuristics on two use-case domains with rich temporal constraints. Our results emphasize how this approach truly requires a combination of learning and reasoning, because the learned policy alone and the purely-symbolic planner are incapable of reaching the performance of the symbolic planner equipped with the learned heuristic.

2 Problem Definition

We start by defining the syntax of temporal planning: we formalize an abstract syntax adherent to the ANML (Smith, Frank, and Cushing 2008) fragment supported by our planner (Valentini, Micheli, and Cimatti 2020) using a lifted representation to separate domain and problem specifications.

For the sake of simplicity, we formalize a language that is un-typed and with Boolean predicates only; our implementation supports the entire ANML typing system and finite- and infinite-codomain functions.

Definition 2.1. *An **atom** is a tuple $\langle p, \vec{v} \rangle$ where p is a predicate with arity n and $\vec{v}$ is a vector of n variables.*

In our temporal language specification, conditions and effects can be declared to happen at any time within the duration of an action, and conditions can be durative, so they are

associated with an interval of times (we called this feature "Intermediate Conditions and Effects").

Definition 2.2. *An **effect** on atom a at relative time τ is a tuple $\langle \tau, a \rangle$ where τ is either* START $+ k$ *or* END $- k$ *with $k \in \mathbb{Q}_{>=0}$. A **condition**[1] on atom a in the relative interval $[\tau_1, \tau_2]$ is a tuple $\langle [\tau_1, \tau_2], a \rangle$ where τ_i is either* START $+ k_i$ *or* END $- k_i$ *with $k_i \in \mathbb{Q}_{>=0}$.*

Then, a planning domain is a set of predicates and actions.

Definition 2.3. *A **planning domain** is a tuple $\langle P, A \rangle$ where P is a finite set of predicates; A is a finite set of actions, each action a has a minimal (d_a^{min}) and maximal (d_a^{max}) duration, a set of parameter variables $\vec{v}$, a set of conditions C_a, a set of add effects E_a^+ and a set of delete effects E_a^- (with $E_a^+ \cap E_a^- = \emptyset$). All the atoms appearing in the definition of a can only use variables appearing in $\vec{v}$.*

We define a ground atom as an atom where all the variables are assigned to an object.

Definition 2.4. *A **ground atom** is a tuple $\langle a, \vec{o} \rangle$ where $a \doteq \langle p, \vec{v} \rangle$ is an atom and $\vec{o}$ is a vector of n objects o_i with $n = arity(p)$.*

Finally, a planning problem is composed of a finite set of objects, an initial state and a goal to reach.

Definition 2.5. *A **planning problem** for a planning domain $\langle P, A \rangle$ is a tuple $\langle O, I, G \rangle$ where O is a finite set of objects o_i; I and G are sets of ground atoms over predicates in P.*

We indicate a *planning instance* as a pair of a planning domain $\mathcal{D}$ and a problem P_i ($\langle \mathcal{D}, P_i \rangle$).

We do not report the full semantics of temporal planning; for the sake of this paper it suffices to say that a planning instance can be grounded and the ground semantics is the usual one: we want to find a valid simulation of the ground system starting from the initial state and terminating in a goal state. This semantics can be found in (Valentini, Micheli, and Cimatti 2020). Moreover, in this paper we disregard action self-overlapping (Gigante et al. 2020); that is, we forbid an instance of an ground action to overlap in time with another instance of the same ground action.

In order to solve a ground instance, TAMER (Valentini, Micheli, and Cimatti 2020) searches an interleaving of events (also called happenings or time-points) that represent the discrete changes of state in a plan ensuring that the abstract sequence of events can be lifted to a plan by scheduling the temporal constraints. TAMER represents search states as follows and performs a search in the space of the possible reachable states starting from the initial state. The transitions considered by the planner for a planning problem P_i (called *events* and indicated as $events(P_i)$) are either instantiations of new actions or expansions of time-points, each indicating an effect, the starting of a condition or its ending.

Definition 2.6. *A **search state** is a tuple $\langle \mu, \delta, \lambda, \chi, \omega \rangle$ s.t.:*
• μ records the ground predicates that are true in the state;
• δ is a multiset of ground predicates, representing the active durative conditions to be maintained valid;

[1]We only formalize closed condition intervals; open and semi-open intervals are supported by our implementation.

Algorithm 1 TAMER search algorithm

```
1: procedure SEARCH(w)
2:     i ← GETINIT( );  g(i) ← 0;  Q ← NEWPRIORITYQUEUE( )
3:     PUSH(Q, i, h(i))
4:     while c ← POPMIN(Q) do
5:         if |c.λ| = 0 then return GETPLAN(c.χ)
6:         else
7:             for all s ∈ SUCC(c) do
8:                 g(s) ← g(c) + 1
9:                 PUSH(Q, s, (1 − w) × g(s) + w × h(s))
```

• λ is a list of lists of time-points. It constitutes the "agenda" of future commitments to be resolved.
• χ is a Simple Temporal Network (STN) defined over time-points that stores and checks the metric and precedence temporal constraints;
• ω is the last time-point evaluated in this search branch.

We indicate the set of possible states for a given instance $\langle \mathcal{D}, P_i \rangle$ as $\mathcal{S}_{\langle \mathcal{D}, P_i \rangle}$. The exploration performed by TAMER is detailed in algorithm 1: SUCC indicates the possible successor states of a given state (see (Valentini, Micheli, and Cimatti 2020) for the details).

For the purpose of this paper, we need to define a set of problems of interest for a given domain: the objective of our learning technique will be to automatically synthesize a heuristic to guide a planner for efficiently solve any instance in the identified set. We make two assumptions on this set. First, we require the set to be finite: in principle one could have an infinite set and a sampler, but some details of our learning algorithm currently assume a finite set of problems. Second, we assume the number of objects is bounded: this is needed because we use a feed-forward neural network that requires a known input dimension to be constructed. For this reason, we need to assume a maximum number of objects that results in a maximum number of ground predicates and in turn a maximum number of inputs for the neural network.

Definition 2.7. *A **bounded planning problem set** with at most k objects for a planning domain $\mathcal{D} \doteq \langle P, A \rangle$ written $\mathcal{P}_{\mathcal{D}}^k$ is a finite set of planning problems $P_i \doteq \langle O_i, I_i, G_i \rangle$ for $\mathcal{D}$ such that each $|O_i| \leq k$.*

In essence, our objective consists in synthesizing a heuristic function that can guide the search of TAMER. The heuristic takes in input a search state and the description of the problem being solved (i.e. it takes the state of the search, the goal formulation and the set of objects, the initial state is ignored).

Definition 2.8. *The **optimal distance heuristic** for a bounded planning problem set $\mathcal{P}_{\mathcal{D}}^k$ is a function*

$$h^*_{\mathcal{P}_{\mathcal{D}}^k} : \left(\bigcup_{P_i \in \mathcal{P}_{\mathcal{D}}^k} \mathcal{S}_{\langle \mathcal{D}, P_i \rangle} \right) \times \mathcal{P}_{\mathcal{D}}^k \to \mathbb{R}$$

s.t. for each $P_i \in \mathcal{P}_{\mathcal{D}}^k$ and each state $s \in \mathcal{S}_{\langle \mathcal{D}, P_i \rangle}$, $d \doteq h^*(s, P_i)$ is the minimum number for which $\text{SUCC}^d(s)$ *is a goal state.*

The aim of this paper is to automatically learn an approximation of h^* given a temporal planning domain and a training bounded planning problem set.

3 Planning Heuristics as Reinforcement Learning

In order to learn an approximation of h^*, we first cast the learning problem as a model-free Reinforcement Learning (RL) problem, in which an instance is non-deterministically picked from the set $\mathcal{P}^k_{\mathcal{D}}$ without the agent knowing about the choice and then an episodic RL algorithm is started to synthesize a value function.

We start by defining the Markov Decision Process (MDP) over which we will run our RL algorithm.

Definition 3.1. *A Markov Decision Process (MDP) is a tuple $\mathcal{M} \doteq \langle S, A, T, R, s_0\rangle$ where S is a set of states, A is a set of actions, $T : S \times A \to p(S)$ is the transition function that given a state and an action returns a probability distribution for the successor state, $R : S \times A \times S \to \mathbb{R}$ is the immediate reward function and s_0 is the initial state.*

In RL, we want to construct (an estimation of) the optimal value function for an MDP $\mathcal{M}$. We assume to interact with the environment through a policy $\pi : S \to A$ that selects the action to be applied in each state. After specifying an action a_t in state s_t, the environment returns a state $s_{t+1} \sim T(s_t, a_t)$ and the reward $r_t \doteq R(s_t, a_t, s_{t+1})$. The goal of RL is to find the policy yielding the maximal cumulative reward discounted by γ, defined below.

Let the state-action value of a policy π be as follows.

$$Q^\pi_{\mathcal{M}}(s, a) \doteq \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid s_t = s, a_t = a \right]$$

The value function is given by: $V^\pi_{\mathcal{M}}(s) \doteq \mathbb{E}\left[Q_{\mathcal{M}}(s, \pi(s))\right]$. The objective of RL is to find the optimal policy $\pi^*(s) \doteq \arg\max_\pi Q_{\mathcal{M}}(s, \pi(s))$. Moreover, in this paper, we are interested in computing the optimal value function ($V^*_{\mathcal{M}} \doteq V^{\pi^*}_{\mathcal{M}}$) for extracting heuristic estimates.

Definition 3.2. *Given a bounded planning problem set $\mathcal{P}^k_{\mathcal{D}}$, its **MDP encoding** $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$ is the MDP $\langle S, A, T, R, \vdash\rangle$ where:*

- $S \doteq \{\vdash\} \cup \bigcup_{P_i \in \mathcal{P}^k_{\mathcal{D}}} \langle \mathcal{S}_{\langle \mathcal{D}, P_i\rangle}, P_i\rangle$;
- $A \doteq \{\xi\} \cup \bigcup_{P_i \in \mathcal{P}^k_{\mathcal{D}}} events(\langle \mathcal{D}, P_i\rangle)$;
- $T(s, a) \doteq \begin{cases} \{\langle I_{P_i}, \frac{1}{|\mathcal{P}^k_{\mathcal{D}}|}\rangle \mid P_i \in \mathcal{P}^k_{\mathcal{D}}\} & \text{if } s = \vdash,\ a = \xi \\ \{\langle a[s], 1\rangle\} & \text{if } s \neq \vdash \end{cases}$

where I_{P_i} indicates the initial search state of problem P_i and $a[s]$ indicates the (unique) successor state of s using action a. Here, we encoded the successor states using discrete uniform probability distributions (we wrote pairs of successor states with the associated probability).

- $R(s, a, s') \doteq \begin{cases} 1 & \text{if } s' = \langle s_i, \langle O, I, G\rangle\rangle \text{ and } s_i \models G \\ -1 & \text{if } \nexists b.\ s'' = b[s'] \\ 0 & \text{otherwise.} \end{cases}$

Intuitively, we are defining a MDP in which a first, probabilistic transition is used to uniformly select a problem P_i to be solved from the set $\mathcal{P}^k_{\mathcal{D}}$; such a transition drives the MDP in a state where one problem to be solved is identified and such a problem is in its initial state I_{P_i}. From this state on, the MDP is fully deterministic and the search space is homomorphic to the planning space for problem P_i (that is, all transitions are deterministic and the successor function changes the first element of the state tuple according to the successor function of the planning problem). Note that since we disallowed action self-overlapping, the decision to take a certain event is unambiguous as there can be at most one action instance running at each time. The reward of the encoding MDP is shaped to give a 1 when the system is in a state over problem P_i that satisfy the problem goals, a -1 in dead-ends and 0 everywhere else. This makes the maximal possible cumulative reward to be 1 assuming that after the goal is reached the planning successor function deadlocks. Figure 1 depicts the encoding MDP state space and rewards.

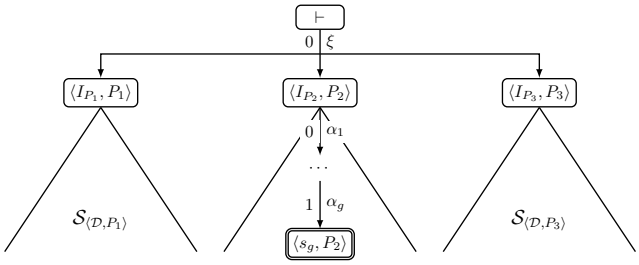


Figure 1: The state space and rewards of $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$.

Note that the resulting MDP is a faithful representation of the set of planning instances we want to solve, no abstraction is taken. If we could solve this MDP, we would be able to solve all the planning instances with the resulting policy without search. At this point, we can introduce the main theorem summarizing the basic intuition of this work: we can transform the optimal value function for the MDP into the optimal heuristic for all the planning problems.

Theorem 3.1. *For a bounded planning problem set $\mathcal{P}^k_{\mathcal{D}}$ the following equation holds.*

$$h^*_{\mathcal{P}^k_{\mathcal{D}}}(s) = \begin{cases} \log_\gamma(V^*_{\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}}(s)) & \text{if } V^*_{\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}}(s) > 0 \\ \infty & \text{otherwise} \end{cases}$$

Proof. (Sketch) The MDP $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$ is deterministic except for the first action ξ starting from state $\vdash$ that is however not needed for the heuristic since $\vdash$ is not a search state of any problem. We are therefore interested only in the value of the other states that is unaffected by action ξ since our MDP is a tree.

On a deterministic system with the reward shape of $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$, the optimal policy is the policy reaching a goal state in the minimum number of steps. Let $\langle s_0, \cdots, s_g\rangle$ be the optimal path from state s_0 to the nearest state satisfying a goal s_g. The discounted reward in s_i clearly is $V^*_{\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}}(s_i) = \gamma^{g-i}$, and the distance to the goal is $h^*_{\mathcal{P}^k_{\mathcal{D}}}(s_i) = g - i$. If a state s cannnot reach any goal, then $V^*_{\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}}(s) <= 0$. Hence, we can retrieve the distance from the discounted reward as per the theorem statement. □

Algorithm 2 Vectorization of an STN χ

procedure STN2VECTOR(χ)
  $\vec{r} \leftarrow \langle 0 \text{ for all actions } a_i \rangle$; $\tau \leftarrow$ GETMINMAKESPANSOLUTION(χ)
  $lastSafe \leftarrow 0$ ▷ A "safe" state is a state where no action is running
  $balance \leftarrow 0$ ▷ The difference between the started and terminated actions
  for all time points tp sorted by $\tau[tp]$ **do**
    if tp is a starting of an action **then** $balance \leftarrow balance + 1$
    else if tp is the termination of an action **then** $balance \leftarrow balance - 1$
    if $balance = 0$ **then**
      $\vec{r} \leftarrow \langle 0 \text{ for all actions } a_i \rangle$
      $lastSafe \leftarrow \tau[tp]$
    else
      $\vec{r}[action(tp)] \leftarrow \tau[tp] - lastSafe$
    if $tp = \omega$ **then break** ▷ ω is the last scheduled time-point
  return $\vec{r}$

4 Reinforcement Learning Algorithm

In this section, we detail a dedicated RL algorithm derived from classical Value Iteration that uses a Neural Network to estimate the optimal value function $V^*_{\mathcal{M}_{\mathcal{P}^k_\mathcal{D}}}$. The overarching idea is to use RL to estimate $V^*_{\mathcal{M}_{\mathcal{P}^k_\mathcal{D}}}$ and from that estimation, derive an estimation of $h^*_{\mathcal{P}^k_\mathcal{D}}$.

Problem scaling and vector representation. The first ingredient needed for our algorithm is to create a uniform vector representation of the MDP state. To do so, we first need to scale the state representation so that all the problems in $\mathcal{P}^k_\mathcal{D}$ can be represented uniformly despite the fact that the number of actions and fluents can be different from one another. To overcome this issue, we exploit the bound k on the number of objects. For each object o_i, we introduce a fresh Boolean constant[2] $o_i^\exists$ that is set to true if the object o_i exists in an instance. In this way, all the instances can be represented uniformly by considering all the possible k objects, by adding a precondition $o_i^\exists$ to each action where o_i appears and by setting the initial value of all predicates depending on a non-existing o_i to false. This simple transformation, essentially scales any problem in $\mathcal{P}^k_\mathcal{D}$ to a problem with exactly k objects and a fixed number of actions, that has the same plans of the original problem.

At this point, we are left with a set of problems that can be grounded, resulting in a consistent number of fluents. The neural network we will use to represent the RL policy requires a vector representation of a state of the MDP $\mathcal{M}_{\mathcal{P}^k_\mathcal{D}}$. Given a search state $\langle \mu, \delta, \lambda, \chi, \omega \rangle$ and a problem P_i, we define the vectorization of the MPD state as follows. First, we vectorize the predicate values (i.e. the μ part of the state), we pick a fixed ordering for the ground predicates of the biggest possible problem in $\mathcal{P}^k_\mathcal{D}$. Note that the cardinality of the ground states is exactly k^x with $x \doteq \sum_{p \in P} arity(p)$. We set the input vector value to 1 (resp. 0) if the corresponding ground predicate is true (resp. false). The second part of the vector is a representation of the status of the events (i.e. the λ). For each possible ground action we have a vector element set to the size of the corresponding list of time-points in λ or to 0 if the action is not started in the current state. The third part of the input vector contains the constants of the problem, i.e. the fluents that are never changed by effects. Constants are encoded as normal fluents. The fourth part of the vector encodes the goals. For each fluent we have an entry that is either set to the desired goal value of the predicate/fluent (using the same encoding of the fluents section) or to -1 to indicate that we do not care for this value in this problem. The fifth and final part of the input vector encodes the temporal part of the state and can be seen as a summary of the STN χ. Since the STN grows while planning search unfolds, we need a way to compress as much information as possible in form of a fixed-size vector. We use a simple encoding that captures the time passed in the minimal-makespan solution of the current STN χ since a running action has been started. This is formally reported in algorithm 2. The final vector for a state s (indicated as $\vec{s}$) is the concatenation in a single, linear vector of all the five vector sections above.

[2] A constant is a fluent that is assigned in the initial state and never changed. ANML explicitly supports constants.

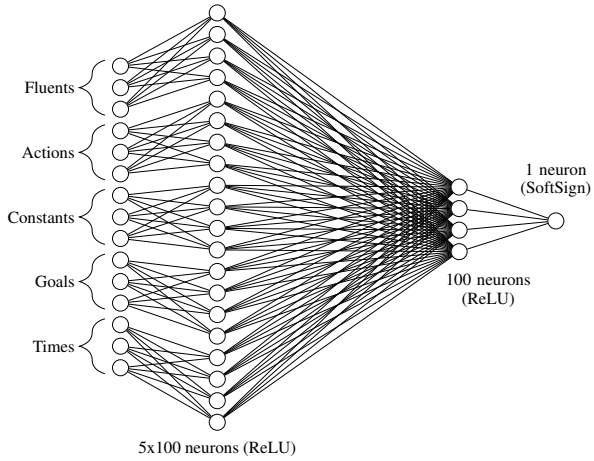


Figure 2: The neural network architecture.

Neural Network. Given the vectorization of a state, the neural network architecture we use is depicted in figure 2. We split the input vector into the five "sections" described above. For each of them, we have a dense layer with output size 100 and ReLU activation function. In this way, we obtain a first hidden layer of 500 neurons. Then, we have a second layer with output size 100 and ReLU activation function. Finally, we compute the output of the network using a single neuron densely connected with the second hidden layer that uses a softsign ($y = \frac{x}{1+|x|}$) activation function.

The neural network will be trained to approximate the optimal value function $V^*_{\mathcal{M}_{\mathcal{P}^k_\mathcal{D}}}$. Note that the expected reward along any path must be in the range $[-1, 1]$ because of the reward shape of $\mathcal{M}_{\mathcal{P}^k_\mathcal{D}}$, hence the use of the softsign function to compress the values in the admissible range. To train the neural network, we use an Adam optimizer and the Mean Squared Error (MSE) loss function.

Learning Algorithm. The full RL algorithm scheme is reported in algorithm 3. The algorithm main function

Algorithm 3 Reinforcement Learning Algorithm

procedure RL2PLANHEURISTIC(tis, $N_{episodes}$)
  $V_{nn} \leftarrow$INITNN( ) ▷ Creates NN with the described architecture
  $mem \leftarrow$LIST( ) ▷ The algorithm experience memory
  $i2s \leftarrow \{i \rightarrow 0 \mid i \in tis\}$ ▷ maps i to # of times i was solved
  for $i \in 1, \ldots, N_{episodes}$ **do**
    $\langle s, goals\rangle = inst \leftarrow$PICKKEYINVPROPORTIONALLYTOVALUE($i2s$)
    $\langle done, solved\rangle \leftarrow \langle False, False\rangle$
    $\pi \leftarrow \langle s\rangle$
    while not $done$ **do**
      $\epsilon \leftarrow \epsilon_{max} \times e^{(\frac{\ln(\epsilon_{min}/\epsilon_{max})}{N_{episodes}} \times i)}$ ▷ Decay $\epsilon_{max} \rightarrow \epsilon_{min}$
      if RANDOM( ) $< \epsilon$ **then** ▷ With probability ϵ
        $\alpha \leftarrow$SELECTACTIONUSINGHEURISTIC(s)
      else
        $\alpha \leftarrow$SELECTACTIONUSINGPOLICY(V_{nn}, s)
      $\langle s', done, \rho\rangle \leftarrow$DOSTEP($\pi$, s, α, $inst$) ▷ Simulate α move
      APPEND(mem, $\langle s, \rho\rangle$)
      if $\rho[\alpha] = 1$ **then**
        $solved \leftarrow True$
      APPEND(π, $\langle s'\rangle$)
      $s \leftarrow s'$
    $V_{nn} \leftarrow$REPLAY(V_{nn}, mem) ▷ Do a learning step
    if $solved$ **then**
      $i2s[inst] \leftarrow i2s[inst] + 1$ ▷ Update solved # count for $inst$
  return V_{nn}

procedure PICKKEYINVPROPORTIONALLYTOVALUE(b)
  $V \leftarrow \{v \mid i \rightarrow v \in b\}$ ▷ Get the values of the map b
  $m \leftarrow$CEIL($1.1 \times$MAX(V)) ▷ Allow a 10% slack
  $t \leftarrow m \times |b| - (\sum_{v \in V} v)$ ▷ Factor to narmalize probabilities
  $perc \leftarrow \{i \rightarrow \frac{m-v}{t} \mid i \rightarrow v \in b\}$ ▷ Probability to pick each element i
  return RANDOMSELECTIONBASEDONPERCENTAGE($perc$)

procedure SELECTACTIONUSINGHEURISTIC(s)
  $h \leftarrow$EMPTYMAP( ) ▷ A map from successor states to their heuristic values
  for all $\alpha \in$GETAPPLICABLEEVENTS(s) **do**
    $s' \leftarrow$SIMULATEACTIONAPPLY(s, α)
    $h[\alpha] = h_{add}(s')$
  return PICKKEYINVPROPORTIONALLYTOVALUE(h)

procedure SELECTACTIONUSINGPOLICY(V_{nn}, s)
  $app \leftarrow$GETAPPLICABLEEVENTS(s)
  $ns \leftarrow \{\alpha \rightarrow \vec{s'} \mid s' =$ SIMULATEACTIONAPPLY$(s, \alpha), \alpha \in app\}$
  return $\arg\max_{\alpha \in app} V_{nn}(ns[\alpha])$

procedure DOSTEP(π, s, α, $inst$)
  $\rho \leftarrow \{\beta \rightarrow$ GETREWARD$(\pi, s, \beta, inst) \mid \beta \in GA_{inst}\}$
  $s' \leftarrow$SIMULATEACTIONAPPLY(s, α)
  $done \leftarrow (\rho[\alpha] = 1)$ **or** $|\pi| \geq$ GETMAXDEPTH( )
  return $\langle s', done, \rho\rangle$

procedure REPLAY(net, mem)
  $batch \leftarrow$SAMPLE(mem) ▷ Pick elements from memory to learn from
  $x \leftarrow \langle \vec{s} \mid \langle s, \rho\rangle \in batch\rangle$; $y \leftarrow$EMPTYLIST( )
  for all $\langle s, \rho\rangle \in batch$ **do**
    $app \leftarrow$GETAPPLICABLEEVENTS(s)
    $ns \leftarrow \{\alpha \rightarrow \vec{s'} \mid s' =$ SIMULATEACTIONAPPLY$(s, \alpha), \alpha \in app\}$
    $y_s \leftarrow \max_{(\alpha \rightarrow r \in \rho)}(r + \gamma \times net(ns[\alpha]))$ ▷ Update equation
    APPEND(y, MAX(y_s))
  $net \leftarrow$TRAINBATCH(net, x, y) ▷ Backpropagation learning
  return net

procedure GETREWARD(π, s, α, $inst$)
  $s' \leftarrow$SIMULATEACTIONAPPLY(s, α)
  if $s' \models goals$ **then return** 1
  else if GETAPPLICABLEEVENTS(s') $= \emptyset$ **then return** -1
  else ▷ c counts the sub-goals achieved for the first time by α
    $c \leftarrow |\{g \mid g \in$ GOALS$(inst), s' \models g, \forall s'' \in \pi.\ s'' \not\models g\}|$
    return $\frac{c}{|goals|} \times 10^{-5}$

RL2PLANHEURISTIC takes a set of training ground instances tis and a number of episodes to run for $N_{episodes}$; its goal is to evolve a value function represented as a neural network V_{nn} that approximates the optimal value function. The experience is collected in a finite-size memory mem that caches pairs $\langle s, \rho\rangle$; where s is a state and ρ is a mapping of all the applicable events in s to their immediate reward. Differently from a standard RL algorithm, we manipulate the probability of selecting a specific instance among the ones in the training set by favoring the ones that have been solved (i.e. that reached a reward of 1) less often. This amounts to dynamically adapting the probability distribution of the ξ transition in the MDP $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$. Concretely, we record for each planning instance, how many time it has been solved in the $i2s$ map and we use the PICKKEYINVPROPORTIONALLYTOVALUE function to select an instance for each episode. This function essentially computes the histogram of the solving times for each instance and picks an instance proportionally to the inverse of this histogram augmented by 10% to allow a non-zero probability of selecting each instance. This manipulation of the probabilities is used to focus the learning on instances that have been solved less often and are therefore likely to be more difficult. This is needed because of the nature of the planning problems: some might have short, simple plans while other can be hard; in the training set both these cases co-exist and we want to obtain a flexible policy rather than a policy highly optimized for the simple cases.

We use an exponential epsilon-decay strategy to balance between random exploration and policy exploitation, but we exploit the planning heuristic (h_{add} in our case) to skew the probabilities among the possible events. This is done in the SELECTACTIONUSINGHEURISTIC function that re-uses the PICKKEYINVPROPORTIONALLYTOVALUE function to randomly pick an action with a probability inversely proportional to the heuristic value[3].

The trajectory simulation is standard and uses the TAMER planning engine as a simulator. In the memory mem, we store for each state in the trajectory the reward of each possible successor state. We forcibly bound the length of the traces to a maximum depth given by the GETMAXDEPTH function: we want to avoid the exploration of very long (or even infinite) paths, in fact, by allowing an arbitrary number of steps we might get trapped in loops yielding 0 reward and never finish an episode. In the following, we indicate the maximum depth used to bound the paths as Δ_{RL}.

We use a reward function that is slightly adjusted with respect to the one presented in definition 3.2: in particular, we grant a small (10^{-5} in total) reward for the sub-goals (a sub-goal is an element of G) achieved for the first time in a trace and we give 0 reward for traces that reach the maximum depth. This is done by the GETREWARDFUNCTION that analyzes the trace and checks, for each sub-goal, if it is achieved for the first time or not. Note that this change has a small numerical impact on the expected reward and hence on theorem 3.1, but we picked a number that is small enough

[3]Since the heuristic estimates the distance to the goal, we prefer events leading to successor states having a small heuristic value.

to be practically negligible while giving useful intermediate reward signals.

The learning algorithm is then a standard value iteration with finite memory using the neural network V_{nn}; the pseudo-code is reported in function REPLAY. The function takes advantage of the determinism of the transitions in each ground planning instance. In fact, by removing the ξ transition from MDP $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$, the state space of each instance is fully deterministic and tree-shaped. For this reason, we omitted the learning rate (by implicitly setting it to 1) and we need no expectation operator on the outcome of α. The value iteration update rule (line 52) simply collapses to:

$$V_{i+1}(s) \leftarrow \max_{\alpha}(R(s, \alpha, s') + \gamma \times V_i(s'))$$

where α ranges over the applicable events in s and s' is the successor state of s obtained by applying α.

Planning Algorithm. The output of the learning algorithm is a policy that estimates the reward for $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$. We use this policy as a heuristic function in our planning algorithm according to theorem 3.1 with some practical adjustments to take into account the maximum exploration depth (Δ_{RL}) we fixed for the algorithm.

$$h_{nn}(s) \doteq \begin{cases} \min(\log_\gamma(V_{nn}(\vec{s})), \Delta_h) & \text{if } V_{nn}(\vec{s}) > 0 \\ \Delta_h & \text{if } V_{nn}(\vec{s}) = 0 \\ 2\Delta_h - \min(log_\gamma(-V_{nn}(\vec{s})), \Delta_h) & \text{otherwise} \end{cases}$$

Where $\Delta_h \geq \Delta_{RL}$. Intuitively, we exploit theorem 3.1 when $V_{nn}(\vec{s}) > 0$, but we clip the logarithm output to the maximum depth Δ_h because the RL exploration was limited to a depth of Δ_{RL}. Note that the output of V_{nn} is constrained between -1 and 1 excluded, so the logarithm in the first case is guaranteed to be positive (because $\gamma < 1$). Moreover, if the neural network returns 0, we return Δ_h as heuristic value and if it is negative (due to the dead-ends), we return a value that is between Δ_h and $2\Delta_h$. Note that this heuristic never returns ∞, as we cannot formally guarantee that a state is a dead-end (while h_{add} can sometimes determine that a state shall be pruned). Therefore, we use the range of number between Δ_h and $2\Delta_h$ to give informative results. The Δ_h constant used in this heuristic does not need to be equal to the one (Δ_{RL}) used in the RL algorithm, we just require that $\Delta_h \geq \Delta_{RL}$. This consideration is important because empirically, we discovered that using a larger value for Δ_h yields better results. This is probably due to the "flattening" of the heuristic value due to the $\min$ operators in the heuristic: the smaller Δ_h, the more values of $h_{nn}(s)$ get compressed to Δ_h, losing the possibility of discriminating among them.

5 Related Work

Several works aimed at combining learning with planning.

Macro-actions (Coles and Smith 2007; Botea et al. 2005) consist in the combination of several actions in a single one: creating "shortcuts" in the search-space. Case-based planning (Spalazzi 2001; Bonisoli et al. 2015) constructs a database of plans for a specific domain that can be used as a source of learned knowledge to efficiently solve new problems. Some authors (Asai and Fukunaga 2018) also focused on the problem of learning symbolic models from data.

The learning of heuristics to speed up the planner is the most related topic. To the best of our knowledge, no work currently addresses the problem of learning heuristics for temporal planning: only few papers deal with this problem in the case of classical planning. In their seminal work, de la Rosa, Olaya, and Borrajo use a case-based database to inform heuristics (de la Rosa, Olaya, and Borrajo 2007). Yoon, Fern, and Givan used machine-learning techniques to learn control policies that are then exploited in a classical, heuristic-search planner (Yoon, Fern, and Givan 2008). Another approach in this area is (Arfaee, Zilles, and Holte 2011), where the authors use the search spaces generated by employing one, weak classical planning heuristic to learn an incrementally better one. (Choudhury et al. 2018) aims at learning heuristic functions for robotic planning by imitation of a oracle used for training. (Virseda, Borrajo, and Alcazar 2013) uses machine learning to compose a fixed set of classical planning heuristics into one, single heuristic value for cost-based planning. Recently, (Ferber, Helmert, and Hoffmann 2020) showed a comprehensive hyper-parameter experimentation for the case of supervised-learning of a classical planning heuristic represented as a neural-network. Differently from all these previous works, this paper tackles expressive temporal planning with intermediate conditions and effects and provides a fully-automated technique to learn heuristics from simulations via RL. Moreover, we do not focus on a single instance or a group of instances with the same structure, but we allow for arbitrary sets of instances sharing the same domain and having a known upper-bound on the number of objects.

Also in the context of classical planning, some approaches aimed at learning domain-specific planners. (Spector 1994) used genetic programming to automatically code a planner for a specific domain. (Khardon 1999) learned decision-lists to guide the planner, but both these approaches were unable to reliably produce good results. DISTILL (Winner and Veloso 2003) works by synthesizing the source code of a planner that can solve each of the example problems and then code-merging operators are used to generalize the code. Another approach was developed in the CLAY framework (Srivastava and Kambhampati 1998), where automatic deductive program synthesis was used to construct domain-dependent planners. In this paper, we contribute to this line by providing an automated technique to automatically learn domain-dependent temporal planning heuristics: this is not the same as producing the code of a domain-dependent planner, but a planner equipped with our heuristic becomes a specialized planner for a certain domain.

Another related field is generalized planning, where the objective is the synthesis of plans (in forms of programs or automata) that work for a set of instances sharing some characteristics (Celorrio, Aguas, and Jonsson 2019). A recent and relevant advancement in this area is (Toyer et al. 2018) presenting "Action Schema Networks" (ASN). In this work, a generalized policy is extracted by means of deep learning from a set of problem instances on the same domain and a planning-specific transfer-learning technique is used to generalize and exploit the policy for new problems. In this paper, we are not tackling generalized planning: we maintain

(and rely on) reasoning capabilities in the planner, so instead of generating a plan that works for all the instances, we learn a heuristic to be informative in a certain domain.

Finally, some works used external control knowledge to guide planners (Doherty and Kvarnström 2001; Bacchus and Kabanza 2000). In temporal planning, (Micheli and Scala 2019) focuses on temporal control knowledge to express complex problem constraints, but no learning is present.

6 Experimental Evaluation

In this section we experimentally evaluate the merits of our approach by both comparing the planner equipped with the learned heuristic against baseline techniques and also by assessing the sensitivity of our learning approach to the different kinds of input described in section 4.

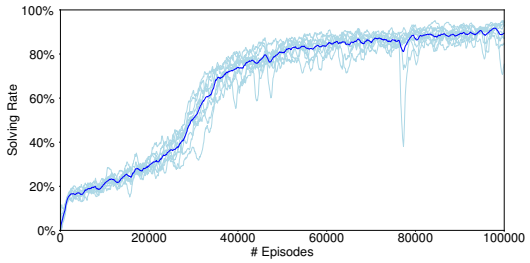
We consider two benchmark planning domains. The first one is the MAJSP domain used in (Micheli and Scala 2019) and (Valentini, Micheli, and Cimatti 2020); the domain consists of a job-shop scheduling problem in which a fleet of moving agents transport items and products between operating machines. We created 770 instances by varying the number of items and the number of treatments. Second, we created a new domain (called "kitting") in which a robot has to collect several components distributed in different locations of a warehouse in order to compose a pre-fixed kit and then deliver it to a specific location synchronizing with a human operator. We created 1092 instances of this domain by scaling the kit size (up to 5 components) and the number of kits to deliver (up to 3).

We implemented the learning part of our framework in Python3 using an adaptation of our planner, TAMER, as simulator via a dedicated API. We used the PyTorch framework for representing and training the value function neural networks. The learning process takes in input all the training instances and, using TAMER as simulator, outputs the trained value function as a neural network. In the learning algorithm we set the following parameters: $\gamma = 0.99$, the maximum size of the memory mem is $50K$, the REPLAY batch size is 1000, $\Delta_{RL} = 140$, $\epsilon_{max} = 0.5$ and $\epsilon_{min} = 0.001$. For the planning part, we extended TAMER to be able to use the trained neural network (TAMER (h_{nn})) as a heuristic (i.e. we equipped TAMER with h_{nn}) and we set $\Delta_h = 1200$ and the weight w for the planner search to 0.8.

All the experiments have been conducted on a Xeon E5-2620 2.10GHz; the experimental material is available at https://es-static.fbk.eu/people/amicheli/resources/prl20.

Performance comparison. To measure the effectiveness of our framework we performed a 10-fold cross validation: for each domain, we generated the set of ground instances and we randomly partitioned such set into 10 equal sized subsamples. In turn, we use each subsample as the testing data for the planning part, and the remaining 9 subsamples as training data for the learning part, resulting in ten runs.

We consider three competitors. TAMER (h_{add}) is the fully-symbolic planner described in (Valentini, Micheli, and Cimatti 2020) that uses no learned information, TAMER (h_{nn}) is the same planner equipped with the learned heuristic and π_{nn} is the execution of the learned policy with no backtracking ($\pi_{nn}(s) = \arg\max_\alpha V_{nn}(\alpha[s])$).



fold (size: 77)	TAMER (h_{add})		# episodes	π_{nn}		TAMER (h_{nn})	
	solved	avg plan size		solved	avg plan size	solved	avg plan size
1	52	14	50k	66	25	**73**	18
			100k	71	22	**73**	18
2	58	14	50k	70	22	**75**	17
			100k	70	19	72	17
3	58	14	50k	70	21	73	17
			100k	73	19	**75**	17
4	57	13	50k	66	21	72	17
			100k	68	20	**76**	17
5	55	15	50k	66	25	**75**	19
			100k	69	21	69	19
6	60	14	50k	66	23	76	17
			100k	69	17	**77**	17
7	54	14	50k	68	21	**76**	18
			100k	75	21	73	18
8	57	14	50k	61	23	**73**	18
			100k	**73**	20	69	18
9	57	14	50k	71	25	**74**	18
			100k	66	21	70	18
10	52	14	50k	72	21	**77**	19
			100k	65	22	54	16
all	560	14	50k	676	23	**744**	18
			100k	699	20	708	18

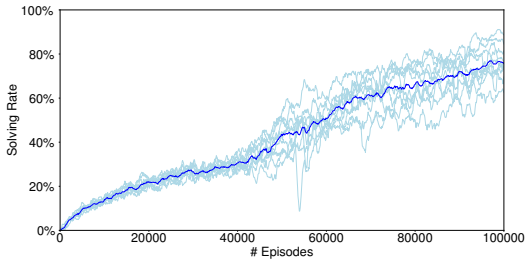
Figure 3: Results on the MAJSP domain: learning curves (above) and coverage (table). We plot the curve for each fold in light blue and the average solving rate in dark blue. For each fold and each approach, we report the total number of solved instances and the average plan length.

We imposed a 600s/20GB time/memory limit for executing all the planning approaches; instead, the learning algorithm has been executed for 100000 episodes.

Figures 3 and 4 report the learning curves and the coverage results for all the ten folds. In the learning curves, we plotted on the y-axis the solving rate of the previous 1000 episodes, that is we plot the percentage of episodes (over the previous 1000) that reached a goal state while learning. In dark blue we averaged the 10 runs (one for each fold). In the tables, for the π_{nn} and TAMER (h_{nn}) approaches, we also report the performance of a snapshot of the learned value function after 50000 and 100000 episodes to assess the learning speed. The last row of each table reports the average plan length and the total number of solved instances.

The results show how the RL algorithm is able to learn in all the ten folds, reaching a high solving rate for both domains with a small variance between the ten runs. It is interesting to note that in the MAJSP domain after 40000 episodes the curve spikes and the average solving rate immediately reaches 80%, while the learning curve for the kitting domain exhibits a steady linear growth.

The tables show how both the learning-based approaches (π_{nn} and TAMER (h_{nn})) are significantly superior to the plain TAMER(h_{add}). In fact, the two selected domains are hard for the normal reasoning techniques because they exhibit complex temporal constraints, cyclic behaviors (e.g.



fold (size: 109)	TAMER (h_{add})		# episodes	π_{nn}		TAMER (h_{nn})	
	solved	avg plan size		solved	avg plan size	solved	avg plan size
1	44	15	50k	66	18	99	21
			100k	97	21	**107**	21
2	35	15	50k	66	21	95	22
			100k	82	21	**97**	21
3	38	15	50k	55	18	83	20
			100k	97	20	**99**	20
4	45	15	50k	68	20	98	19
			100k	88	22	**100**	21
5	47	15	50k	85	19	**101**	19
			100k	88	19	**101**	19
6	38	15	50k	53	20	85	20
			100k	78	22	**108**	23
7	30	15	50k	44	18	75	19
			100k	90	24	**106**	23
8	42	15	50k	65	18	95	20
			100k	95	21	**104**	21
9	36	15	50k	44	15	70	17
			100k	89	22	**91**	20
10	40	14	50k	71	19	95	21
			100k	92	21	**102**	21
all	395	15	50k	617	19	896	20
			100k	896	21	**1015**	21

Figure 4: Results for the Kitting domain.

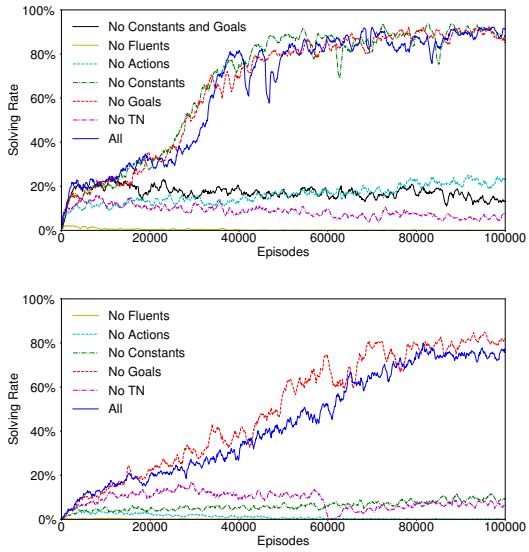


Figure 5: Learning curves of the MAJSP (above) and kitting (below) domains with different input configurations.

in kitting we need to move between the deposit location and the different shelves several times) and because they are combinatorially hard (e.g. the JSP component of MAJSP). Moreover, the TAMER(h_{nn}) approach is able to solve consistently more instances than any competitor: even when the policy execution π_{nn} comes close to the coverage of TAMER(h_{nn}), the average plan length is higher. This is due to the combination of the heuristic function (derived from the learned value function) with the path cost $g(s)$ in the search algorithm. This combination balances the systematic search performed by the planner with the information gathered during learning. We also highlight that both TAMER(*) approaches are guaranteed to eventually find a plan if it exists, while the plain execution of the learned policy (π_{nn}) can diverge or fail to find a plan.

Sensitivity Analysis. A second experiment is aimed at assessing the relevance of the different inputs we provide to the neural network V_{nn} during learning. We tried to learn from the whole set of ground instances for each domain and we disabled each of the five kinds of inputs to the network by removing the corresponding input neurons and the attached part of first layer before starting the learning algorithm.

Figure 5 shows the learning curves for both the domains. The results indicate that, for MAJSP, the encodings of fluents, actions and temporal network are needed to reach a good learning performance, while the other inputs (goals and constants) seem less impacting on this domain as their learning curve is similar to the one with all the inputs provided. This phenomenon is due to the nature of the MAJSP domain and the way it is encoded: essentially each item needs to be treated in a certain way and the goal just requires a subset of the items to be processed. However, the information on which items are relevant for the current instance is present in both the goal formulation and in the constants that are used to indicate which objects do exist (the $o_i^{\exists}$ constants). For this reason, we experimented with a network deprived of both the goals and constants inputs and we can see how it performs badly, confirming the need of all the provided input. The situation for kitting is similar, but only the network without goals is able to learn comparably with the fully-informed one. This is again due to the problem nature: the goal of kitting is to deliver a certain number of kits, but their composition (that determines the path to be taken between the shelves) is encoded using constants that, in this case, become necessary for learning a useful value function.

7 Conclusions

This paper presents the first approach to learn heuristic functions for temporal planning. Leveraging recent advancements in RL, we designed a workflow that is able to use a finite set of instances with different number of objects and synthesize a heuristic that can effectively solve problems with a bounded number of objects. The approach exploits modern neural networks and is experimentally shown to be superior to both planning and reinforcement learning alone.

There are several avenues for future research. First, the approach is limited because the instances being solved need to have a known bound on the number of objects; moreover, we currently assume that the training set is finite instead one can consider the case where an instance sampler is given ranging over a set of possibly infinite instances. A third direction is to generalize the network architecture: the current one has been experimentally derived, but having a structured way to construct the architecture given the domain and the bound would widen applicability of the technique.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artif. Intell.* 175(16-17):2075–2098.

Asai, M., and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In McIlraith, S. A., and Weinberger, K. Q., eds., *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 6094–6101. AAAI Press.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1):123 – 191.

Bonisoli, A.; Gerevini, A.; Saetti, A.; and Serina, I. 2015. Effective plan retrieval in case-based planning for metric-temporal problems. *Journal of Experimental and Theoretical Artificial Intelligence* 27:1–45.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-ff: Improving AI planning with automatically learned macro-operators. *J. Artif. Intell. Res.* 24:581–621.

Celorrio, S. J.; Aguas, J. S.; and Jonsson, A. 2019. A review of generalized planning. *Knowledge Eng. Review* 34:e5.

Choudhury, S.; Bhardwaj, M.; Arora, S.; Kapoor, A.; Ranade, G.; Scherer, S. A.; and Dey, D. 2018. Data-driven planning via imitation learning. *I. J. Robotics Res.* 37(13-14).

Coles, A., and Smith, A. 2007. Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res.* 28:119–156.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *ICAPS 2010*.

de la Rosa, T.; Olaya, A. G.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In *Case-Based Reasoning Research and Development, 7th International Conference on Case-Based Reasoning, ICCBR 2007, Belfast, Northern Ireland, UK, August 13-16, 2007, Proceedings*, 137–148.

Doherty, P., and Kvarnström, J. 2001. Talplanner: A temporal logic-based planner. *AI Magazine* 22(3):95–102.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2012. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments - Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural network heuristics for classical planning: A study of hyperparameter space. ECAI.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*.

Gigante, N.; Micheli, A.; Montanari, A.; and Scala, E. 2020. Decidability and complexity of action-based temporal planning over dense time. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 9859–9866. AAAI Press.

Khardon, R. 1999. Learning action strategies for planning domains. *Artif. Intell.* 113(1-2):125–148.

Micheli, A., and Scala, E. 2019. Temporal planning with temporal metric trajectory constraints. In *AAAI 2019*, 7675–7682.

Rankooh, M. F., and Ghassem-Sani, G. 2015. Itsat: an efficient sat-based temporal planner. *Journal of Artificial Intelligence Research*.

Smith, D.; Frank, J.; and Cushing, W. 2008. The anml language. In *KEPS 2008*.

Spalazzi, L. 2001. A survey on case-based planning. *Artif. Intell. Rev.* 16(1):3–36.

Spector, L. 1994. Genetic programming and AI planning systems. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, 1329–1334.

Srivastava, B., and Kambhampati, S. 1998. Synthesizing customized planners from specifications. *J. Artif. Intell. Res.* 8:93–128.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 6294–6301.

Valentini, A.; Micheli, A.; and Cimatti, A. 2020. Temporal planning with intermediate conditions and effects. In *AAAI 2020*.

Virseda, J.; Borrajo, D.; and Alcazar, V. 2013. Learning heuristic functions for cost-based planning. In *Proceedings of the 4th Workshop on Planning and Learning*, 6–13.

Winner, E., and Veloso, M. M. 2003. DISTILL: learning domain-specific planners by example. In *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, 800–807.

Yoon, S. W.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *J. Mach. Learn. Res.* 9:683–718.

A Framework for Reinforcement Learning and Planning: Extended Abstract*

Thomas M. Moerland,[1,2] **Joost Broekens,**[2] **Catholijn M. Jonker**[1,2]
[1] Interactive Intelligence, Delft University of Technology, The Netherlands
[2] Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands

Abstract

Two successful approaches to Markov Decision Process optimization are planning and reinforcement learning. Both research communities operate large separate. This framework attempts to bridge both fields, by disentangling their common algorithmic space, showing that both fields face exactly the same algorithmic decisions. The full paper is available from https://arxiv.org/pdf/2006.15009.pdf.

Sequential decision making, commonly formalized as Markov Decision Process (MDP) optimization, is a key challenge in artificial intelligence research. The two prime research directions in this field are *reinforcement learning* (Sutton and Barto 2018), a subfield of machine learning, and *planning* (also known as *search*), of which the discrete and continuous variants have been studied in the fields of artificial intelligence (Russell and Norvig 2016) and control (Bertsekas 1995), respectively. Planning and learning approaches differ with respect to a key assumption: is the dynamics model of the environment known (planning) or unknown (reinforcement learning).

Departing from this distinctive assumption, both research fields have largely developed their own methodology, in relatively separated communities. There has been cross-breeding as well, better known as 'model-based reinforcement learning' (recently surveyed by Moerland, Broekens, and Jonker (2020)). While the combination of planning and learning has shown great empirical success (Silver et al. 2017), literature still lacks a fundamental view on the relation between both fields, and how their approaches overlap and differ.

Therefore, this paper* introduces the Framework for Reinforcement learning and Planning (FRAP), which identifies the essential algorithmic decisions that any planning or RL algorithm has to make. It consists of six main dimensions, which we will shortly discuss in more detail. However, the main message of the framework is that any RL or planning algorithm, from Q-learning (Watkins and Dayan 1992) to A* (Hart, Nilsson, and Raphael 1968), will have to make a decision on each of these dimensions. Therefore, planning and learning are not only related, but really two sides of the same coin. We illustrate this point in the full paper*, by formally comparing a variety of planning and RL papers along the dimensions of our framework.

*Full paper available at: https://arxiv.org/pdf/2006.15009.pdf

Framework

We will here shortly introduce the structure of FRAP. It consists of six main dimensions, of which some have multiple sub-considerations, which are summarized in Table 1. FRAP centers around the concept of *trials* and *back-ups*. A trial is a single call to the environment, where we impute a state-action pair and get back a next state (distribution) and associated reward (distribution). Trials are the fundamental way in which we get information about the environment. After one or more trials, we want to back-up the acquired information to better decide where we want to make the next trial. We may disentangle this process into six key questions:

1. *Where to put our computational effort?*
   We first determine for which states we seek a solution at all. As a crucial distinction, we may either consider all states (as used by Dynamic Programming approaches), or only the reachable states (which we can track by only sampling from a start state distribution).
2. *Where to make the next trial?*
   We then determine where to make the next trial. There are several relevant considerations, like which set of state-action pairs are candidate for selection in the current iteration (e.g., all actions at the current state, or all state-actions at the frontier), and how to add exploration (since greedy selection leads to suboptimal behaviour).
3. *How to estimate the cumulative return?*
   After we make the trial, we need an estimate of the remaining cumulative reward after the trial. We can either sample and/or bootstrap, which we both need to decide on.
4. *How to back-up?*
   We then want to back-up this new information (obtained from the trial). We need to decide on the back-up policy, and on how to deal with the expectations over the actions and dynamics in the one-step Bellman equation. As an example, both a well-known planning algorithm like MCTS

Table 1: Overview of dimensions in the Framework for Reinforcement learning and Planning (FRAP). For any planning or reinforcement learning algorithm, we should be able to identify the decision on each of the dimensions. The subconsiderations and possible options are shown in the right columns. IM = Intrinsic Motivation.

Dimension	**Consideration**	**Choices**
1. Comp. effort	- State set	All $\leftrightarrow$ reachable $\leftrightarrow$ relevant
2. Trial selection	- Candidate set	Step-wise $\leftrightarrow$ frontier
	- Exploration	Random $\leftrightarrow$ Value-based $\leftrightarrow$ State-based -For value: mean value, uncertainty, priors -For state: ordered, priors (shaping), novelty, knowledge IM, competence IM
	- Phases	One-phase $\leftrightarrow$ two-phase
	- Reverse trials	Yes $\leftrightarrow$ No
3. Return estim.	- Sample depth	$1 \leftrightarrow n \leftrightarrow \infty$
	- Bootstrap func.	Learned $\leftrightarrow$ heuristic $\leftrightarrow$ none
4. Back-up	- Back-up policy	On-policy $\leftrightarrow$ off-policy
	- Policy expec.	Expected $\leftrightarrow$ sample
	- Dynamics expec.	Expected $\leftrightarrow$ sample
5. Representation	- Function type	Value $\leftrightarrow$ policy $\leftrightarrow$ both (actor-critic) - For all: generalized $\leftrightarrow$ not generalized
	- Function class	Tabular $\leftrightarrow$ function approximation - For tabular: local $\leftrightarrow$ global
6. Update	- Loss	- For value: e.g., squared -For policy: e.g., (det.) policy gradient $\leftrightarrow$ value gradient $\leftrightarrow$ cross-entropy, etc.
	- Update	Gradient-based $\leftrightarrow$ gradient-free - For gradient-based, special cases: replace & average update

(Kocsis and Szepesvári 2006) and a well-known RL algorithm like SARSA (Rummery and Niranjan 1994) make the same algorithmic choice here (an on-policy, sample action, sample dynamics back-up).

5. *How to represent the solution?* We also want to be able to store the new information. Here, planning and reinforcement learning have emphasized different approaches, since planning methods mostly focus on tabular/atomic representations (like nodes), while reinforcement learning approaches have emphasized approximate (learned) representations of the solution.
6. *How to update the solution?* Finally, we need to update our solution (from 5) based on the back-up estimate (from 4). We can distinguish gradient-based and gradient-free updates. FRAP also shows how common planning updates can be cast into this categorization.

The framework shows that planning and learning essentially do the same thing. As an illustration, note that a MCTS of 500 traces is conceptually not too different from 500 episodes of a model-free Q-learning agent in the same environment. In both cases, we repeatedly move forward in the environment to acquire new information, make back-ups to store this information, with the goal to make better informed decisions in the next trace/episode. The model-free RL agent is restricted in the order in which it can visit states, but otherwise, the methodology of exploration, back-ups, representation and updates is the same.

The main paper includes a large table comparing a variety of planning, model-free RL and model-based RL papers along the dimensions of our framework, which illustrates the validity of FRAP. In short, FRAP provides a common language to categorize algorithms in both fields, hopefully serving as a bridge between both. We hope it also inspires new research, for example by identifying novel possible combinations of planning and learning, or stimulating the design of a new algorithm in one field based on inspiration from the other.

References

Bertsekas, D. P. 1995. *Dynamic programming and optimal control*, volume 1.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, volume 6, 282–293. Springer.

Moerland, T. M.; Broekens, J.; and Jonker, C. M. 2020. Model-based Reinforcement Learning: A Survey. *arXiv preprint arXiv:2006.16712*.

Rummery, G. A., and Niranjan, M. 1994. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England.

Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Silver, D.; van Hasselt, H.; Hessel, M.; Schaul, T.; Guez, A.; Harley, T.; Dulac-Arnold, G.; Reichert, D.; Rabinowitz, N.; Barreto, A.; et al. 2017. The predictron: End-to-end learning and planning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 3191–3199. JMLR. org.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.

Think Neither Too Fast Nor Too Slow: The Computational Trade-off Between Planning And Reinforcement Learning

Thomas M. Moerland,[1,2*] **Anna Deichler,**[1,4*] **Simone Baldi,**[3,4] **Joost Broekens,**[2] **Catholijn M. Jonker**[1,2]
[1] Interactive Intelligence, Delft University of Technology, The Netherlands
[2] Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
[3] School of Cyberscience and Engineering, Southeast University, China
[4] Delft Center for Systems and Control, Delft University of Technology, The Netherlands

Abstract

Planning and reinforcement learning are two key approaches to sequential decision making. Multi-step approximate real-time dynamic programming, a recently successful algorithm class of which AlphaZero (Silver et al. 2018) is an example, combines both by nesting planning within a learning loop. However, the combination of planning and learning introduces a new question: how should we balance time spend on planning, learning and acting? The importance of this trade-off has not been explicitly studied before. We show that it is actually of key importance, with computational results indicating that we should neither plan too long nor too short. Conceptually, we identify a new spectrum of planning-learning algorithms which ranges from exhaustive search (long planning) to model-free RL (no planning), with optimal performance achieved midway.

1 Introduction

Sequential decision-making, commonly formalized as Markov Decision Process (MDP) optimization, is a key challenge in artificial intelligence (AI) and machine learning research. Important solution approaches include planning (or search) (Russell and Norvig 2016) and reinforcement learning (Sutton and Barto 2018). Recently, a class of algorithms, known as multi-step approximate real-time dynamic programming (MSA-RTDP), combines both fields. MSA-RTDP iterates planning, which uses a learned value/policy function, and learning, which uses output from the planning procedure. A successful example in this class is the AlphaZero algorithm, which achieved super-human performance in the game of Go, Chess, and Shogi (Silver et al. 2017; 2018).

This iterated planning and learning procedure introduces a crucial new question: how long should we plan at a given state? We hypothesize that this is a crucial trade-off for planning-learning integrations: when we plan too extensively, we make too little progress in the domain and have less training targets for learning, while when we plan too briefly, our local decisions and training targets are likely to be less optimal. This trade-off was never present in online planning, where the budget per real step is typically as high as the application permits (in the order of milliseconds for a video game, or in the order of seconds to minutes for a game of Chess (Campbell, Hoane Jr, and Hsu 2002)). It was neither present in model-free reinforcement learning (RL), since those approaches do not have access to a dynamics model and can therefore not plan. Model-based RL, where we use observed data to approximate the dynamics model, has mostly focused on dealing with enhancing data efficiency and dealing with uncertainty in the learned models (Sutton 1991; Chua et al. 2018). Instead, we focus on the situation with a known, perfect model without uncertainty, to fully investigate the trade-off between planning and learning once a good model is available.

We therefore study the AlphaZero algorithm on several known tasks, where we fix the overall computational budget, but vary the planning budget per real step and associated training iteration. Our results show that, for a fixed overall time budget, approaches with an intermediate planning budget per time-step achieve the highest final performance. First, this is an important empirical insight for model-based reinforcement learning and MSA-RTDP algorithms. Moreover, the fundamental mutual benefit of planning and learning, which outperforms their isolated application, may also provide an argument for the existence of fast prediction (System 1) and explicit planning (System 2) in human decision making. This theory, better known as dual process theory (Evans 1984), was more recently popularized as 'thinking fast and slow' (Kahneman 2011). A short summary of our results could be: 'think too fast nor too slow'.

The remainder of this paper is organized as follows. Section 2 provides essential background on Markov Decision Process optimization, while Section 3 introduces the algorithm class of interest, multi-step approximate real-time dynamic programming. Section 4 and 5 detail methodology and results, respectively. The final sections cover Related work (Sec. 6), Discussion (Sec. 7) and Conclusion (Sec. 8). Code to replicate experiments is available from https://github.com/ratponto/tree-rl-adaptive.

[*] Authors contributed equally.

2 Preliminaries

We study the *Markov Decision Process* (MDP) (Puterman 2014) optimization problem. An MDP is defined by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $T : \mathcal{S} \times \mathcal{A} \to p(\mathcal{S})$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, an initial state distribution $p(s_0)$ and a discount parameter $\gamma \in [0, 1]$.

We can interact with the environment through a policy $\pi : \mathcal{S} \to p(\mathcal{A})$. After specifying an action a_t in state s_t, the environment returns a next state $s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)$ and associated reward $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$. We are interested in finding the policy that gives the highest cumulative pay-off. Define the state-action value as:

$$Q(s, a) \doteq \mathbb{E}_{\pi,\mathcal{T}}\left[\sum_{k=0}^{K} \gamma^k r_{t+k} \middle| s_t = s, a_t = a\right] \quad (1)$$

and $V(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q(s, a)]$. There is only one optimal value function $Q^\star(s, a)$ (Sutton and Barto 2018), and our goal is to find an optimal policy $\pi^\star$ that achieves the optimal value:

$$\pi^\star = \arg\max_\pi Q(s, a). \quad (2)$$

The possible approaches to this problem crucially rely on our type of access to the environment dynamics $\mathcal{T}$ and reward function $\mathcal{R}$. In model-free reinforcement learning, the environment cannot be reverted, and we therefore have to sample forward from the state that we reach. This property, also referred to as an 'unknown model', is also part of the real world. In contrast, in planning and model-based RL, we are either given or have learned a reversible model, better known as a 'known model', which we can query for a next state and reward for any state-action pair that we impute.

A classic approach in the latter case (known model) is Dynamic Programming (DP) (Bellman 1966). For example, in Q-value iteration we sweep through a state-action value table, where at each location we update $Q(s, a)$ according to:

$$Q(s, a) \leftarrow \mathbb{E}_{s'\sim\mathcal{T}(\cdot|s,a)}\left[\mathcal{R}(s, a, s') + \gamma \max_{a\in\mathcal{A}} Q(s, a)\right] \quad (3)$$

Dynamic programming is guaranteed to converge to the optimal policy. However, due to the curse of dimensionality, it can not be applied in high-dimensional problems. In the next section we introduce a recently popularized extension of DP.

3 Multi-step Approximate Real-Time Dynamic Programming

Multi-step approximate real-time dynamic programming (Efroni, Ghavamzadeh, and Mannor 2019) has recently shown impressive empirical results, for example beating humans and achieving state-of-the-art performance in the game of Go (Silver et al. 2017), Chess and Shogi (Silver et al. 2018). MSA-RTDP is based on Dynamic Programming concepts, but adds three additional concepts:

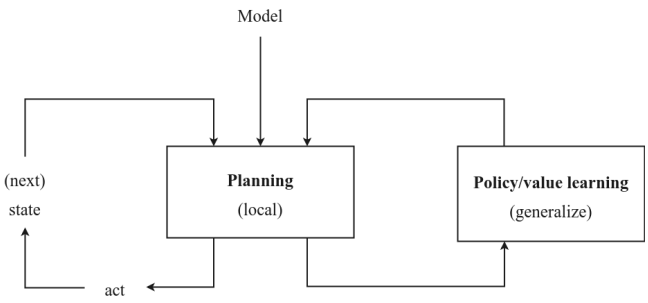


Figure 1: Multi-step Real-time Dynamic Programming. The three key procedures are 1) Planning, 2) Learning, and 3) Real steps (acting).

- 'Real time' (Barto, Bradtke, and Singh 1995) implies that we act on traces through the environment that start from some initial state $s_0 \sim p(s_0)$. This property is assumed by most RL and planning algorithms. Compared to the DP sweeps, it avoids work on states that we will never reach.
- 'Approximate' implies that we will use function approximation to store a global parametrized solution, in the form of a value $V_\theta(s)$/$Q_\theta(s, a)$ and/or policy function $\pi_\theta(a|s)$, where $\theta \in \Theta$ denote the parameters of the approximation. Compared to a tabular representation, approximate representations can deal with high-dimensional state spaces and benefit from generalization between similar states, although they do make approximation errors. Approximate solutions are especially popular in RL literature.
- 'Multi step' implies that for every Dynamic Programming back-up, we are allowed to make a multi-step lookahead, i.e., we can *plan*.

The resulting multi-step approximate RTDP algorithm class has three key components, which are visualized in Figure 1:

1. **Plan**: At every state s_t in the trace, we get to expand some computational budget `B` of forward planning, which could for example be a depth-d full-breadth search (Russell and Norvig 2016), or a more complicated planning procedure like Monte Carlo Tree Search (Browne et al. 2012). The planning procedure can use learned value/policy functions to aid planning, for example through *bootstrapping* (Sutton and Barto 2018).
2. **Learn**: After planning, we use the output of planning (our improved knowledge about the optimal value and policy at s_t) to train our global value/policy approximation.
3. **Real step**: We finally use the planning output to decide which action a_t we will commit to, and make a 'real step', transitioning to a sampled next state $s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)$. The next iteration of planning continues from s_{t+1}.

MSA-RTDP has two special cases that depend on the computation planning budget `B` per real step. One the one extreme, $\texttt{B} \to \infty$, we completely enumerate all possible future traces, better known as *exhaustive search* (Russell and

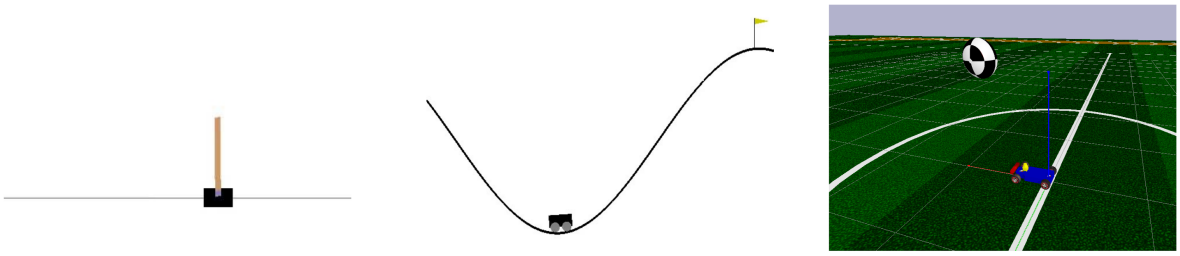


Figure 2: Image stills from the studied tasks. Left: CartPole, where we attempt to balance the pole. Middle: MountainCar, where we attempt to reach the top-left flag by swinging back and forth. Right: RaceCar, where we need to control a car to reach a goal, indicated by a ball.

Norvig 2016). On the other extreme, $\texttt{B} = 0$, we do not plan at all, but directly make a real step based on the global approximations, better known as *model-free reinforcement learning* (Sutton and Barto 2018).

Anthony, Tian, and Barber (2017) already related this approach to cognitive psychology research, in particular dual process theory (Evans 1984; Kahneman 2011). The global value/policy approximation, which makes fast predictions about the value of actions, can be considered a System 1 ('Thinking Fast'), while explicit forward planning to improve over these fast approximations seems related to System 2 ('Thinking slow').

4 Methods

For this paper, we will follow the AlphaGo Zero (Silver et al. 2017) variant of MSA-RTDP. AlphaGo Zero uses a variant of MCTS (Browne et al. 2012) for planning, and deep neural networks for leaning of a policy $\pi_\theta(a|s)$ and value $V_\theta(s)$ approximation. A key aspect of iterated planning-learning is their mutual influence, where planning improves the learned function, and the learned function directs new planning iterations. We will detail both these integrations, starting with training target construction based on planning output.

To train the policy network, we normalize the action visitation counts $n(s,a)$ at the tree root state s to a probability distribution, and train on a cross-entropy loss:

$$\mathrm{L}_\pi(\theta) = \sum_a \frac{n(s,a)}{n(s)} \log \pi_\theta(a|s). \tag{4}$$

For value network training, we use a target based on the reweighted value estimates at the root of the MCTS,

$$\hat{V}(s) = \sum_a \frac{n(s,a)}{n(s)} \bar{Q}(s,a), \tag{5}$$

where $\bar{Q}(s,a)$ denotes the mean pay-off of all traces through (s,a), and train on a squared error loss,

$$\mathrm{L}_V(\theta) = \left(V_\theta(s) - \hat{V}(s)\right)^2. \tag{6}$$

This is a slight variation of the original AlphaZero implementation, based on recent results of Efroni et al. (2018). The above equations define the planning to learning connection in Fig. 1.

For the reverse connection, influencing planning based on the learned functions, we i) replace the MCTS rollout by a bootstrap estimate from the value network, and ii) modify the MCTS selects step to

$$\arg\max_a \left[\bar{Q}(s,a) + c \cdot \pi_\theta(a|s) \cdot \sqrt{\frac{n(s,a)}{1+n(s)}}\right], \tag{7}$$

where $c \in \mathbb{R}$ is a constant that scales exploration pressure.

We vary the planning budget per timestep through adjustment of the number of traces per MCTS iteration, denoted by n_{MCTS}, while keeping the overall computational budget (in the form of wall clock time) fixed. We experiment with two well-known control tasks, CartPole and MountainCar, available from the OpenAI Gym (Brockman et al. 2016), and with the RaceCar task, available in the PyBullet package (Coumans and Bai 2016). For MountainCar, we use a reward function variant with $r = -0.005$ on every step, and $r = +1$ when the Car reaches the top of the hill. Visualizations of the tasks are shown in Figure 2.

The total computational budget (planning, training and acting) was fixed in advance on every environment: 500 seconds for CartPole, 150 minutes for MountainCar, and 270 minutes for RaceCar. These budgets were predetermined to allow for convergence on each domain. Therefore, long planning per timestep (higher n_{MCTS}) also implies less real steps and less new training targets over the entire training period.

Hyperparameters The effect of search budget may also interact with the setting of other hyperparameters. We chose the following approach. We quickly search for a general hyperparameter configuration that shows increasing learning curves on all domains. Crucially, the search budget was varied in this quick search, but we were unaware of its actual values, to not bias the other hyperparameter settings towards good performance on a particular search budget. We will touch upon alternative approaches in the Discussion.

We here report the fixed values for the other hyperparameters. For neural network training, we used batches of size 16 with a replay buffer of size 5e3 and learning rate of 1e-3 on all domains, optimized with ADAM optimizer (Kingma and

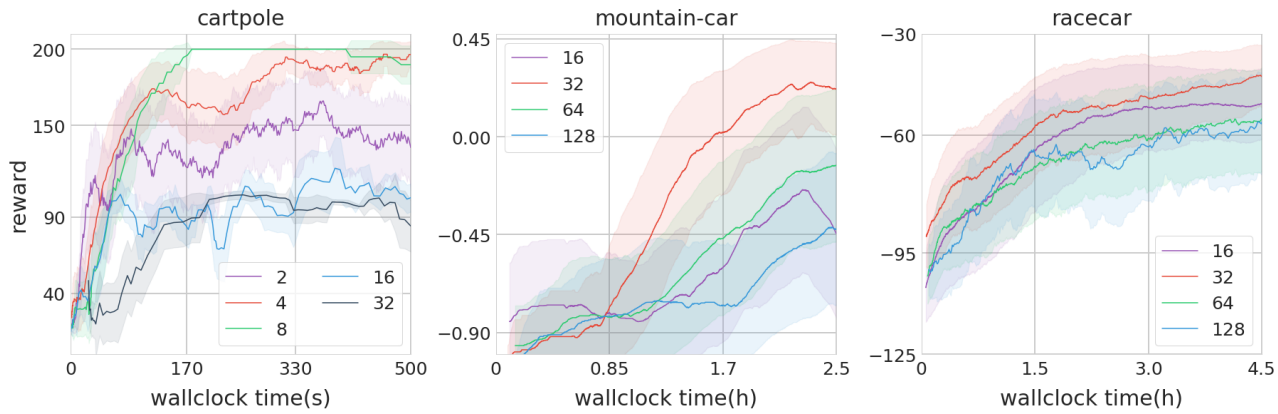


Figure 3: Learning curves on CartPole, MountainCar and RaceCar environments. The colour legend per plot displays the MCTS trace budget before every real step (n_{MCTS}). There is no clear normalization criterion for the return scales on each domain, so we report their absolute values. We see that AlphaGo Zero learns on all tasks, with best performance on CartPole, MountainCar and RaceCar achieved for budgets of, respectively, 8, 32 and 32 traces per timestep.

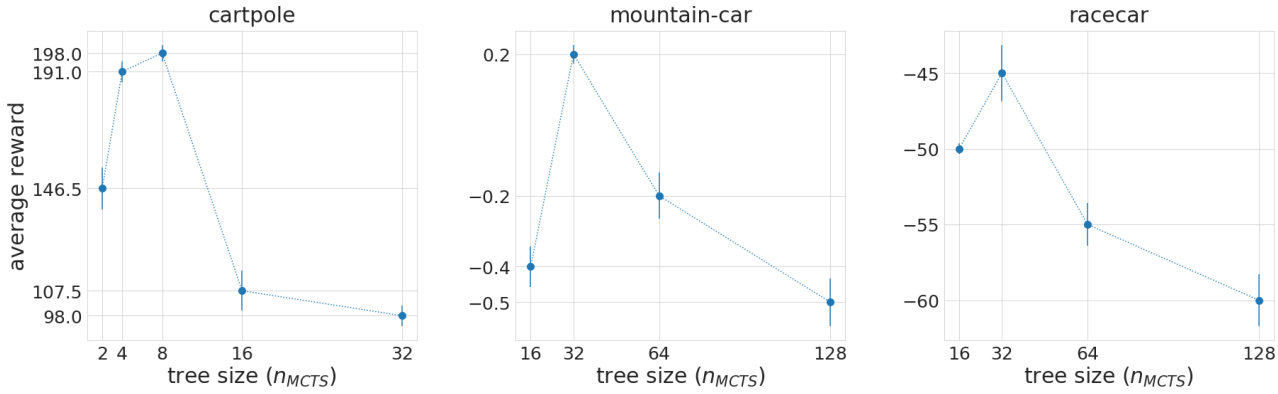


Figure 4: Trade-off between planning and learning. The horizontal axis shows the computational budget per MCTS search in the form of the total number of traces. The vertical axis shows the cumulative reward achieved by the specific set-up. Data based on last 15% of the learning curves in Fig. 3. Note that the total computation time for every repetition was fixed, i.e., higher planning budget per timestep will yield less real steps and less targets for training the neural networks. We observe a clear trade-off on all domains, with optimal results achieved for intermediate search budgets.

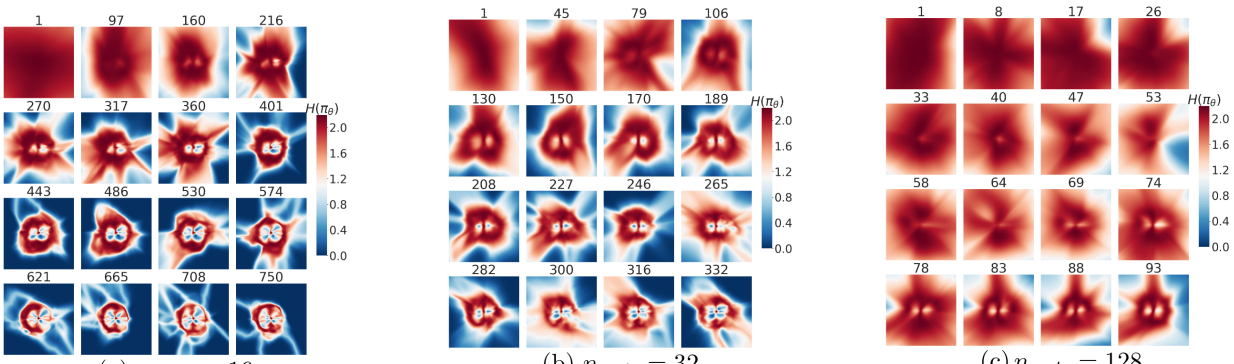
Ba 2014). Policy and value network shared their hidden layers, with 256 hidden nodes per layer. Since the reward scales between the task varied greatly, the c parameter (Eq. 7) did require adjustment per domain: for CartPole we decayed it from 0.8 to 0.05 in 500 steps, for MountainCar from 5 to 0.5 in 5000 steps, and for RaceCar from 1.0 to 0.05 in 1500 steps. All results are averaged over 3 repetitions.

5 Results

Figure 3 shows learning curves for the three environments. We see that the AlphaZero algorithm manages to learn all three tasks. The largest variation in performance is seen on the CartPole task. Clearly, the most stable performance for CartPole uses $n_{\text{MCTS}} = 8$. Compared to CartPole, MountainCar has a sparser reward. We therefore require longer total budget and more traces per timestep to achieve best performance, which is attained with $n_{\text{MCTS}} = 32$. Finally, RaceCar has a larger action space than both other domains, wich requires longer training, and generally more traces per timestep. The best performance is achieved for $n_{\text{MCTS}} = 32$ traces.

The learning curves indicate that optimal performance is achieved for an intermediate search budget. To better illustrate this observation, we aggregate the average pay-offs from the last 15% of total time for every planning budget in each environment. These results are visualized in Figure 4. The horizontal axis now displays search budget, while the vertical axis displays mean pay-off at the end of training. For all three environments, we observe clear optimal performance for an intermediate search budget per real step.

To further investigate what happens during training, we visualize the output of the policy network on RaceCar for different search budgets in Figure 5. The right, middle and left progression refer to n_{MCTS} settings of 16, 32 and 128,



(a) $n_{mcts} = 16$ (b) $n_{mcts} = 32$ (c) $n_{mcts} = 128$

Figure 5: Training progression of policy network on RaceCar, for a) $n = 16$ trace budget per MCTS iteration, b) $n = 32$ trace budget, and c) $n = 128$ trace budget. Each plot (a-c) visualizes a progression over training, where the number above the subplot indicates the episode number. A subplot within each plot visualizes the two-dimensional state space (x-y location of the ball in first person view), where each state is colour coded according to the entropy of the policy network at that state. High entropy (red colour) implies an uncertain policy, while low entropy (blue) implies a converged policy network. We see that the right progression ($n_{\text{MCTS}} = 128$) qualitatively seems to slow, as there are too little training targets. The left progression ($n_{\text{MCTS}} = 16$) seems to converge fast, but Fig. 4 shows that convergence is premature, as the achieved return is worse than the middle progression ($n_{\text{MCTS}} = 128$).

respectively. Each subplot shows the two-dimensional RaceCar state space, which describes the (x,y)-location of the ball in first person view. Each state in this state space is coloured according to the entropy of the policy network. Red colour implies high entropy and therefore an uncertain policy, while blue colour implies low entropy and a near converged policy. The number above each subplots indicates the episode number.

First of all, we may note that the entropy of the policy is high in the entire state space at the beginning of all three search budgets, which is to be expected. Second, we can clearly observe a difference in the number of completed episodes. Looking at the bottom-right subplot of the left ($n_{\text{MCTS}} = 16$), middle ($n_{\text{MCTS}} = 32$) and right ($n_{\text{MCTS}} = 128$) plot, we observe that we completed 750, 332 and 93 full episodes for the search budgets of 16, 32 and 128 traces per real step, respectively. Of course, a higher search budget implies that we complete less episodes.

More interestingly, we can qualitatively compare the convergence of the policy networks in all three scenarios. When we compare the high search budget (right) with the intermediate one (middle), we see that the high search budget shows a similar progression, but it progresses slower. For example, the policy network at episode 93 for $n_{\text{MCTS}} = 128$ shows similarity with the situation after episode 170 for $n_{\text{MCTS}} = 32$, with near convergence (blue) at the border of the state space, and demarcation of early convergence areas (white) in the center of state space. Although we did require less episodes to reach that situation for $n_{\text{MCTS}} = 128$, it did take more computation due to the relatively high planning effort per real step. Therefore, the high planning budget cannot benefit enough from generalization of information. The reverse situation is visible when we compare the left plot ($n_{\text{MCTS}} = 16$) with the middle plot ($n_{\text{MCTS}} = 32$). In the left plot, the policy network seems to converge faster, with a very certain policy (blue) in most of the state space at the end of the total time budget. However, if we look at the performance in Fig. 4, the convergence was actually premature, as we probably trained on planning targets that were too unstable. We will further interpret these observations in the discussion.

6 Related Work

AlphaGo Zero (Silver et al. 2017) and Alpha Zero (Silver et al. 2018), as used in this work as well, are examples of multi-step approximate real-time dynamic programming. AlphaGo Zero treats the trade-off between planning and learning as a fixed hyperparameter, where they use 1600 MCTS traces per real step in the game of Go, and 800 MCTS traces per real step for both Chess and Shogi. A very similar algorithm is Expert Iteration (ExIt) (Anthony, Tian, and Barber 2017), which shows state-of-the-art performance in the game Hex. The authors do not report the MCTS budget per search used during training.

The earliest idea of iterated search and learning seems to date back to Samuel's checkers programme (Samuel 1967). In later work, Carmel and Markovitch (1999) explicitly studies *lookahead-based exploration*. The authors do mention that 'it is rational for the agent to invest in computation in order to save interaction', but do not further investigate this trade-off. Chang et al. (2015) made a step towards multi-step approximate real-time dynamic programming with Locally Optimal Learning to Search (LOLS). LOLS iterates i) Monte Carlo search, which leverages the policy, and ii) policy training, which is based on the estimated values during planning. Other algorithms that update a global value approximation based on nested search are Sheppard (2002) and Veness et al. (2009).

A theoretical study of multi-step greedy real-time dynamic programming was recently provided by Efroni, Ghavamzadeh, and Mannor (2019). One of their results shows that the *sample* complexity of multi-step greedy RTDP scales as $\Omega(1/d)$, where d denotes the depth of the lookahead, while the *computational* complexity scales as $\Omega(d)$. We directly see the trade-off appearing here, as deeper planning decreases the required number of real steps at the expense of increased computation. Our work provides an empirical investigation of the effect of this trade-off. Our results also seem to indicate that the optimal, intermediate planning budget also correlates with the dimensionality of the problem, where more complex problems require a higher budget.

Our empirical results are also partly visible in the concurrent work of Wang et al. (2019). These authors benchmark several model-based RL algorithms. They do not focus on iterated search and RL algorithms, like multi-step approximate real-time dynamic programming, but do include results of standard RL methods that train on learned dynamics models. Their results show a similar trade-off. However, their results could also be caused by the uncertainty in a learned model, which makes planning far ahead less reliable. In contrast, our work shows a more fundamental trade-off exist, even in the case of a converged/perfect model.

As mentioned before, from a psychological perspective, our work can be related to *dual process theory*. Developed in the 70's and 80's by Evans (1984), it describes the presence of a System 1 and System 2 in human cognition. System 1 and 2 have more recently been popularized as 'Thinking Fast and Slow' (Kahneman 2011; 2003), respectively. System 1 includes fast, reactive, automatic behaviour, much like a neural network prediction, while System 2 includes slow, calculating, effortful decision-making, which bears similarity to local planning. This paper identifies the mutual benefit of both for optimal sequential decision making, and may as such also provide a computational motivation for the presence of both systems in humans.

7 Discussion

The computational experiments in this work clearly show a trade-off between planning, learning and acting. We identify planning budget per timestep as the major factor of importance: with a higher budget per timestep, we generate less training targets (and therefore spend less time on training) and make less real steps (complete less full episodes).

Figure 6 conceptually illustrates the observations from this paper. On the left of this plot, we find model-free RL, where the planning budget per timestep $\texttt{B} = 0$, and we only make real steps. Although model-free RL has shown impressive results (Mnih et al. 2015), it is known to be notoriously unstable, especially in combination with function approximation (Sutton and Barto 2018). On the right of this plot we find exhaustive search, where the computational budget per timestep $\texttt{B} \rightarrow \infty$, and we try to completely enumerate all futures from the root before choosing an action. Exhaustive search has high computational complexity that scales exponentially in the depth of the problem, and is therefore generally not a feasible approach. The problem is that it never generalizes information between states it encounters (no learning), and therefore repeats much work.

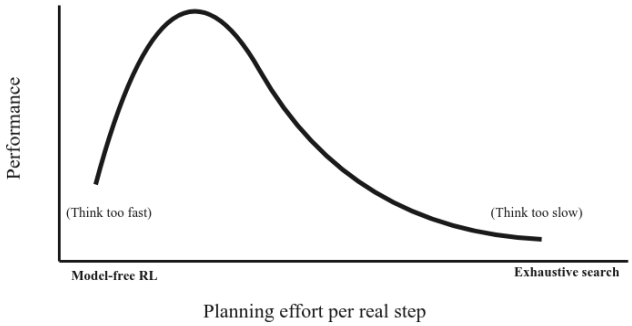


Figure 6: Conceptual illustration of the trade-off between planning and learning. The horizontal axis shows the computational budget of planning before every real step. On the left extreme we find model-free RL, which samples only a single transition before every step. On the far right, we find exhaustive search, which completely enumerates the search tree before executing a step. The curve illustrates the experimental results, which show a trade-off.

Given the above observations, the shape of Figure 6 may come to no surprise, as it appears to keep the best of both worlds. On the one hand, we use local planning to i) create better training targets for our global value/policy approximation, and ii) correct for local errors in these approximations by looking ahead to more clearly discriminable states. On the other hand, learning adds to pure planning the ability to generalize and store global solutions in memory, which avoids repeating much work, as for example present in exhaustive search.

As mentioned in Sec. 4, the effect of planning budget per timestep may interact with the value of other hyperparameters. For this work we chose to quickly search for a general hyperparameter setting on all domains, while being agnostic to the search budget in that phase. There could be two alternative approaches. First, we could separately optimize all other hyperparameters for every search budget on every domain. This would squeeze out the optimal performance, but is very computationally demanding. Second, we could specify an interval for every hyperparameter with reasonable values, and test on a set of random samples from these ranges, which would test robustness to hyperparameter variation. These could be interesting extensions with slightly different messages. Nevertheless, our approach is also unbiased, shows consistent results over tasks, and complies with empirical search budget decisions in other papers, for example in AlphaGo Zero (Silver et al. 2017) (which used 1600 MCTS traces per real step, not 1 or 10 million).

Neuroscience has suggested that both systems in dual process theory compete for control over the decision (Daw, Niv, and Dayan 2005). Our work provides computational motivation that both systems are complementary, and actually both necessary for optimal decision making. This may also provide an evolutionary motivation for their existence.

A clear direction of future work would be to adaptively adjust the planning budget per timestep in a data-driven way. Cognitive science has for long investigated how humans decide on planning duration, aiming to find a 'satisficing' (a portmanteau of satisfy and suffice) solution (Schwartz et al. 2002). Computational models of such data-dependent trade-offs, possibly based on the remaining uncertainty in the plan, may further improve performance of planning-learning intergrations.

8 Conclusion

This paper investigated the computational trade-off between planning and learning. Our results indicate that high performance requires both local planning and global function approximation, and that the planning budget per real time-step should neither be too high nor too low. This is an important insight for the empirical application of model-based RL algorithms, but may also provide a computational motivation for the existence of a dual system in human cognition. Moreover, it opens up towards future research on this trade-off, for example identifying whether the budget per time-step should be a context-dependent function of the observed data.

References

Anthony, T.; Tian, Z.; and Barber, D. 2017. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, 5360–5370.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial intelligence* 72(1-2):81–138.

Bellman, R. 1966. Dynamic programming. *Science* 153(3731):34–37.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Campbell, M.; Hoane Jr, A. J.; and Hsu, F.-h. 2002. Deep blue. *Artificial intelligence* 134(1-2):57–83.

Carmel, D., and Markovitch, S. 1999. Exploration strategies for model-based learning in multi-agent systems: Exploration strategies. *Autonomous Agents and Multi-agent systems* 2(2):141–172.

Chang, K.-W.; Krishnamurthy, A.; Agarwal, A.; Daume, H.; and Langford, J. 2015. Learning to Search Better than Your Teacher. In *International Conference on Machine Learning*, 2058–2066.

Chua, K.; Calandra, R.; McAllister, R.; and Levine, S. 2018. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, 4754–4765.

Coumans, E., and Bai, Y. 2016. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*.

Daw, N. D.; Niv, Y.; and Dayan, P. 2005. Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control. *Nature Neuroscience* 8(12):1704–1711.

Efroni, Y.; Dalal, G.; Scherrer, B.; and Mannor, S. 2018. Beyond the One-Step Greedy Approach in Reinforcement Learning. In *International Conference on Machine Learning*, 1386–1395.

Efroni, Y.; Ghavamzadeh, M.; and Mannor, S. 2019. Multi-Step Greedy and Approximate Real Time Dynamic Programming. *arXiv preprint arXiv:1909.04236*.

Evans, J. S. B. 1984. Heuristic and analytic processes in reasoning. *British Journal of Psychology* 75(4):451–468.

Kahneman, D. 2003. Maps of bounded rationality: Psychology for behavioral economics. *American economic review* 93(5):1449–1475.

Kahneman, D. 2011. *Thinking, fast and slow*. Macmillan.

Kingma, D., and Ba, J. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Puterman, M. L. 2014. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Samuel, A. L. 1967. Some studies in machine learning using the game of checkers. II - Recent progress. *IBM Journal of research and development* 11(6):601–617.

Schwartz, B.; Ward, A.; Monterosso, J.; Lyubomirsky, S.; White, K.; and Lehman, D. R. 2002. Maximizing versus satisficing: Happiness is a matter of choice. *Journal of personality and social psychology* 83(5):1178.

Sheppard, B. 2002. World-championship-caliber Scrabble. *Artificial Intelligence* 134(1-2):241–275.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S. 1991. Dyna, an integrated architecture

for learning, planning, and reacting. *ACM Sigart Bulletin* 2(4):160–163.

Veness, J.; Silver, D.; Blair, A.; and Uther, W. 2009. Bootstrapping from game tree search. In *Advances in neural information processing systems*, 1937–1945.

Wang, T.; Bao, X.; Clavera, I.; Hoang, J.; Wen, Y.; Langlois, E.; Zhang, S.; Zhang, G.; Abbeel, P.; and Ba, J. 2019. Benchmarking Model-Based Reinforcement Learning. *CoRR* abs/1907.02057.

Learning Heuristic Selection with Dynamic Algorithm Configuration

David Speck[1,*], **André Biedenkapp**[1,*], **Frank Hutter**[1,2],
Robert Mattmüller[1], **Marius Lindauer**[3]
[1]University of Freiburg, [2]Bosch Center for Artificial Intelligence, [3]Leibniz University Hannover
⟨speckd, biedenka, fh, mattmuel⟩@cs.uni-freiburg.de, lindauer@tnt.uni-hannover.de

Abstract

A key challenge in satisficing planning is to use multiple heuristics within one heuristic search. An aggregation of multiple heuristic estimates, for example by taking the maximum, has the disadvantage that bad estimates of a single heuristic can negatively affect the whole search. Since the performance of a heuristic varies from instance to instance, approaches such as algorithm selection can be successfully applied. In addition, alternating between multiple heuristics during the search makes it possible to use all heuristics equally and improve performance. However, all these approaches ignore the internal search dynamics of a planning system, which can help to select the most helpful heuristics for the current expansion step. We show that dynamic algorithm configuration can be used for dynamic heuristic selection which takes into account the internal search dynamics of a planning system. Furthermore, we prove that this approach generalizes over existing approaches and that it can exponentially improve the performance of the heuristic search. To learn dynamic heuristic selection, we propose an approach based on reinforcement learning and show empirically that domain-wise learned policies, which take the internal search dynamics of a planning system into account, can exceed existing approaches in terms of coverage.

Introduction

Heuristic forward search is one of the most popular and successful techniques in classical planning. Although there is a large number of heuristics, it is known that the performance, i.e. the informativeness, of a heuristic varies from instance to instance (Wolpert and Macready 1995; Droste, Jansen, and Wegener 2002). While in optimal planning it is easy to combine multiple admissible heuristic estimates using the maximum, in satisficing planning the estimates of inadmissible heuristics are difficult to combine in general (Röger and Helmert 2010). The reason for this is that highly inaccurate and uninformative estimates of a heuristic can have a negative effect on the entire search process when aggregating all estimates. Therefore, an important task in satisficing planning is to utilize multiple heuristics within one heuristic search.

Röger and Helmert (2010) introduced the idea of a search with multiple heuristics, maintaining a set of heuristics, each associated with a separate open list to allow switching between such heuristics. This bypasses the problem of aggregating different heuristic estimates, while the proposed alternating procedure uses each heuristic to the same extent. Another direction is the selection of the best algorithm or heuristic a priori based on the characteristics of the present planning instance (Seipp et al. 2012; Cenamor, de la Rosa, and Fernández 2016; Sievers et al. 2019). In other words, different search algorithms and heuristics are part of a portfolio from which one is selected to solve a particular problem instance. The automated procedure for performing the former is referred to as algorithm selection (Rice 1976) while optimization of algorithm parameters is referred to as algorithm configuration (Ansótegui, Sellmann, and Tierney 2009; Hutter et al. 2009; López-Ibáñez et al. 2016). Both methodologies have been successfully applied to planning (Fawcett et al. 2011; 2014; Seipp et al. 2015; Sievers et al. 2019) and various other areas of artificial intelligence such as machine learning (Snoek, Larochelle, and Adams 2012) or satisfiability solving (Hutter et al. 2017). However, algorithm selection and configuration ignore the non-stationarity of which configuration performs well. In order to remedy this, Biedenkapp et al. (2020) showed that the problem of selecting and adjusting configurations during the search based on the current solver state and search dynamics can be modelled as contextual Markov decision processes and addressed by standard reinforcement learning methods.

In planning, there is only little work that take into account the search dynamics of a planner to decide which planner to use. Cook and Huber (2016) showed that switching between different heuristic searches (planners) based on the search dynamics obtained during a search leads to better performance than a static selection of a heuristic. However, in this approach, several disjoint searches (planners) are executed, which do not share the search progress (Aine and Likhachev 2016). Ma et al. (2020) showed that a portfolio-based approach that can switch the planner at halftime, depending on the performance of the previously selected one, can improve

*Contact Author, Equal contribution

performance over a simple algorithm selection at the beginning. Recent works have investigated switching between different search strategies depending on the internal search dynamics of a planner (Gomoluch, Alrajeh, and Russo 2019; Gomoluch et al. 2020). One approach that shares the search progress is to maintain multiple heuristics as separate open lists (Röger and Helmert 2010). Furthermore, it has been shown that boosting, i.e., giving preference to heuristics that have recently made progress, can improve search performance (Richter and Helmert 2009). While in these works heuristic values are computed for each state, Domshlak, Karpas, and Markovitch (2010) investigated the question, whether the time spent for the computation of the heuristic value for a certain state pays off.

Another avenue of work considers how to "directly" create or learn new heuristic functions. One example is the work of Ferber, Helmert, and Hoffmann (2020), which utilizes supervised learning to learn a heuristic function where the input is the planning (world) state itself. Further, Virseda, Borrajo, and Alcázar (2013) used regression techniques to automatically learn good combinations of heuristics, which can lead to an informative heuristic estimate, but which are also learned a priori and do not adapt to current search dynamics and progress. Finally, Thayer, Dionne, and Ruml (2011) showed that admissible heuristics can be transformed online, into inadmissible heuristics, which makes it possible to tailor a heuristic to a specific planning instance.

In this work we introduce and define dynamic algorithm configuration (Biedenkapp et al. 2020) for planning by learning a policy that dynamically selects a heuristic within a search based on the current search dynamics. We prove that a dynamic adjustment of heuristic selection during the search can exponentially improve the search performance of a heuristic search compared to a static heuristic selection or a *non-adaptive* policy like alternating. Furthermore, we show that such a dynamic control policy is a strict generalization of other already existing approaches to heuristic selection. We also propose a set of state features describing the current search dynamics and a reward function for training a reinforcement learning agent. Finally, an empirical evaluation shows that it is possible to learn a dynamic control policy on a per-domain basis that outperforms approaches that do not involve search dynamics, such as ordinary heuristic search with a single heuristic and alternating between heuristics.

Background

We first introduce classical planning, then discuss greedy best-first search with multiple heuristics, and finally present the concept of dynamic algorithm configuration based on reinforcement learning. Note that the terminology and notation of planning and reinforcement learning are similar, so we use the symbol $\sim$ for all notations directly related to reinforcement learning; e.g. π denotes a plan of a planning task, while $\tilde{\pi}$ is a policy obtained by reinforcement learning.

Classical Planning

A problem instance or task in classical planning, modeled in the SAS$^+$ formalism (Bäckström and Nebel 1995), is a tuple $i = \langle \mathcal{V}, s_0, \mathcal{O}, s_\star \rangle$ consisting of four components. $\mathcal{V}$ is a finite set of state variables, each associated with a finite domain D_v. A fact is a pair (v, d), where $v \in \mathcal{V}$ and $d \in D_v$, and a partial variable assignments over $\mathcal{V}$ is a consistent set of facts, i.e. a set that does not contain two facts for the same variable. If s assigns a value to each $v \in \mathcal{V}$, s is called a state. States and partial variable assignments are functions which map variables to values, i.e. $s(v)$ is the value of variable v in state s (analogous for partial variable assignments). $\mathcal{O}$ is a set of operators, where an operator is a pair $o = \langle pre_o, \mathit{eff}_o \rangle$ of partial variable assignments called preconditions and effects, respectively. Each operator has cost $c_o \in \mathbb{N}_0$. The state s_0 is called the initial state and the partial variable assignment $s_\star$ specifies the goal condition, which defines all possible goal states $S_\star$. With $\mathcal{S}$ we refer to the set of all states defined over $\mathcal{V}$, and with $|i|$ we refer to the size of the planning task i, i.e. the number of operators and facts.

We call an operator $o \in \mathcal{O}$ applicable in state s iff pre_o is satisfied in s, i.e. $s \models pre_o$. Applying operator o in state s results in a state s' where $s'(v) = \mathit{eff}_o(v)$ for all variables $v \in \mathcal{V}$ for which eff_o is defined and $s'(v) = s(v)$ for all other variables. We also write $s[o]$ for s'. The objective of classical planning is to determine a plan, which is defined as follows. A *plan* $\pi = \langle o_0, \ldots, o_{n-1} \rangle$ for planning task i is a sequence of applicable operators which generates a sequence of states $s_0, \ldots, s_n$, where $s_n \in S_\star$ is a goal state and $s_{i+1} = s_i[o_i]$ for all $i = 0, \ldots, n-1$. The cost of plan π is the sum of its operator costs.

Given a planning task, the search for a good plan is called satisficing planning. In practice, heuristic search algorithms such as greedy best-first search have proven to be one of the dominant search strategy for satisficing planning.

Greedy Search with Multiple Heuristics

Greedy best-first search is a pure heuristic search which tries to estimate the distance to a goal state by means of a heuristic function. A heuristic is a function $h : \mathcal{S} \mapsto \mathbb{N}_0 \cup \{\infty\}$, which estimates the cost to reach a goal state from a state $s \in \mathcal{S}$. The perfect heuristic $h^\star$ maps each state s to the cost of the cheapest path from s to any goal state $s_\star \in S_\star$. The general idea of greedy best-first search with a single heuristic h is to start with the initial state and to expand the most promising states based on h until a goal state is found (Pearl 1984). During the search, relevant states are stored in an open list that is sorted by the heuristic values of the contained states in ascending order so that the state with the lowest heuristic values, i.e. the most promising state, is at the top. More precisely, in each step a state s with minimal heuristic value is expanded, i.e. its successors $S' = \{s[o] \mid o \in O\}$ are generated and states $s' \in S'$ not already expanded are added to the open list according to their heuristic values $h(s')$. Within an open list, for states with the same heuristic value (h-value) the tie-breaking rule that is used is according to the first-in-first-out principle.

In satisficing planning it is possible to combine multiple heuristic values for the same state in arbitrary ways. It has been shown, however, that the combination of several heuristic values into one, e.g. by taking the maximum or a (weighted) sum, does not lead to informative heuristic estimates (Röger and Helmert 2010). This can be explained

by the fact that if one or more heuristics provide very inaccurate values, the whole expansion process is affected. Röger and Helmert (2010) introduced the idea to maintain multiple heuristics $H = \{h_0, \dots, h_{n-1}\}$ within one greedy best-first search. More precisely, it is possible to maintain a separate open list for each heuristic $h \in H$ and switch between them at each expansion step while always expanding the most promising state of the currently selected open list. The generated successor states are then evaluated with *each* heuristic and added to the corresponding open lists. This makes it possible to share the search progress (Aine and Likhachev 2016). Especially, an alternation policy, in which all heuristics are selected one after the other in a cycle such that all heuristics are treated and used equally, has proven to be an efficient method. Such equal use of heuristics can help to progress the search space towards a goal state, even if only one heuristic is informative. However, in some cases it is possible to infer that some heuristics are currently, i.e. in the current region of the search space, more informative than others, which is ignored by a strategy like alternation. More precisely, with alternation, the choice of the heuristic depends only on the current time step and not on the current search dynamics or planner state. In general, however, it is possible to dynamically select a heuristic based on internal information provided by the planner. This is the key idea behind our approach described in the following.

Dynamic Algorithm Configuration

Automated algorithm configuration (AC) has proven a powerful approach to leveraging the full potential of algorithms. Standard AC views the algorithms being optimized as black boxes, thereby ignoring an algorithm's temporal behaviour and ignoring that an optimal configuration might be non-stationary (Arfaee, Zilles, and Holte 2011).

Dynamic algorithm configuration (DAC) is a new meta-algorithmic framework that makes it possible to learn to adjust the hyperparameters of an algorithm given a description of the algorithm's behaviour (Biedenkapp et al. 2020). We first describe DAC on a high level. Given a parameterized algorithm A with its configuration space $\tilde{\Theta}$, a set of problem instances $\mathcal{I}$ the algorithm has to solve, a state description $\tilde{s}_t^i$ of the algorithm A solving an instance $i \in \mathcal{I}$ at step $t \in \mathbb{N}_0$, and a reward signal $\tilde{r}$ assessing the reward of using a control policy $\tilde{\pi} \in \tilde{\Pi}$ to control A on an instance $i \in \mathcal{I}$ (e.g. runtime or number of state expansions), the goal is to find a (*dynamic*) control policy $\tilde{\pi}^* : \mathbb{N}_0 \times \tilde{\mathcal{S}} \times \mathcal{I} \to \tilde{\Theta}$, that adaptively chooses a configuration $\tilde{\theta} \in \tilde{\Theta}$ given a state $\tilde{s}_t \in \tilde{\mathcal{S}}$ of A at time $t \in \mathbb{N}_0$ to optimize the reward of A across the set of instances $\mathcal{S}$, i.e. $\tilde{\pi}^* \in \arg\max_{\tilde{\pi} \in \tilde{\Pi}} \mathbb{E}[\tilde{r}(\tilde{\pi}, i)]$. Note that the current time step $t \in \mathbb{N}_0$ and instance $i \in \mathcal{I}$ can be encoded in the state description $\tilde{\mathcal{S}}$ of an algorithm A, which leads to a dynamic control policy, which is defined as $\tilde{\pi}_{\text{dac}} : \tilde{\mathcal{S}} \to \tilde{\Theta}$.

Figure 1 depicts the interaction between a control policy $\tilde{\pi}$ and a planning system A schematically. At each time step t, the planner sends the current internal state $\tilde{s}_t^i$ and the corresponding reward $\tilde{r}_t^i$ to the control policy $\tilde{\pi}$ based on which the controller decides which parameter setting $h_{t+1} \in \tilde{\Theta}$ to use. The planner progresses according to the decision to

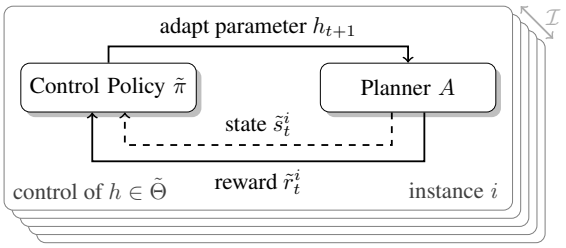


Figure 1: Dynamic configuration of parameter $h \in \tilde{\Theta}$ of algorithm A on an instance $i \in \mathcal{I}$, at time step $t \in \mathbb{N}_0$. Until i is solved or a budget is exhausted, the controller adapts parameter h, based on the internal state $\tilde{s}_t^i$ of A.

the next internal state $\tilde{s}_{t+1}^i$ with reward $\tilde{r}_{t+1}^i$. This formalisation of dynamic algorithm configuration makes it possible to recover prior meta-algorithmic frameworks as special cases which we discuss below.

Dynamic Heuristic Selection

In this section, we will explain how dynamic algorithm configuration can be used in the context of dynamic heuristic selection and how it differs from time-adaptive or in short *adaptive* algorithm configuration and algorithm selection, which have already been used in the context of search with multiple heuristics. Röger and Helmert (2010) introduced the idea of maintaining a set of heuristics H each associated with a separate open list in order to allow the alternation between such heuristics. Considering H as the configuration space $\tilde{\Theta}$ of a heuristic search algorithm A and each state expansion as a time step t, it is possible to classify different dynamic heuristic selection strategies within the framework of algorithm configuration. For example, alternation is an time-adaptive control policy because it maps each time step to a specific heuristic, i.e. configuration, independent of the instance or the state of the planner. The selection of a particular heuristic depending on the current instance before solving the instance, known as "portfolio planner", is an algorithm selection policy that depends only on the instance and not on the current time step or the internal state of the planner. Noteworthy exceptions are policies that compare the heuristic values of states, such as the expansion of the state with the overall minimal heuristic value or according to a Pareto-optimality analysis (Röger and Helmert 2010). Such policies depend on the current state of the planner, but ignore the time step and the current instance being solved. This indicates that all three components — the instance, the time step, and the state of the planner — can be important and helpful in selecting the heuristic for the next state expansion. The following summarizes the existing approaches to heuristic selection within the framework of algorithm configuration.

- *Algorithm Selection*:
  - Policy: $\tilde{\pi}_{\text{as}} : \mathcal{I} \to H$
  - Example: Portfolios (Seipp et al. 2012; Cenamor, de la Rosa, and Fernández 2016; Sievers et al. 2019)

- *Adaptive Algorithm Configuration*:
  - Policy: $\tilde{\pi}_{\text{aac}} : \mathbb{N}_0 \to H$
  - Example: Alternation (Röger and Helmert 2010)
- *Dynamic Algorithm Configuration*:
  - Policy: $\tilde{\pi}_{\text{dac}} : \mathbb{N}_0 \times \tilde{\mathcal{S}} \times \mathcal{I} \to H$
  - Example: Approach proposed in this paper

An Approach based on Reinforcement Learning

In this section, we describe all the parts required to dynamically configure a planning system so that for each individual time step, a dynamic control policy can decide which heuristic to use based on a dynamic control policy. Here, a time step is a single expansion step of the planning system.

State Description Learning dynamic configuration policies requires descriptive state features that inform the policy about the characteristics and the behavior of the planning system in the search space. Preferably, such features are domain-independent, such that the same features can be used for a wide variety of domains. In addition, such state features should be cheap to compute in order to keep the overhead as low as possible. As consequence of both desiderata and the intended learning task we propose to use the following state features computed over the entries contained in the corresponding open list of each heuristic:

$\max_h$**:** maximum h value for each heuristic $h \in H$;

$\min_h$**:** minimum h value for each heuristic $h \in H$;

μ_h**:** average h value for each heuristic $h \in H$;

σ_h^2**:** variance of the h values for each heuristic $h \in H$;

$\#_h$**:** number of entries for each heuristic $h \in H$;

t**:** current time/expansion step $t \in \mathbb{N}_0$.

To measure progress, we do not directly use the values of each state feature, but compute the *difference of each state feature* between successive time steps $t-1$ and t. The configuration space is a finite set of n heuristics to choose from, i.e., $\tilde{\Theta} = H = \{h_0, \ldots, h_{n-1}\}$.

Note that the described set of state features is domain independent, but does not contain any specific context information. In general, however, it is possible to describe an instance or domain with state features that describe, for example, the variables, operators or the causal graph (Sievers et al. 2019). If the goal is to learn robust policies that can handle highly heterogeneous sets of instances, it is possible to add contextual information about the planning instance at hand, such as the problem size or the required preprocessing steps (Fawcett et al. 2014), to the state description. However, in this work, limit ourselves to domain-wise dynamic control policies and show that the concept of DAC can improve heuristic search in theory and practice.

Reward Function Similar to the state description, the reward function we want to optimize should ideally be domain-independent, cheap and quick to compute. Since the goal is usually to quickly solve as many tasks as possible, a good reward feature should reflect this desire.

We use a reward of -1 for each expansion step that the planning system has to perform in order to find a solution. Using this reward function, a configuration policy learns to select heuristics that minimize the expected number of state expansions until a solution is found. This sparse reward function ignores aspects such as the quality of a plan, but its purpose is to reduce the search effort and thus improve search performance. Clearly, it is possible to define other reward functions with, e.g., dense rewards to make learning easier. We nevertheless demonstrate that already with our reward function and state features it is possible to learn dynamic control policies, which dominate algorithm selection and adaptive control policies in theory and practice.

Dynamic Algorithm Configuration in Theory

In optimal planning, where the goal is to find a plan with minimal cost, the performance of heuristic search can be measured by the number of state expansions (Helmert and Röger 2008). This is different for satisficing planning, because plans with different costs can be found and there are generally no "must expand" states that need to be expanded to prove that a solution is optimal. However, the number of state expansion until *any* goal state is found can be used to measure the guidance of a heuristic or heuristic selection (Richter and Helmert 2009; Röger and Helmert 2010).

We want to answer the question of whether it can theoretically be beneficial to use dynamic control policies $\tilde{\pi}_{\text{dac}}$ over algorithm selection policies $\tilde{\pi}_{\text{as}}$ or adaptive control policies $\tilde{\pi}_{aac}$. Proposition 1 proves that for each heuristic search algorithm in combination with each collection of heuristics there is a dynamic control policy $\tilde{\pi}_{\text{dac}}$ which is as good as $\tilde{\pi}_{\text{as}}$ or $\tilde{\pi}_{aac}$ in terms of state expansions. The key insight is that dynamic control policies are a strict generalization of algorithm selection policies and adaptive control policies that always allow the simulation of the former policies.

Proposition 1. *Independent of the heuristic search algorithm and the collection of heuristics, for each algorithm selection policy $\tilde{\pi}_{as}$ and adaptive algorithm configuration policy $\tilde{\pi}_{aac}$ there is a dynamic control policy $\tilde{\pi}_{dac}$ which expands at most as many states as $\tilde{\pi}_{as}$ and $\tilde{\pi}_{aac}$ until a plan is found for a given planning instance.*

Proof. DAC policies generalize algorithm selection and adaptive algorithm configuration policies, thus it is always possible to define $\tilde{\pi}_{\text{dac}}$ as $\tilde{\pi}_{\text{dac}} = \tilde{\pi}_{\text{as}}$ or $\tilde{\pi}_{\text{dac}} = \tilde{\pi}_{\text{aac}}$. □

With Proposition 1 it follows directly that an optimal algorithm configuration policy $\tilde{\pi}^*_{\text{dac}}$ is at least as good as an optimal algorithm selection policy $\tilde{\pi}^*_{\text{as}}$ and an optimal adaptive algorithm configuration policy $\tilde{\pi}^*_{\text{aac}}$.

Corollary 2. *Independent of the heuristic search algorithm and the collection of heuristics, an optimal dynamic control policy $\tilde{\pi}^*_{dac}$ expands at most as many states as an optimal algorithm selection policy $\tilde{\pi}^*_{as}$ and an optimal adaptive algorithm configuration policy $\tilde{\pi}^*_{aac}$ until a plan π is found for a planning task.* □

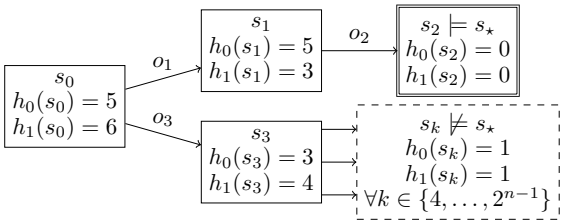


Figure 2: Visualization of the induced transition system of the planning task family i_n.

It is natural to ask the question to what extent the use of a dynamic control policy instead of an algorithm selection or an adaptive control policy can improve the search performance of heuristic search. We will show that for each algorithm selection policy $\tilde{\pi}_{\text{as}}$ and adaptive algorithm configuration policy $\tilde{\pi}_{\text{aac}}$, we can construct a family of planning tasks so that a dynamic control policy $\tilde{\pi}_{\text{dac}}$ will expand exponentially fewer states until a plan is found. For this purpose, we introduce a family of planning instances i_n with $O(n)$ propositional variables and $O(n)$ operators. The induced transition system of i_n is visualized in Figure 2. There is exactly one goal path s_0, s_1, s_2, which is induced by the unique plan $\pi = \langle o_1, o_2 \rangle$. Furthermore, exactly two states are directly reachable from the initial state, s_1 and s_3. While state s_1 leads to the unique goal state s_2, from s_3 onward exponentially many states $s_4, \ldots, s_{2^n-1}$ in $n = |i_n|$, i.e. $\Omega(2^n) = \Omega(2^{|i_n|})$, can be reached by the subsequent application of multiple actions.

Theorem 3. *For each adaptive algorithm configuration policy $\tilde{\pi}_{aac}$ there exists a family of planning instances i_n, a collection of heuristics H and a dynamic control policy $\tilde{\pi}_{dac}$, so that greedy best-first search with H and $\tilde{\pi}_{aac}$ expands exponentially more states in $|i_n|$ than greedy best-first search with H and $\tilde{\pi}_{dac}$ until a plan π is found.*

Proof. Let $\tilde{\pi}_{\text{aac}}$ be an adaptive algorithm configuration policy. Now, we consider the family of planning tasks i_n (Figure 2) with $|i_n| = O(n)$ and a collection of two heuristics $H = \{h_0, h_1\}$. The heuristic estimates of h_0 and h_1 are shown in Figure 2 and the open lists of greedy best-first search at each time step t are visualized in Figure 3. In time step 0, it is irrelevant which heuristic is selected, always leading to time step 1, where state s_3 is the most promising state according to heuristic h_0, while state s_1 is the most promising state according to heuristic h_1. In time step 1, $\tilde{\pi}_{\text{aac}}$ can either select heuristic h_0 or h_1. We first assume that $\tilde{\pi}_{\text{aac}}$ selects h_0 so that state s_3 is expanded, leading to exponentially many states s_k, which are all evaluated with $h_0(s_k) = h_1(s_k) = 1$ and thus are all expanded before s_1. Therefore, the unique goal state s_2 is found after all other states in the state space $\mathcal{S}$ have been expanded.

In comparison, for $\tilde{\pi}_{\text{dac}}$ we can pick the policy that always selects the heuristic with minimum average heuristic value of all states in the corresponding open list, i.e. $\arg\min_{h \in H} \mu_h$. Following $\tilde{\pi}_{\text{dac}}$, first h_0 and then h_1 is selected, generating the goal state s_2 in time step 1. Therefore,

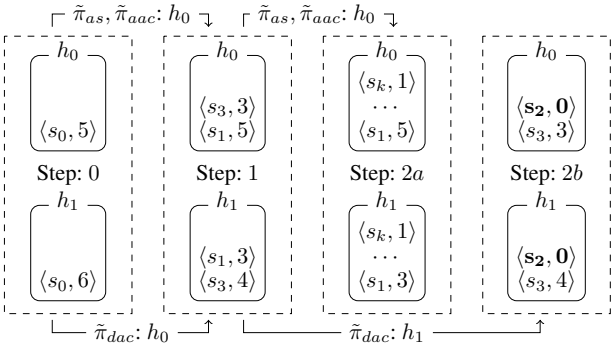


Figure 3: Visualization of two heuristics used to solve an instance of the planning task family i_n.

$\tilde{\pi}_{\text{dac}}$ only expands 2 states, while $\tilde{\pi}_{\text{aac}}$ expands 2^{n-2} states until a goal state is found.

Finally, for a policy $\tilde{\pi}_{\text{aac}}$ that selects h_1 at time step 1, it is possible to swap the heuristic estimates of h_0 and h_1 in the constructed collection of heuristics, resulting in the same number of state extensions. □

Theorem 4. *For each algorithm selection policy $\tilde{\pi}_{as}$ there exists a family of planning instances i_n, a collection of heuristics H and a dynamic control policy $\tilde{\pi}_{dac}$, so that greedy best-first search with H and $\tilde{\pi}_{as}$ expands exponentially more states in $|i_n|$ than greedy best-first search with H and $\tilde{\pi}_{dac}$ until a plan π is found.*

Proof. Let $\tilde{\pi}_{\text{as}}$ be an algorithm selection policy. Now, we consider the family of planning tasks i'_n, which is similar to the family of planning tasks i_n (Figure 2), with one modification: the goal state s_2 is not directly reachable from s_1, but via an additional state s'. In other words, we insert the state s' between s_1 and s_2. Furthermore, we again consider a collection of two heuristics $H = \{h_0, h_1\}$ with the heuristic estimates shown in Figure 2 and $h_0(s') = 2$ and $h_1(s') = 10$. The idea is that both heuristics alone lead to the expansion of exponentially many states, whereas a dynamic switch of the heuristic only leads to constantly many expansions.

Policy $\tilde{\pi}_{\text{as}}$ selects exactly one heuristic, h_0 or h_1, for each planning task. If h_0 is selected, with the same argument used in the proof of Theorem 3, exponentially many states in $|i'_n|$ are expanded. If h_1 is selected, in time step 2, states s_3 and s' are contained in both open lists. According to h_1, state s_3 is more promising than s', which leads again to an expansion of exponentially many states in $|i'_n|$.

In comparison, for $\tilde{\pi}_{\text{dac}}$ we pick again the policy that always selects the heuristic with minimum average heuristic value of all states in the corresponding open list, i.e. $\arg\min_{h \in H} \mu_h$. Policy $\tilde{\pi}_{\text{dac}}$ selects first h_0, followed by h_1 and again h_0, resulting in the generation of the goal state after three state extensions. □

In Theorems 3 and 4 we assume for simplicity that expanded states are directly removed from all open lists. In practice, open lists are usually implemented as min-heaps, and it is costly to search and remove states immediately.

Therefore, states that have already been expanded are kept in the open lists and ignored as soon as they have reached the top. We note that this does not affect the theoretical results.

Finally, it is important to emphasize that Proposition 1, Corollary 2 and Theorems 3 and 4 are theoretical results. All results are based on the assumption that it is possible to learn good dynamic control policies. Next, we show that it is possible in practice to learn such dynamic control policies.

Empirical Evaluation

We conduct experiments[1] to measure the performance of our reinforcement learning (RL) approach on domains of the International Planning Competition (IPC). For each domain, the RL policies are trained on a training set and evaluated on a disjoint prior unseen test set of the same domain. Note that such policies are not domain-independent, although it is generally possible to add instance- and domain-specific information to the state features. We leave the task of learning domain-independent policies for future work.

Setup

All experiments are conducted with FAST DOWNWARD (Helmert 2006) as the underlying planning system. We use ("eager") greedy best-first search (Richter and Helmert 2009) and min-heaps to represent the open lists (Röger and Helmert 2010). Furthermore, we implemented an extension for FAST DOWNWARD, which makes it possible to communicate with a controller (dynamic control policy) via TCP/IP and thus to send relevant information (state features and reward) in each time/expansion step and to receive the selected parameter (heuristic). This architecture allows the planner and controller to be decoupled, making it easy to replace components. We considered four different heuristic estimators as configuration space, i.e. $\tilde{\Theta} = H = \{h_{\text{ff}}, h_{\text{cg}}, h_{\text{cea}}, h_{\text{add}}\}$ which can be changed at each time step:

- h_{ff}: the FF heuristic (Hoffmann and Nebel 2001),
- h_{cg}: the causal graph heuristic (Helmert 2004),
- h_{cea}: the context-enhanced additive heuristic (Helmert and Geffner 2008), and
- h_{add}: the additive heuristic (Bonet and Geffner 2001).

For the evaluation of the test set, i.e. the final planning runs, we used a maximum of 4 GB memory and 5 minutes runtime. All experiments were run on a compute cluster with nodes equipped with two Intel Xeon Gold 6242 32-core CPUs, 20 MB cache and 188GB (shared) RAM running Ubuntu 18.04 LTS 64 bit.

Similar to Biedenkapp et al. (2020), we use ϵ-greedy deep Q-learning in the form of a double DQN (van Hasselt, Guez, and Silver 2016) implemented in CHAINER (Tokui et al. 2019) (CHAINERRL v0.7.0) to learn the dynamic control policies. The networks are trained using ADAM[2] (Kingma and Ba 2014) for 10^6 update steps. In order to avoid bad policies being executed arbitrarily long during training, we use a cutoff of $7\,500$ control/expansion steps. Although some instances are not solved within this cutoff, even with the optimal policy, the underlying assumption is that good policies for smaller instances generalize to larger instances within a domain. Note that it is in general also possible to add a certain time cutoff. To determine the quality of a learned policy, we evaluated it every $30\,000$ steps during training and save the best policy we have seen so far. In total, we performed 5 independent runs of our control policies for each domain, for which we report the average performance. The policies are represented by neural networks for which we determined the hyperparameters in a white-box experiment on a new artifical domain and kept these hyperparameters fixed for all domains in the experiments.

White-Box Experiments. We conducted preliminary experiments on a newly created ARTIFICIAL domain with two artificial heuristics. This domain is designed so that in each step, only one of two heuristics is informative. In other words, similar to the constructed example in the proof of Theorem 4, at each time step, only one heuristic leads to the expansion of a state which is on the shortest path to a goal state. In order to obtain a good control policy that leads to few state expansions, it is necessary to derive a *dynamic* control policy from the state features. We generated 30 training instances on which we performed a small grid search over the following parameters *#layers* $\in \{2, 5\}$, *hidden units* $\in \{50, 75, 150, 200\}$ and *epsilon decay* $\in \{2.5 \times 10^5, 5 \times 10^5\}$. We determined that a 2-layer network with 75 hidden units and a linear decay for ϵ over 5×10^5 steps from 1 to 0.1 worked best.

Interestingly, it was possible to learn policies with a performance close to the optimal policy, see Figure 4. Both individual heuristics perform poorly (even when using an oracle selector). Randomly deciding which heuristic to play performs nearly as good as the alternating strategy that alternates between the heuristics at each step. In the beginning the learned policy needs some time to figure out in which states a heuristic might be preferable. However, it quickly learns to choose the correct heuristic, outperforming all other methods and nearly recovering the optimal policy.

Experiments

We evaluated the performance of our RL approach on six domains of the International Planning Competition (IPC). These domains were chosen because there are instance generators available online[3] that make it possible to create a suitable number of instances of different sizes. Furthermore, instances of these domains usually require a significant number of state expansions in order to find a plan. For this purpose, we generated 200 instances for all domains and randomly divided them into disjoint training and test sets with the same size of 100 instances each. For each domain we trained five dynamic control policies on the training set and compared them with other approaches on the unseen test set. We are mainly interested in comparing different policies for heuristic selection, which is why, here, the planner

[1] Resources: https://github.com/speckdavid/rl-plan

[2] We use CHAINER's v0.7.0 default parameters for ADAM.

[3] https://github.com/AI-Planning/pddl-generators

Algorithm	CONTROL POLICY			SINGLE HEURISTIC				BEST AS (ORACLE)		
Domain (# Inst.)	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}	**RL**	ALT	SINGLE h
BARMAN (100)	**84.4**	83.8	83.3	66.0	17.0	18.0	18.0	**89.0**	84.0	67.0
BLOCKSWORLD (100)	**92.9**	83.6	83.7	75.0	60.0	92.0	92.0	**96.3**	88.0	93.0
CHILDSNACK (100)	**88.0**	86.2	86.7	75.0	86.0	86.0	86.0	**88.0**	**88.0**	86.0
ROVERS (100)	95.2	**96.0**	**96.0**	84.0	72.0	68.0	68.0	**96.0**	**96.0**	91.0
SOKOBAN (100)	87.7	87.1	87.0	88.0	**90.0**	60.0	89.0	88.6	87.0	**92.0**
VISITALL (100)	56.9	51.0	51.5	37.0	**60.0**	**60.0**	**60.0**	**61.4**	52.0	60.0
SUM (600)	**505.1**	487.7	488.2	425.0	385.0	384.0	413.0	**519.3**	495.0	489.0

Table 1: Average coverage of different policies for the selection of a heuristic in each expansion step when evaluating the strategies on the prior unseen *test* set. The first three columns are control policies, the next four are individual heuristic searches, while the last three represent the best algorithm selection of the corresponding strategies, i.e. oracle selector for each instance.

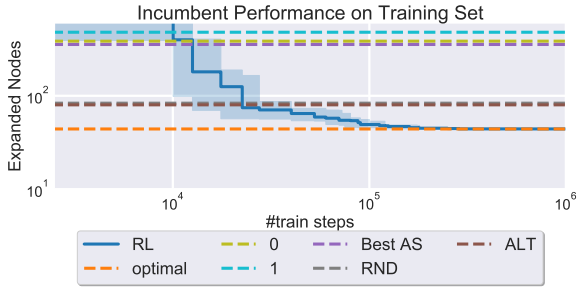


Figure 4: Performance of the best learned policy during training (RL), compared to the the performance of the individual heuristics (0 & 1), the oracle selector (BEST AS), an alternating schedule (ALT), a random policy (RND) and the optimal policy. Dashed lines indicate the performance of our baselines, the solid line the mean performance and the shaded area the standard deviation of our approach.

always maintains all four open lists, even if only one heuristic is used, and the controller, i.e. the dynamic control policy, alone decides which heuristic is selected.

Table 1 shows the percentage of solved instances per domain, i.e. the average coverage, on the test set. Each domain has a score in the range of 0-100, with larger values indicating more solved instances on average. More precisely, it is possible to obtain a score between 0 and 1 for each planning instance. A value of 0 means that the instance was never solved by the approach, 0.5 means that the instance was solved in half the runs, and 1 means that the instance was always solved. These scores are added up to give the average coverage per domain.

The first three columns correspond to control policies. Entry RL is the average coverage of the five trained dynamic control policies based on reinforcement learning, each averaging over 25 runs with different seeds. Entry RND denotes the average coverage of 25 runs, where a random heuristic is selected in each step. Entry ALT stands for the average over all possible permutations of the execution of alternation. Note that there are $4! = 24$ different ways of executing alternation with four different heuristics. The SINGLE HEURISTIC columns show the coverage when only the corresponding heuristic is used. Finally, the columns for selecting the best algorithm selection (BEST AS) stand for the use of an oracle selector, which selects the best configuration of the corresponding technique for each instance. In other words, the best algorithm selection for RL is to choose the best dynamic control policy from the five trained policies for each instance, the best algorithm selection for ALT is to choose the best permutation of alternation for each instance and the best algorithm selection for SINGLE h is to choose the best heuristic for each instance.

The results of Table 1 show that our approach (RL) performs best on average in terms of coverage (individual coverage of the five trained RL policies: 505.4, 500.6, 501.6, 507.4, 510.1). ALT is slightly better than the uniform randomized choice of a heuristic RND, which indicates that the most important advantage of ALT is to use each heuristic equally with frequent switches and not to switch between them systematically. Furthermore, consistent with the results of Röger and Helmert (2010), single heuristics perform worse than the use of multiple heuristics. Interestingly, in the domain VISITALL, single heuristics have the highest coverage and while RND and ALT have a low coverage, RL performs better. This indicates that in this domain, the dynamic control policies of RL were able to infer that a static policy is good or to exclude certain single heuristics. In BLOCKSWORLD, RL has the highest coverage among all approaches. A possible explanation is that a dynamic policy is the key to solving difficult instances in this domain. This assumption is supported by the observation that the best algorithm selection, i.e. the oracle selection of RL, clearly exceeds the other approaches in BLOCKSWORLD. Finally, in ROVER, the use of multiple heuristics seems to be important, and while RL scores better than using single heuristics, the learned policy scores worse than RND and ALT. This may be due to overfitting which we will discuss below.

Considering the columns of best algorithm selection, it is possible to observe that an oracle single heuristic selection or oracle alternating selection would not perform better than the average performance of our learned RL which policies shows that 1) heuristic search with multiple heuristics can in practice benefit from dynamic algorithm configuration and 2) it is possible to learn good dynamic policies domain-wise. Even under the unrealistic circumstances of an *optimal* algorithm selector, our learned policies perform better and thus outperform all possible algorithm selection policies.

Algorithm	CONTROL POLICY			SINGLE HEURISTIC			
Metric	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}
COVERAGE	**84.2**	81.3	81.4	70.8	64.2	64.0	68.8
GUIDANCE	**38.5**	37.4	37.5	30.8	27.6	28.6	30.4
SPEED	**66.6**	62.8	62.8	54.9	50.4	50.3	54.0
QUALITY	**76.2**	76.0	76.0	65.8	57.6	56.2	60.9

(a) Test set

Algorithm	CONTROL POLICY			SINGLE HEURISTIC			
Metric	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}
COVERAGE	**87.0**	83.6	83.0	71.7	64.3	65.0	68.5
GUIDANCE	**39.8**	38.3	38.4	31.4	26.6	28.8	30.2
SPEED	**69.3**	65.3	65.4	56.0	49.1	51.1	54.2
QUALITY	**79.5**	77.9	77.5	66.8	57.3	58.0	61.3

(b) Training set

Table 2: A comparison of different control policies and single heuristic search measuring coverage, guidance, speed and solution quality on the prior unseen *test* set (a) and the *training* set (b). A higher score means better performance for all four metrics.

Table 2 shows four different metrics including the coverage from above. We additionally evaluate the guidance, speed and quality for each approach with a rating scale (Richter and Helmert 2009; Röger and Helmert 2010). For *guidance*, tasks solved within one state expansion get one point, while unsolved tasks or tasks solved with more than 10^6 state expansions get zero points. Between these extremes the scores are interpolated logarithmically. For *speed* the algorithm gets one point for tasks solved within one second, while the algorithm gets zero points for unsolved tasks or tasks solved in 300 seconds. For *quality* the algorithm gets a score of c^*/c for a solved task, where c is the cost of the reported plan and c^* is the cost of the best plan found with any approach. Finally, the sum of each metric is divided by the number of domains to obtain a total score between 0 and 100. Considering those metrics, control policies perform better than single heuristic approaches. Furthermore, dynamic control polices obtained by RL perform best according to all metrics. However, this analysis favors approaches which solve more instances than others. Recall that plan quality is not taken into account when learning a policy, which explains the small advantage of RL in plan quality, even though more instances have been solved by RL.

Next we compare the performance of our approach RL on the *training* set (Table 2b) with the performance of RL on the test set (Table 2a). It is possible to observe that RL performs better on the *training* set which can be attributed to overfitting and can explain why in some instances the performance of RL is worse than other approaches on the *test* set (see column RL of Table 2a and 2b). This issue of RL can be addressed in several ways, such as tuning the hyperparamters, expanding the training set or adding state features.

Finally, we want to mention the computational overhead of our RL approach compared to ALT and SINGLE HEURISTIC search approaches. While the performance of RL still exceeds the SINGLE HEURISTIC search of FAST DOWNWARD for all four heuristics, RL performs slightly worse than the internal heuristic alternation strategy of FAST DOWNWARD. In the future the overhead can be reduced by integrating the reinforcement learning part directly in FAST DOWNWARD instead of communicating via TCP/IP.

Conclusion

We theoretically and empirically evaluated the use of dynamic algorithm configuration for planning. More specifically, we have shown that dynamic algorithm configuration can be used for dynamic heuristic selection that takes into account the internal search dynamics of a planning system. Dynamic policies for heuristic selection generalize policies of existing approaches like algorithm selection and adaptive algorithm control, and their use can improve search performance exponentially. We presented an approach based on dynamic algorithm configuration and showed empirically that it is possible to learn policies capable of outperforming other approaches in terms of coverage.

For future work we will investigate domain-specific state features to learn domain-independent dynamic policies. Further, it is possible to dynamically control several parameters of a planner and to switch dynamically between different search algorithms. This raises the question how the search progress (Aine and Likhachev 2016) can be shared when using different search strategies. In particular, if we want to combine different search techniques, such as heuristic search (Bonet and Geffner 2001), symbolic search (Torralba et al. 2017; Speck, Geißer, and Mattmüller 2018) and planning as satisfiability (Kautz and Selman 1992; Rintanen 2012), it is an open question how to share the search progress.

Acknowledgments David Speck was supported by the German Research Foundation (DFG) as part of the project EPSDAC (MA 7790/1-1). André Biedenkapp, Marius Lindauer and Frank Hutter acknowledge funding by the Robert Bosch GmbH.

References

Aine, S., and Likhachev, M. 2016. Search portfolio with sharing. In *Proc. ICAPS 2016*, 11–19.

Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. of CP'09*, 142–157.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *AIJ* 175:2075–2098.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Biedenkapp, A.; Bozkurt, H. F.; Eimer, T.; Hutter, F.; and Lindauer, M. 2020. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In *Proc. of ECAI'20*.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2016. The IBaCoP planning system: Instance-based configured portfolios. *JAIR* 56:657–691.

Cook, B., and Huber, M. 2016. Dynamic heuristic planner selection. In *Proc. SMC 2016*, 2329–2334.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *Proc. AAAI 2010*.

Droste, S.; Jansen, T.; and Wegener, I. 2002. Optimization with randomized search heuristics - the (A)NFL theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science* 287(1):131–144.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Automated configuration of Fast Downward. In *IPC 2011 planner abstracts*, 31–37.

Fawcett, C.; Vallati, M.; Hutter, F.; Hoffmann, J.; Hoos, H.; and Leyton-Brown, K. 2014. Improved features for runtime prediction of domain-independent planners. In *Proc. ICAPS 2014*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural network heuristics for classical planning: A study of hyperparameter space. In *Proc. ECAI 2020*.

Gomoluch, P.; Alrajeh, D.; and Russo, A. 2019. Learning classical planning strategies with policy gradient. In *Proc. ICAPS 2019*, 637–645.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2020. Learning neural search policies for classical planning. In *Proc. ICAPS 2020*, 522–530.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *Proc. ICAPS 2008*, 140–147.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proc. AAAI 2008*, 944–949.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hutter, F.; Hoos, H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: An automatic algorithm configuration framework. *JAIR* 36:267–306.

Hutter, F.; Lindauer, M.; Balint, A.; Bayless, S.; Hoos, H.; and Leyton-Brown, K. 2017. The configurable SAT solver challenge (CSSC). *AIJ* 243:1–25.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. ECAI 1992*, 359–363.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs.LG]*.

López-Ibáñez, M.; Dubois-Lacoste, J.; Caceres, L. P.; Birattari, M.; and Stützle, T. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3:43–58.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Adaptive planner scheduling with graph neural networks. In *Proc. AAAI 2020*. 5077–5084.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65–118.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, 273–280.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *AIJ* 193:45–86.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, 246–249.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning portfolios of automatically tuned planners. In *Proc. ICAPS 2012*, 368–372.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Proc. AAAI 2015*, 3364–3370.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proc. AAAI 2019*, 7715–7723.

Snoek, J.; Larochelle, H.; and Adams, R. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NeurIPS'12*, 2960–2968.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018. Symbolic planning with edge-valued multi-valued decision diagrams. In *Proc. ICAPS 2018*, 250–258.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proc. ICAPS 2011*, 250–257.

Tokui, S.; Okuta, R.; Akiba, T.; Niitani, Y.; Ogawa, T.; Saito, S.; Suzuki, S.; Uenishi, K.; Vogel, B.; and Yamazaki, H. V. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proc. of KDD'19*, 2002–2011.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *AIJ* 242:52–79.

van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *Proc. of AAAI'16*, 2094–2100.

Virseda, J.; Borrajo, D.; and Alcázar, V. 2013. Learning heuristic functions for cost-based planning. In *ICAPS 2013 Workshop on Planning and Learning*.

Wolpert, D. H., and Macready, W. G. 1995. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.

Knowing When To Look Back: Bidirectional Rollouts in Dyna-style Planning

Yat Long Lo, [1, 3] Jia Pan, [1] Albert Y.S. Lam [2, 3]
[1]Department of Computer Science, University of Hong Kong
[2]Department of Electrical and Electronic Engineering, University of Hong Kong
[3] Fano Labs, Hong Kong

Abstract

In model-based reinforcement learning (MBRL), a model of the world is used to generate transitional data for an agent to learn from, in order to reduce sample complexity. Dyna is one of the most widely adopted MBRL frameworks that perform planning, acting, and learning in an online manner. Most of the previous works on Dyna perform one-step rollouts from sampled states to generate data based on the agent's history. Recently, it has been shown that planning shape and directionality (forward or backward planning) of the rollouts can impose a significant impact on the performance given a fixed planning budget. In this work, we conduct a systematic study on how these two factors affect the performance of model-based agents. We hypothesize that forward planning and backward planning serve complementary purposes, i.e. exploration and value propagation, in which careful state-dependent allocation of planning budget can improve learning efficiency. We further provide an online method to automate the decision between forward planning and backward planning using error-based epistemic uncertainty. We examine our proposed method in the tabular and linear function approximation settings for both perfect and learned models on GridWorld and Cartpole environments and propose the use of an ensemble of world models to counter compounding errors of long rollouts in the learned models. Our results show that both planning shape and directionality have a profound impact on Dyna methods' efficacy and bidirectional rollouts can improve learning efficiency using the same number of planning steps.

1 Introduction

With numerous successes of model-free reinforcement learning (RL) in various tasks from video game playing (Volodymyr et al. 2015) to robotics control (Gu et al. 2017), model-based RL has gained wide interest in the research community as a means to reduce the number of interactions with the environment, i.e., sample complexity, which can be expensive when being applied to real-world problems. Model-based RL commonly contains a model of the world that can be either hand-crafted or learned. Akin to how humans imagine scenarios in our heads, an agent learns from the data generated by the world model without actually taking actions within the real environment. The learning process from these generated data is known as planning.

Dyna (Sutton 1991) is a general framework of reinforcement learning that combines both model-free and model-based RL into a single coherent architecture. This framework has the appealing property of asynchronous processing of planning and learning in which an agent's decision-making and learning processes can operate at the same time as the world model is learning and planning. The world model is learned from an agent's real experience while both real and simulated experiences generated by the world model are used to update the value function or policy. We refer to the use of simulated experience to learn here as Dyna-style planning. Given the general and learning-algorithm-independent nature of Dyna, we study different planning properties following this framework.

Dyna offers flexible control over the planning process by how much computational budget is devoted to planning and how is the budget distributed (Holland, Talvitie, and Bowling 2018). Most of the previous works on Dyna, including the original Dyna-Q algorithm (Sutton 1991), performed one-step rollouts from sampled states to perform planning. The potential benefits of using such flexible control are less investigated. Two major factors over such planning control are planning shape (Holland, Talvitie, and Bowling 2018) and directionality. Holland, Talvitie, and Bowling (2018) studied the impact of planning shape on performance, demonstrating that longer or mid-length rollouts are often more beneficial than one-step rollouts for the same amount of planning steps. In terms of directionality, forward planning is the most common that the world model generates the reward and the next state given a sampled state and action. Moore and Atkeson (1993) used a form of backward planning by prioritizing states of high error and updating the values of their predecessors, which was shown to improve learning efficiency by propagating values along a trajectory faster. Edwards, Downs, and Davidson (2018) proposed forward-backward RL that uses a backward world model to generate data starting from goal states to handle sparse reward environments, but it assumed the knowledge of goal states and only performed rollouts from those goal

states. Overall, to our best of our knowledge, there is no systematic study on both planning shape and directionality and methods that take advantage of both forward and backward planning.

With a focus on planning shape (i.e., rollout lengths), we encounter the issue of compounding of errors (Asadi, Misra, and Littman 2018) in longer rollouts. The issue originates from the inaccuracy of generated data by learned models that can detrimentally affect performance. This is exacerbated when the rollouts are long, outweighing any benefits the rollout lengths can bring. To mitigate this problem, Talvitie (2014) and Venkatraman, Hebert, and Bagnell (2015) proposed hallucination to prepare the model to handle the fake and inaccurate inputs generated by itself. Asadi et al. (2019) proposed using a multi-step model directly to avoid feeding generated outputs back to the world model.

In this paper, we first conduct a systematic study on how both planning shape and directionality affect performance. Then, we hypothesize that forward planning and backward planning serve complementary roles of exploration and value propagation. We propose an automatic method for bidirectional rollouts that takes advantage of their hypothesized roles using error-based epistemic uncertainty. For the same amount of planning budget, we show that our proposed method for bidirectional rollouts produces the best overall performance among different planning shapes when compared with only-forward rollouts and only-backward rollouts, offering supporting evidence to our hypothesis for roles of forward planning and backward planning. We test our method in the tabular and linear function approximation settings for both perfect and learned models on GridWorld and Cartpole environments. Additionally, we propose the use of an ensemble of world models to counter compounding errors of long rollouts in learned models that drop inaccurate data based on the degree of disagreement among them. The simple and straightforward approach is found to be effective in avoiding catastrophic failures in long rollouts.

2 Background

2.1 Reinforcement Learning

In reinforcement learning, an agent interacts with its environment by taking actions at discrete time steps $t = 0, 1, 2, \cdots$. The environment is commonly formulated as a Markov Decision Process (MDP) with states $\mathcal{S}$, actions $\mathcal{A}$, transition probabilities $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, rewards $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ and discount function $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ (White 2017). At each time step t, the agent is in a state S_t, and takes an action A_t. In response, the environment emits a reward R_{t+1} and takes the agent to a state S_{t+1}. The goal of the agent is to maximize the return, defined as the discounted sum of the cumulative rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{1}$$

In this paper, as we focus on the rollout mechanisms in Dyna-style planning, the methods used should be independent of the choice of learning algorithms. Thus, we use Q-learning (Watkins and Dayan 1992) as our default learning algorithm to isolate the effect of the planning parameters that we are interested in. In Q-learning, the agent learns to approximate the state-action value function and acts near-greedily according to those state-action values. The state-action values for a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ are the expected return for that policy beginning from state s and action a:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a], \tag{2}$$

where $\mathbb{E}_\pi[\cdot]$ denotes taking expectation under policy π.

For linear function approximation, We parameterize the state-action value functions, denoted as $\hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$, where $\boldsymbol{\theta}_t$ refers to the weight vectors. State-action value functions are commonly learned by bootstrapping the state-action value of the next state, minimizing the temporal difference error. The update rule for Q-learning to learn state-action value functions is as follows:

$$\begin{aligned} \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(R_{t+1} + \gamma \max_{A_{t+1}} \hat{q}(S_{t+1}, A_{t+1}) \\ - \hat{q}(S_t, A_t)) \nabla_{\boldsymbol{\theta}} \hat{q}(S_t, A_t), \end{aligned} \tag{3}$$

where α is the learning rate. Each linear function approximator takes in a state feature vector as an input and produces state-action value for each possible action.

2.2 Model-Based Reinforcement Learning

Dyna Learning the value function often requires a huge number of samples. Model-based approaches aim to lower such sample complexity using world models to generate imagined experiences. Dyna-Q (Sutton 1991) learns a value function from both real and imagined experiences using the same update rule as Q-learning. Specifically, a world model is used to generate a complete transition to be used by a learning algorithm. The world models are either handcrafted or learned. Here, we parameterize learned world models with neural networks.

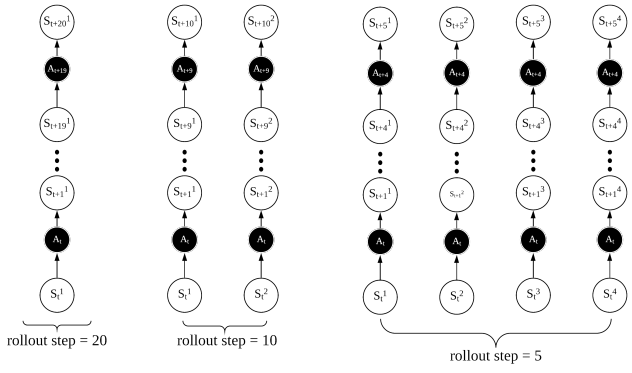


Figure 1: Different planning shapes

Planning Shape Planning shape is a term coined by Holland, Talvitie, and Bowling (2018) to formalize how a planning budget is distributed. The budget refers to the number of planning steps per real step taken in the environment. The

distribution can take many shapes. For instance, as illustrated in Figure 1, if the planning budget is 20 steps, an agent can sample one state for 20 rollout steps, sample two states for 10 rollout steps, sample four states for 5 rollout steps, and so on. It is found that the distribution can have a profound impact on the performance of a Dyna agent. Specifically, 1-step rollouts do not seem to benefit performance when compared with longer rollouts.

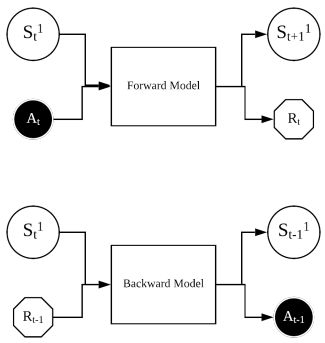


Figure 2: Forward and backward models

Directionality As shown in Figure 2, forward and backward model have different input-output relationships. Same as the way defined in Sutton and Barto (2018), a forward model takes in a state and an action to produce the corresponding next state and reward. Rollouts can be performed iteratively by feeding the next state and a sampled action into the forward model. On the other hand, a backward model takes in a state and a reward to produce a corresponding previous state and the action that led to the current state. Using either the forward or backward model gives the agent a complete transition to learn but the value of the state that is being updated is different. Note that another possible variant of the backward model is to also predict the reward that an agent gets to reach a particular state. This removes the assumption of the knowledge of a reward function but increases the learning burden of the agent. In this work, we assume that we have a reward oracle that tells the agent the correct reward as it rolls out backward.

3 Bidirectional Rollouts Using Epistemic Uncertainty

Given a fixed planning budget, when should an agent perform forward and backward planning? If we crudely equate learning from data generated by world models with animals' imagination, the question would become when one should look ahead and back from a scenario (a state). We hypothesize that forward and backward plannings serve complementary purposes of exploration and value propagation, respectively. Specifically, forward planning helps improve the value estimation of a state by generating unfamiliar experiences of possible futures. On the other hand, backward planning facilitates propagating the value learned for a particular state back to its predecessors. Hence, as the value estimation of a state-action pair improves through actual environment learning and forward planning, backward planning should be performed to a greater extent to inform value estimations of the state's predecessors. We hypothesize realizing this intuition can improve learning efficiency, especially in sparse reward settings. This is because what is learned would be propagated faster through backward planning at the right time when the value estimation of a state becomes accurate enough.

To know when the estimation is accurate enough, we propose using the simple quantity of the learning error which requires no extra computation to obtain. The learning error has a positive relationship with epistemic uncertainty, which is induced during the learning process. As the error goes down, an agent should be more certain about the value estimation of that state. In other words, simply based on the learning error, we can control how much backward planning to perform. We show how to apply such intuition in both the tabular and function approximation settings.

3.1 Tabular Setting

In the tabular setting, using Q-learning as an example, a simple extension can be done by having a maximum error (ME) table of the same size as the Q-value table to store the maximum error of each state-action pair. With the maximum error known, we have a reference point to the assess learning progress for each pair. To translate such information into resource allocation decisions in terms of how much to roll out forward and backward, we use the maximum error as a normalizing factor to obtain a value between 0 and 1. The allocation equations are as follows:

$$n\_forward = (e_t / ME(S_t, A_t)) \times max\_plan\_steps, \quad (4)$$

$$n\_backward = max\_plan\_steps - n\_forward, \quad (5)$$

where max_plan_steps is the number of planning steps an agent can take, $n\_forward$ and $n\_backward$ are the number of forward rollout steps and the number of backward rollout steps, e_t is the last temporal difference error of the state-action pair, and $ME(S_t, A_t)$ refers to the maximum error of the pair obtained from the maximum error table. We can see that as the error goes down, fewer resources will be allocated to perform forward rollouts with more resources left for backward rollouts to propagate the learned values backward. We hypothesize such a way of allocation would improve learning efficiency.

3.2 Function Approximation Setting

Our simple extension in the tabular setting is not scalable in the function approximation setting, especially in environments with large and continuous state spaces. Specifically, three problems arise, namely the memory issue of the ME table in continuous state spaces, the unreliability of the error measure under the function approximation setting, and the poor quality of learned world models. We propose three techniques to handle these issues

State Discretization: Having a ME table is intractable and inefficient in continuous state spaces. To handle this issue, we propose the use of state discretization to aggregate similar states together. This is done by binning each dimension of the state features separately, denoted as a function $discretize$. The unique combination of bins across all state dimensions would be one entry in the ME table. This method is highly scalable as the number of state dimensions grows linearly with the number of bins (Ghiassian et al. 2020). Hence, when we encounter a large and continuous state space, we no longer need a huge or possibly infinite-sized table but a reasonably sized one after state discretization. With state discretization, the planning budget allocation equation is as follows:

$$\begin{aligned} n\_forward \ &= (e_t / ME(discretize(S_t, A_t))) \\ &\times max\_plan\_steps. \end{aligned} \tag{6}$$

Here, we have an additional $discretize$ function that discretizes the state and returns the corresponding index to access the maximum error value for a subset of states in the ME table.

Exponential Moving Error: Another issue comes from the unreliability of using the learning error as a pseudo-measure of epistemic uncertainty. In the function approximation setting, learning no longer strictly linearly reduces error. An improvement in the value estimation of a state-action pair may worsen the estimation of another pair. Additionally, in some unseen or rarely seen states, or some catastrophic states to the value function, the error values can become exceptionally large. As a result, the relationship between the error and epistemic uncertainty is no longer as linear as we may expect in the function approximation setting. To overcome this, we propose the use of exponential moving error in the replacement of maximum error. By doing so, the error measure used for normalization is less likely to be skewed by large values and is more adaptive to recent learning progress. The update of ME table entries is as follows:

$$\begin{aligned} &ME(discretize(S_t, A_t)) = \\ &\beta \times ME(discretize(S_t, A_t)) + (1 - \beta) \times e_t, \end{aligned} \tag{7}$$

where β is the decaying constant, a hyperparameter to be tuned.

An Ensemeble of World Models: The last issue is the poor quality of world models. As we move towards harder problems beyond the tabular setting, we can no longer have perfect world models. In simple environments like tabular GridWorld, perfect world models can be created, given that we have a deterministic environment with well-defined action space. On the other hand, in harder problems like physics simulators and robotics, it is almost impossible to have a perfect world model, letting alone two perfect world models for forward and backward rollouts. The issue is significantly worse when such a world model is learned. When we use an inaccurate model to perform rollouts, the longer a rollout, the more inaccurate the predictions, caused by the issue of compounding errors (Asadi, Misra, and Littman 2018). To alleviate this issue, we propose the use of an ensemble of world models. The ensemble approach has been a commonly used approach in machine learning. The high-level idea is to improve prediction accuracy by training a set of models that differ by factors like different random initialization to achieve better performance. Osband et al. (2016) proposed the approach of learning an ensemble of value functions that are initialized differently. As learning progresses, the difference in predictions across these functions serves as an indicator of how certain the agent is with the predictions. Taking inspiration from this idea, we propose having an ensemble of learned world models. Their disagreements in the generated rollouts can serve as an indicator of how certain the models are with their predictions. If it is highly uncertain, the agent drops those transitions and does not learn from them. As a result, the agent can avoid learning from data that are highly erroneous with the potential of negatively affecting the learning progress. Specifically, we use the standard deviation of predictions across all the world models as the numerical measure. If the measure is greater than a threshold κ, the rollout is dropped to avoid the compounding of errors. We denote this threshold as κ. When the rollout is not dropped, we use the averaged prediction across the world models as the generated data. More options other than simple averaging like weighted averaging can be further explored.

By combining all these three techniques, we can extend our intuition to the function approximation setting. We will showcase the results in the experimental results section.

4 Experimental Setup

We conduct extensive experiments on the GridWorld environment for the tabular setting and the Cartpole environment for the linear function approximation setting.

For the GridWorld environment, an agent attempts to reach the goal state at the upper-right corner, starting from the lower-left corner. The agent only receives a reward of +1 when reaching the goal state. The state representation is the coordinate of the agent on the grid. GridWorld of various sizes are used to vary reward sparsity: 5x5, 10x10, 20x20, 40x40 and 80x80. For model-based agents, we used a perfect forward and backward world model with a fixed planning budget of 20 steps. To look at the effect of rollout mechanisms, we examine three types of model-based reinforcement learning agents, namely one that only performs forward rollouts (Dyna_Q_Forward), one that only performs backward rollouts (Dyna_Q_Backward) and one that dynamically performs forward and backward rollouts based on the epistemic uncertainty as discussed above (Dyna_Q_Forward_Backward).

For the Cartpole environment, a pole is attached by an unactuated joint to a cart, moving along a frictionless track. An episode starts with the pole upright and the goal of the agent is to prevent the pole from falling by increasing or decreasing the cart's velocity along the horizontal direction. The agent receives a reward of +1 as long as the pole is up and a terminating reward of -1 if the pole falls. The state representation has four features, namely the cart's position, the cart's velocity, the pole's angle, and the pole's velocity

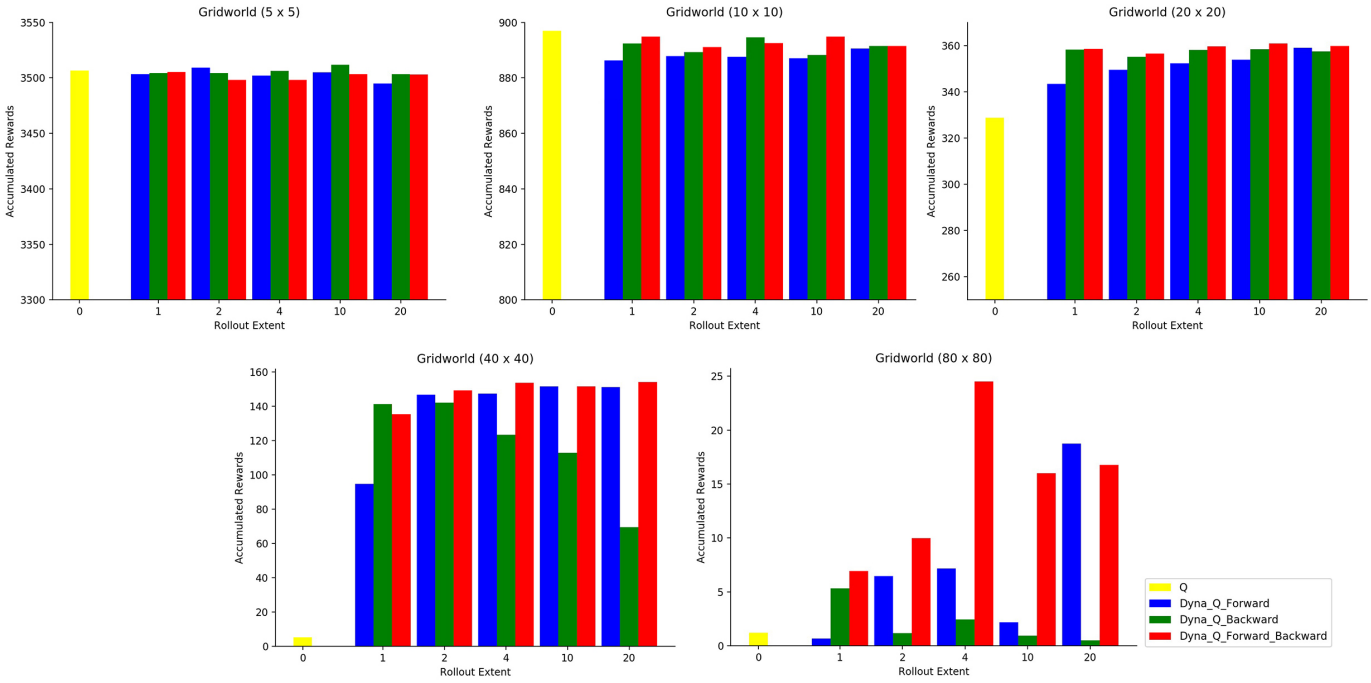


Figure 3: Performance of forward, backward and bidirectional rollouts for GridWorld environments

at the tip. We used the implementation provided by OpenAI (Brockman et al. 2016). To acquire a better feature representation with a higher number of features, we pretrain a feature extractor using Deep Q-learning (Mnih et al. 2015) and adopt the output of the last layer as the inputs to the learning algorithm. We assume imperfect world models by pretraining fully connected neural networks with a fixed planning budget of 20 steps. For the ensemble method, we use an ensemble of six world models for forward models and backward models, respectively. Details on the pretraining of the world models and feature extractor are included in the appendix (7.1 and 7.2) . To assess the performance of our proposed method, we compare with multiple Dyna-Q variants. We consider two algorithms that use only-forward rollouts (Dyna_Q_F) and only-backward rollouts (Dyna_Q_B), respectively, and two other algorithms with the same respective settings but with an ensemble of world model (Dyna_Q_F_DE and Dyna_Q_B_DE). Our proposed method is denoted as Dyna_Q_FB_DE, which uses an ensemble of world model and performs bidirectional rollouts dynamically.

For both sets of experiments, we use model-free Q-learning as our baseline. Parameters sweeps are done on the learning rate, β and κ (see appendix 7.3). Results reported are averaged over 10 runs of different random seeds.

5 Results

5.1 Tabular Setting

In Figure 3, we compare the performance of forward-only (blue), backward-only (green), and bidirectional rollouts (red) in the GridWorld environments of various sizes. To begin with, as problem difficulty increases, we can see model-based agents outperform the baseline Q-learning agent (yellow). This is obvious and expected as model-based agents have additional learning experiences received from their corresponding world models. As the GridWorld size increases, the model-free agent fails to acquire rewards due to the high reward sparsity.

Comparing the two agents that only perform forward and backward rollouts, we can see that as the problem difficulty increases, the agent with only forward rollouts outperforms the agent with only backward rollouts. One possible explanation is the relatively lower capability of exploration using backward rollouts, which prompts the question of whether backward rollouts can be applied conditionally in combination with forward rollouts to achieve even more superior results.

Among the three types of model-based agents, our proposed method with bidirectional rollouts has the best overall performance across GridWorld sizes. More importantly, our proposed method has consistently better performance in GridWorld sizes with larger reward sparsity (20x20, 40x40, and 80x80). This provides supporting evidence of the significant impact of directionality on performance. Specifically, learning efficiency can be improved by utilizing forward and backward planning dynamically based on error-based epistemic uncertainty.

Additionally, looking at the planning shape, our tabular results are in agreement with the conclusions made in Holland, Talvitie, and Bowling (2018) under the nonlinear function approximation setting. We can see that medium-length rollouts are much better than one-step rollouts.

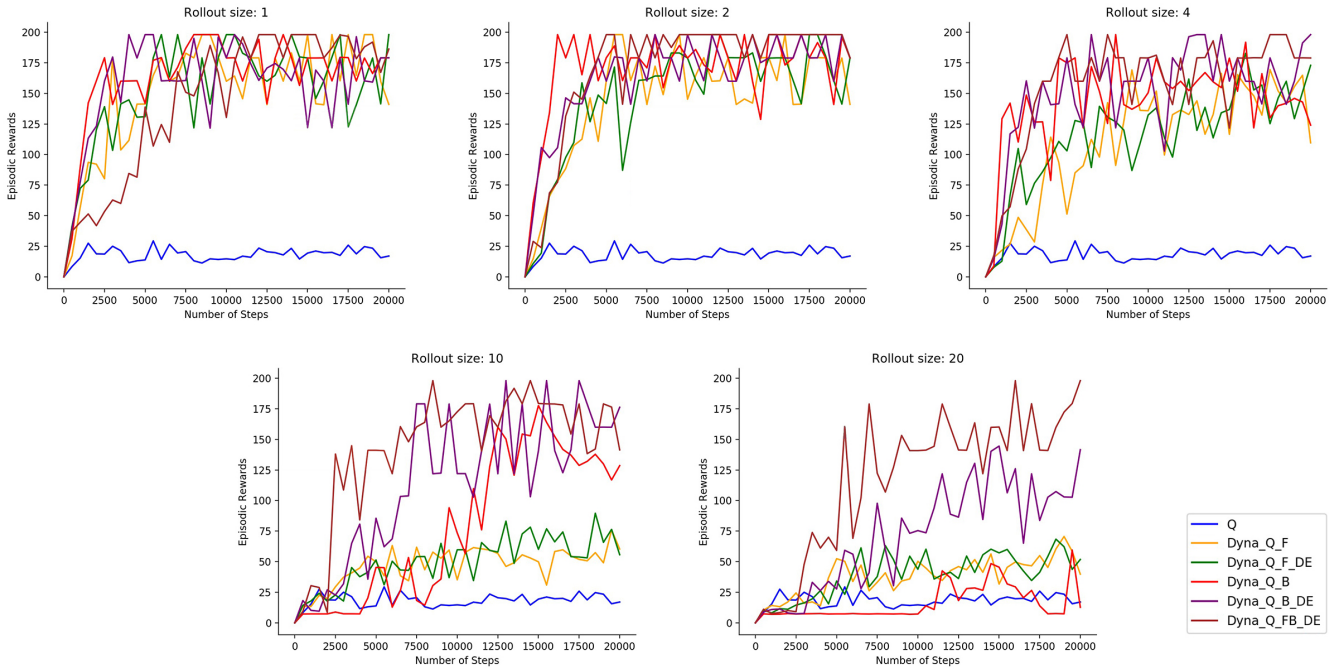


Figure 4: Learning curves of forward, backward and bidirectional rollouts for Cartpole environment

5.2 Function Approximation Setting

In Figure 4, we show the performance of all the mentioned agents across different planning shapes. Being consistent with the results in the GridWorld Experiments, medium-length planning shapes have the best performance for the model-based agents.

We can see that our proposed approach (Dyna_Q_FB_DE) has the best overall results. As the rollout length increases, we observe how an ensemble of world models alleviates the problem of compounding errors by dropping harmful transitions. In other words, the approach is robust when we have imperfect world models and need longer rollouts. The same effect can also be observed in agents that only perform forward or backward rollouts exclusively but with an ensemble of world models (Dyna_Q_F_DE and Dyna_Q_B_DE). By comparing with these two agents, we can see how bidirectional rollouts based on epistemic uncertainty helps with improving learning efficiency. These results in the function approximation setting provide further support to our hypothesis on the differing roles of forward planning and backward planning, and how careful and state-dependent allocation can improve performance.

6 Conclusion and Future Work

In this work, we investigate how the planning shape and directionality of the rollout mechanism affect the performance of a model-based reinforcement learning agent. We hypothesize that if an agent can perform forward and backward plannings dynamically, it can achieve better performance and learning efficiency. We postulate that forward and backward plannings have complementary roles of exploration and value propagation respectively. Once the quality of value estimations improves, more backward planning should be performed to propagate the values backward to previous states for quicker credit assignment. By doing so, a better value function can be obtained at a faster pace.

Based on our hypotheses, we propose an online method to perform bidirectional rollouts using error-based epistemic uncertainty, as a numerical indicator for the quality of value estimations. Specifically, we keep track of the maximum learning error (or exponential moving error in the function approximation case) of each state-action pair to assess learning progress, which is then used to allocate the planning budget for forward and backward plannings. To further extend our method to large and continuous state space, we apply state discretization, an efficient method to overcome the need of keeping track of all possible state-action pairs. Additionally, to counter the problem of error compounding in long rollouts in imperfect world models, we propose using an ensemble of world models to drop harmful and erroneous rollouts from learning.

By conducting experiments in both the tabular and linear function approximation settings, we reaffirm the benefits of medium-length rollouts when compared with one-step rollouts for the same amount of planning steps. We also demonstrate how our proposed method of bidirectional rollouts can improve performance and learning efficiency when compared with our baseline Dyna agents of the same planning budget, particularly in the sparse reward settings. These provide supporting evidence to the hypothesized roles of forward and backward plannings.

For future work, we plan to conduct a larger scale of study on more complicated problems to further assess our hypotheses made in this work. We also plan to develop better and more efficient methods in performing bidirectional rollouts. For instance, we can look at more principled approaches like Gaussian processes to model uncertainty, which can also be used to assess a world model's uncertainty towards its predictions.

References

Asadi, K.; Misra, D.; Kim, S.; and Littman, M. L. 2019. Combating the compounding-error problem with a multi-step model. *arXiv preprint arXiv:1905.13320*.

Asadi, K.; Misra, D.; and Littman, M. L. 2018. Lipschitz continuity in model-based reinforcement learning. *arXiv preprint arXiv:1804.07193*.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Edwards, A. D.; Downs, L.; and Davidson, J. C. 2018. Forward-backward reinforcement learning. *arXiv preprint arXiv:1803.10227*.

Ghiassian, S.; Rafiee, B.; Lo, Y. L.; and White, A. 2020. Improving performance in reinforcement learning by breaking generalization in neural networks. *arXiv preprint arXiv:2003.07417*.

Gu, S.; Holly, E.; Lillicrap, T.; and Levine, S. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, 3389–3396. IEEE.

Holland, G. Z.; Talvitie, E. J.; and Bowling, M. 2018. The effect of planning shape on dyna-style planning in high-dimensional state spaces. *arXiv preprint arXiv:1806.01825*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature* 518(7540):529–533.

Moore, A. W., and Atkeson, C. G. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning* 13(1):103–130.

Osband, I.; Blundell, C.; Pritzel, A.; and Van Roy, B. 2016. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, 4026–4034.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S. 1991. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin* 2(4):160–163.

Talvitie, E. 2014. Model regularization for stable sample rollouts. In *UAI*, 780–789.

Venkatraman, A.; Hebert, M.; and Bagnell, J. A. 2015. Improving multi-step prediction of learned time series models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Volodymyr, M.; Koray, K.; David, S.; Andrei, A. R.; and Joel, V. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.

White, M. 2017. Unifying task specification in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML-17)-Volume 70*, 3742–3750.

7 Appendix

7.1 World Models Pretraining

To pretrain the forward and backward world models, we first collect one million transitions by following a random policy. Then, we perform supervised training using fully-connected neural networks with 1 hidden layer by minimizing the mean-squared errors. The parameters details are given:

Batch Size	256
Epoch	50000
Learning Rate	0.0001
Optimizer	Adam with default parameters
Number of hidden units	512

Table 1: Hyperparameter setting for world models pretraining

For an ensemble of world models, we train each of them with different random seeds.

7.2 Feature Extractor Pretraining

We pretrain a feature extractor using a fully-connected neural network in order to obtain a better feature representation for linear function approximation, used for the experiments with the Cartpole environment. To do so, we pretrain a Deep Q learning (Mnih et al. 2015) agent and use the output of the last layer as features. The parameters details are shown in table 2.

Batch Size	32
Epoch	500000
Learning Rate	0.0001
Optimizer	Adam with default parameters
Number of hidden units	32
Experience replay buffer size	10000
Target network update rate	100

Table 2: Hyperparameter setting for feature extractor pretraining

7.3 Hyperparameter Sweep

Table 3 presents the values of hyperparameters we sweep over to produce the experimental results. We used the same exploration policy for all the agents in this work.

GridWorld	
Number of steps	100000
Learning rate	$2^{-i}, i \in 1, 2, 3, 4, 5, 6, 7, 8$
Rollout sizes	1, 2, 4, 10, 20
World sizes	5, 10, 20, 40, 80
Discount rate	0.999
Planning buffer size	1000
Cartpole	
Number of steps	20000
Feature size	32
Learning rate	$10^{-i}, i \in 1, 2, 3, 4$
Discretization bin size	6, 8, 10, 15, 20
beta	0.99
kappa	$2^{-i}, i \in 4, 5, 6, 7, 8$
Discount rate	0.999
Planning buffer size	10000
ϵ**-greedy policy**	
ϵ starting value	1.0
ϵ minimum value	0.1
ϵ decay	0.9995

Table 3: Hyperparameter sweep for the experiments

PBCS: Efficient Exploration and Exploitation Using a Synergy between Reinforcement Learning and Motion Planning

Guillaume Matheron[1], **Nicolas Perrin**[1], **Olivier Sigaud**[1]
[1]Sorbonne Université, CNRS,
Institut des Systèmes Intelligents et de Robotique, ISIR,
F-75005 Paris, France
guillaume_pub [at] matheron.eu

Abstract

The exploration-exploitation trade-off is at the heart of reinforcement learning (RL). However, most continuous control benchmarks used in recent RL research only require local exploration. This led to the development of algorithms that have basic exploration capabilities, and behave poorly in benchmarks that require more versatile exploration. For instance, as demonstrated in our empirical study, state-of-the-art RL algorithms such as DDPG and TD3 are unable to steer a point mass in even small 2D mazes. In this paper, we propose a new algorithm called "Plan, Backplay, Chain Skills" (PBCS) that combines motion planning and reinforcement learning to solve hard exploration environments. In a first phase, a motion planning algorithm is used to find a single good trajectory, then an RL algorithm is trained using a curriculum derived from the trajectory, by combining a variant of the Backplay algorithm and skill chaining. We show that this method outperforms state-of-the-art RL algorithms in 2D maze environments of various sizes, and is able to improve on the trajectory obtained by the motion planning phase.

Introduction

Reinforcement Learning (RL) algorithms have been used successfully to optimize policies for both discrete and continuous control problems with high dimensionality (Mnih et al. 2013; Lillicrap et al. 2015), but fall short when trying to solve difficult exploration problems (van Hasselt et al. 2018; Achiam, Knight, and Abbeel 2019; Schaul et al. 2015). On the other hand, motion planning (MP) algorithms such as RRT (Lavalle 1998) are able to efficiently explore in large cluttered environments but, instead of trained policies, they output trajectories that cannot be used directly for closed loop control.

In this paper, we consider environments that present a hard exploration problem with a sparse reward. In this context, a *good trajectory* is one that reaches a state with a positive reward, and we say that an environment is *solved* when a controller is able to reliably reach a rewarded state. We illustrate our approach with 2D continuous action mazes as they facilitate the visual examination of the results, but we believe that this approach can be beneficial to many robotics problems.

If one wants to obtain closed loop controllers for hard exploration problems, a simple approach is to first use an MP algorithm to find a single good trajectory τ, then optimize and robustify it using RL. However, using τ as a stepping stone for an RL algorithm is not straightforward. In this article, we propose PBCS, an approach that fits the framework of Go-Explore (Ecoffet et al. 2019), and is based on the Backplay algorithm (Resnick et al. 2018) and skill chaining (Konidaris and Barto 2009; Konidaris et al. 2010). We show that this approach greatly outperforms both DDPG (Lillicrap et al. 2015) and TD3 (Fujimoto, Hoof, and Meger 2018) on continuous control problems in 2D mazes, as well as approaches that use Backplay but no skill chaining.

PBCS has two successive phases. First, the environment is explored until a single good trajectory is found. Then this trajectory is used to create a curriculum for training DDPG. More precisely, PBCS progressively increases the difficulty through a backplay process which gradually moves the starting point of the environment backwards along the trajectory resulting from exploration. Unfortunately, this process has its own issues, and DDPG becomes unstable in long training sessions. Calling upon a skill chaining approach, we use the fact that even if Backplay eventually fails, it is still able to solve some subset of the problem. Therefore, a partial policy is saved, and the reminder of the problem is solved recursively until the full environment can be solved reliably.

In this article, we contribute an extension of the Go-Explore framework to continuous control environments, a new way to combine a variant of the Backplay algorithm with skill chaining, and a new state-space exploration algorithm.

1 Related Work

Many works have tried to incorporate better exploration mechanisms in RL, with various approaches.

Encouraging exploration of novel states The authors of (Tang et al. 2016) use a count-based method to penalize states that have already been visited, while the method proposed by (Benureau and Oudeyer 2016) reuses actions that have provided diverse results in the past. Some methods try to choose policies that are both efficient and novel (Pugh

et al. 2015; Cully and Demiris 2017; Pugh, Soros, and Stanley 2016; Erickson and LaValle 2009), while some use novelty as the only target, entirely removing the need for rewards (Eysenbach et al. 2018; Knepper and Mason 2009). The authors of (Stadie, Levine, and Abbeel 2015; Burda et al. 2018; Pathak et al. 2017) train a forward model and use the unexpectedness of the environment step as a proxy for novelty, which is encouraged through reward shaping. Some approaches try to either estimate the uncertainty of value estimates (Osband et al. 2016), or learn bounds on the value function (Ciosek et al. 2019). All these solutions try to integrate an exploration component within RL algorithms, while our approach separates exploration and exploitation into two successive phases, as in (Colas, Sigaud, and Oudeyer 2018).

Using additional information about the environment Usually in RL, the agent can only learn about the environment through interactions. However, when additional information about the task at hand is provided, other methods are available. This information can take the form of expert demonstrations (Salimans and Chen 2018; Hosu and Rebedea 2016; Resnick et al. 2018; Konidaris et al. 2010; Fournier et al. 2019; Nair et al. 2018; Paine et al. 2019), or having access to a single rewarded state (Florensa et al. 2018). When a full representation of the environment is known, RL can still be valuable to handle the dynamics of the problem: PRM-RL (Faust et al. 2018) and RL-RRT (Chiang et al. 2019) use RL as reachability estimators during a motion planning process.

Building on the Go-Explore framework To our knowledge, the closest approach to ours is the Go-Explore (Ecoffet et al. 2019) framework, but in contrast to PBCS, Go-Explore is applied to discrete problems such as Atari benchmarks. In a first phase, a single valid trajectory is computed using an ad-hoc exploration algorithm. In a second phase, a learning from demonstration (LfD) algorithm is used to imitate and improve upon this trajectory. Go-Explore uses Backplay (Resnick et al. 2018; Salimans and Chen 2018) as the LfD algorithm, with Proximal Policy Optimization (PPO) (Schulman et al. 2017) as policy optimization method. Similar to Backplay, the authors of (Goyal et al. 2019) have proposed Recall Traces, a process in which a backtracking model is used to generate a collection of trajectories reaching the goal.

The authors of (Morere et al. 2020) present an approach that is similar to ours, and also fits the framework of Go-Explore. In phase 1, they use a guided variant of RRT, and in phase 2 they use a learning from demonstration algorithm based on TRPO. Similarly, PBCS follows the same two phases as Go-Explore, with major changes to both phases. In the first phase, our exploration process is adapted to continuous control environments by using a different binning method, and different criteria for choosing the state to reset to. In the second phase, a variant of Backplay is integrated with DDPG instead of PPO, and seamlessly integrated with a skill chaining strategy and reward shaping.

The Backplay algorithm in PBCS is a deterministic variant of the one proposed in (Resnick et al. 2018). In the original Backplay algorithm, the starting point of each policy is chosen randomly from a subset of the trajectory, but in our variant the starting point is deterministic: the last state of the trajectory is used until the performance of DDPG converges (more details are presented in Sect. 8), then the previous state is chosen, and so on until the full trajectory has been exploited.

Skill chaining The process of *skill chaining* was explored in different contexts by several research papers. The authors of (Konidaris and Barto 2009) present an algorithm that incrementally learns a set of skills using classifiers to identify changepoints, while the method proposed in (Konidaris et al. 2010) builds a skill tree from demonstration trajectories, and automatically detects changepoints using statistics on the value function. To our knowledge, our approach is the first to use Backplay to build a skill chain. We believe that it is more reliable and minimizes the number of changepoints because the position of changepoints is decided using data from the RL algorithm that trains the policies involved in each skill.

2 Background

Our work is an extension of the Go-Explore algorithm. In this section, we summarize the main concepts of our approach.

Reset-anywhere. Our work makes heavy use of the ability to reset an environment to any state. The use of this primitive is relatively uncommon in RL, because it is not always readily available, especially in real-world robotics problems. However, it can be invaluable to speed up exploration of large state spaces. It was used in the context of Atari games by (Hosu and Rebedea 2016), proposed in (Schulman et al. 2015) as VINE, and gained popularity with (Salimans and Chen 2018).

Sparse rewards. Most traditional RL benchmarks only require very local exploration, and have smooth rewards guiding them towards the right behavior. Thus sparse rewards problems are especially hard for RL algorithms: the agent has to discover without any external signal a long sequence of actions leading to the reward. Most methods that have been used to help with this issue require prior environment-specific knowledge (Riedmiller et al. 2018).

Maze environments. Lower-dimension environments such as cliff walk (Sutton and Barto 2018) are often used to demonstrate fundamental properties of RL algorithms, and testing in these environments occasionally reveals fundamental flaws (Matheron, Perrin, and Sigaud 2019). We deliberately chose to test our approach on 2D maze environments because they are hard exploration problems, and because reward shaping behaves very poorly in such environments, creating many local optima. Our results in

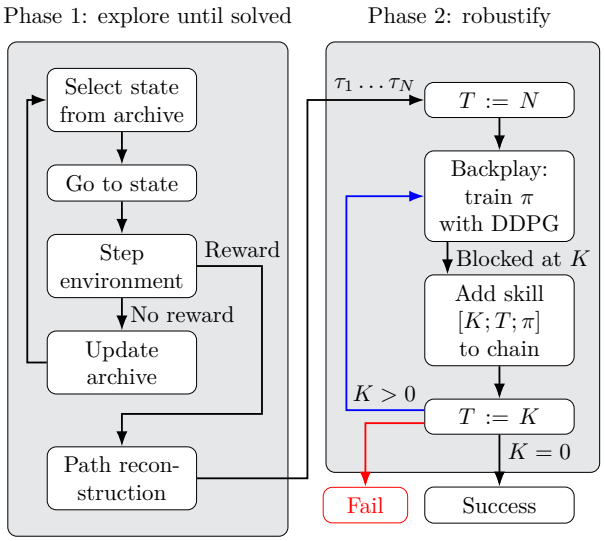


Figure 1: Overview of PBCS. The red path is only used when testing the algorithm without skill chaining, otherwise the blue path is used.

Sect. 4 show that state-of-the-art algorithms such as DDPG and TD3 fail to solve even very simple mazes.

DDPG. Deep Deterministic Policy Gradient (DDPG) is a continuous action actor-critic algorithm using a deterministic actor that performs well on many control tasks (Lillicrap et al. 2015). However, DDPG suffers from several sources of instability. Our maze environments fit the analysis made by the authors of (Penedones et al. 2018), according to whom the critic approximator may "leak" Q-value across walls of discontinuous environments. With a slightly different approach, (Fujimoto, Meger, and Precup 2018) suggests that extrapolation error may cause DDPG to over-estimate the value of states that have never been visited or are unreachable, causing instability. More generally, the authors of (Sutton and Barto 2018) formalize the concept of "deadly triad", according to which algorithms that combine function approximation, bootstrapping updates and off-policy are prone to diverge. Even if the deadly triad is generally studied in the context of the DQN algorithm (Mnih et al. 2013), these studies could also apply to DDPG. Finally, the authors of (Matheron, Perrin, and Sigaud 2019) show that DDPG can fail even in trivial environments, when the reward is not found quickly enough by the built-in exploration of DDPG.

3 Methods

Figure 1 describes PBCS. The algorithm is split in two successive phases, mirroring the Go-Explore framework. In a first phase, the environment is incrementally explored until a single rewarded state is found. In a second phase, a single trajectory provides a list of starting points, that are used to train DDPG on increasingly difficult portions of the full environment. Each time the problem becomes too difficult and DDPG starts to fail, training stops, and the trained agent is recorded as a local skill. Training then resumes for the next skill, with a new target state. This loop generates a set of skills that can then be chained together to create a controller that reaches the target reliably.

Notations

State neighborhood. For any state $\tau_i \in S$, and $\epsilon > 0$, we define $B_\epsilon(\tau_i)$ as the closed ball of radius ϵ centered around τ_i. Formally, this is the set $\{s \in S \mid d(s, \tau_i) \leq \epsilon\}$ where d is the L2 distance.

Skill chaining. Skill chaining consists in splitting a complex task into simpler sub-tasks that are each governed by a different policy. Complex tasks can then be solved by executing each policy sequentially.

Formally, each task T_i has an activation condition $A_i \subset S$, and a policy $\pi_i : S \to A$. A *task chain* is a list of tasks $T_0 \dots T_n$, which can be executed sequentially: the actor uses π_0 until the state of the system reaches a state $s \in A_1$, then it uses π_1, and so on until the end of the episode (which can be triggered by reaching either a terminal state or a predetermined maximum number of steps).

3.1 Phase 1: Explore until Solved

In phase 1, PBCS explores to find a single path that obtains a non-zero reward in the environment. This exploration phase is summarized in this section, and implementation details are available in Appendix S1. An archive keeps track of all the visited states. In this archive, states s are grouped in square state-space bins. A state-counter c_s is attached to each state, and a bin-counter c_b is attached to each bin. All counters are initialized to 0.

The algorithm proceeds in 5 steps, as depicted in Figure 1:

1. **Select state from archive.** To select a state, the non-empty bin with the lowest counter is first selected, then from all the states in this bin, the state with the lowest counter is selected. Both the bin and state counters are then incremented.
2. **Go to state.** The environment is reset to the selected state. This assumes the existence of a "reset-anywhere" primitive, which can be made available in simulated environments.
3. **Step environment.** A single environment step is performed, with a random action.
4. **Update archive.** The newly-reached state is added to the archive if not already present.
5. **Termination of phase 1.** As soon as the reward is reached, the archive is used to reconstruct the sequence of states that led the agent from its initial state to the reward. This sequence $\tau_0 \dots \tau_N$ is passed on to phase 2.

This process can be seen as a random walk with a constraint on the maximum distance between two states: in the beginning, a single trajectory is explored until it reaches a state that is too close to an already-visited state. When this happens, a random visited state is selected as the starting point of a new random walk. Another interpretation of this

process is the construction of a set of states with uniform spatial density. Under this view, the number of states in each cell is used as a proxy for the spatial density of the distribution.

3.2 Phase 2: Robustify

Phase 2 of PBCS learns a controller from the trajectory obtained in phase 1.

Algorithm 1: Phase 2 of PBCS

Input : $\tau_0 \dots \tau_N$ the output of phase 1
Output: $\pi_0 \dots \pi_n$ a chain of policies with activation sets $A_0 \dots A_n$

1 $T = N$
2 $n = 0$
3 **while** $T > 0$ **do**
4 $\quad \pi_n, T = \texttt{Backplay}(\tau_0 \dots \tau_T)$
5 $\quad A_n = B_\epsilon(\tau_T)$
6 $\quad n = n + 1$
7 **end**
8 Reverse lists $\pi_0 \dots \pi_n$ and $A_0 \dots A_n$

Skill Chaining. Algorithm 1 presents the skill chaining process. It uses the `Backplay` function, that takes as input a trajectory $\tau_0 \dots \tau_T$, and returns a policy π and an index $K < T$ such that running policy π repeatedly on a state from $B_\epsilon(\tau_K)$ always leads to a state in $B_\epsilon(\tau_T)$. The main loop builds a chain of skills that roughly follows trajectory τ, but is able to improve upon it. Specifically, activation sets A_n are centered around points of τ but policies π_n are constructed using a generic RL algorithm that optimizes the path between two activation sets. The list of skills is then reversed, because it was constructed backwards.

Backplay. The Backplay algorithm was originally proposed in (Resnick et al. 2018). More details on the differences between this original algorithm and our variant are available in sections 1 and S6.

The `Backplay` function (Algorithm 2) takes as input a section $\tau_0 \dots \tau_T$ of the trajectory obtained in phase 1, and returns a (K, π) pair where K is an index on trajectory τ, and π is a policy trained to reliably attain $B_\epsilon(\tau_T)$ from $B_\epsilon(\tau_K)$. The policy π is trained using DDPG to reach $B_\epsilon(\tau_T)$ from starting point $B_\epsilon(\tau_K)$ [1] , where K is initialized to $T-1$, and gradually decremented in the main loop.

At each iteration, the algorithm evaluates the feasibility of a skill with target $B_\epsilon(\tau_T)$, policy π and activation set $B_\epsilon(\tau_K)$. If the measured performance is 100% without any training (line 5), the current skill is saved and the starting point is decremented. Otherwise, a training loop is executed until performance stabilizes (line 8). This is performed by running Algorithm 3 repeatedly until no improvement over the maximum performance is observed α times in a row. We ran our experiments with $\alpha = 10$, and a more in-depth discussion of hyperparameters is available in Appendix S2 .

[1] More details on why the starting point needs to be $B_\epsilon(\tau_K)$ instead of τ_K are available in Appendix S6

Then the performance of the skill is measured again (line 9), and three cases are handled:

- **The skill is always successful (line 10).** The current skill is saved and the index of the starting point is decremented.
- **The skill is never successful (line 13).** The last successful skill is returned.
- **The skill is sometimes successful.** The current skill is not saved, and the index of the starting point is decremented. In our maze environment, this happens when $B_\epsilon(\tau_K)$ overlaps a wall: in this case some states of $B_\epsilon(\tau_K)$ cannot reach the target no matter the policy.

Algorithm 2: The Backplay algorithm

Input : $(\tau_0 \dots \tau_T)$ a state-space trajectory
Output: π_s a trained policy
K_s the index of the starting point of the policy

1 $K = T - 1$
2 Initialize a DDPG architecture with policy π
3 **while** $K > 0$ **do**
4 $\quad$ Test performance of π between $B_\epsilon(\tau_K)$ and $B_\epsilon(\tau_T)$ over β episodes
5 $\quad$ **if** *performance* $= 100\%$ **then**
6 $\quad\quad \pi_s = \pi$, $K_s = K$
7 $\quad$ **else**
8 $\quad\quad$ Run `Train` (Algorithm 3) repeatedly until performance stabilizes.
9 $\quad\quad$ Test performance of π between $B_\epsilon(\tau_K)$ and $B_\epsilon(\tau_T)$ over β episodes
10 $\quad\quad$ **if** *performance* $= 100\%$ **then**
11 $\quad\quad\quad \pi_s = \pi$, $K_s = K$
12 $\quad\quad$ **end**
13 $\quad\quad$ **if** *performance* $= 0\%$ *and* K_s *exists* **then**
14 $\quad\quad\quad$ **return** (K_s, π_s)
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 $\quad K = K - 1$
18 **end**
19 **return** (K_s, π_s)

Reward Shaping. With reward shaping, we bypass the reward function of the environment, and train DDPG to reach any state τ_T. We chose to use the method proposed by (Ng, Harada, and Russell 1999): we define a potential function in Equation (1a), where $d(s, A_i)$ is the L2 distance between s and the center of A_i. We then define our shaped reward in Equation (1b).

$$\Phi(s) = \frac{1}{d(s, A_i)} \tag{1a}$$

$$R_{\text{shaped}}(s, a, s') = \begin{cases} 10 & \text{if } s \in A_i \\ \Phi(s') - \Phi(s) & \text{otherwise.} \end{cases} \tag{1b}$$

Algorithm 3 shows how this reward function is used in place of the environment reward. This training function runs

Algorithm 3: Training process with reward shaping

Input : τ_K the source state
τ_T the target state
Output: The average performance p

n = 0
for $i = 1 \dots \beta$ **do**
  $s \sim B_\epsilon(\tau_K)$
  for $j = 1 \dots$ *max_steps* **do**
    $a = \pi(s)$ + random noise
    $s' = \text{step}(s, a)$
    $r = \begin{cases} 10 & d(s', \tau_T) \leq \epsilon \\ \frac{1}{d(s',\tau_T)} - \frac{1}{d(s,\tau_T)} & \text{otherwise} \end{cases}$
    DDPG.train(s, a, s', r)
    $s = s'$
    if $d(s', \tau_T) \leq \epsilon$ **then**
      n = n + 1
      break
    end
  end
end
$p = \frac{n}{\beta}$

β episodes of up to `max_steps` steps each, and returns the fraction of episodes that were able to reach the reward. β is a hyperparameter that we set to 50 for our test, and more details on this choice are available in Appendix S2 .

Importantly, reaching a performance of 100% is not always possible, even with long training sessions, because the starting point is selected in $B_\epsilon(\tau_K)$, and some of these states may be inside obstacles for instance.

4 Experimental Results

We perform experiments in continuous maze environments of various sizes. For a maze of size N, the state-space is the position of a point mass in $[0, N]^2$ and the action describes the speed of the point mass, in $[-0.1, 0.1]^2$. Therefore, the step function is simply $s' = s + a$, unless the $[s, s']$ segment intersects a wall. The only reward is -1 when hitting a wall and 1 when the target area is reached. A more formal definition of the environment is available in Appendix S4.

Our results are presented in Table 1. We first tested standard RL algorithms (DDPG and TD3), then PBCS, but without skill chaining (this was done by replacing the blue branch with the red branch in Figure 1). When the full algorithm would add a new skill to the skill chain and continue training, this variant stops and fails. These results are presented in column "PBCS without skill chaining". Finally, the full version of PBCS with skill chaining is able to solve complex mazes up to 15×15 cells, by chaining several intermediate skills.

5 Discussion of Results

As expected, standard RL algorithms (DDPG and TD3) were unable to solve all but the simplest mazes. These algorithms have no mechanism for state-space exploration other than uniform noise added to their policies during rollouts. Therefore, in the best-case scenario they perform a random walk and, in the worst-case scenario, their actors may actively hinder exploration.

More surprisingly, PBCS without skill chaining is still unable to reliably [2] solve mazes larger than 2×2. Although phase 1 always succeeds in finding a feasible trajectory τ, the robustification phase fails relatively early. We attribute these failures to well-known limitations of DDPG exposed in Sect. 2. We found that the success rate of PBCS without skill chaining was very dependent on the discount rate γ, which we discuss in Appendix S3.

The full version of PBCS with skill chaining is able to overcome these issues by limiting the length of training sessions of DDPG, and is able to solve complex mazes up to 7×7, by chaining several intermediate skills.

6 Conclusion

The authors of Go-Explore identified state-space exploration as a fundamental difficulty on two Atari benchmarks. We believe that this difficulty is also present in many continuous control problems, especially in high-dimension environments. We have shown that the PBCS algorithm can solve these hard exploration, continuous control environments by combining a motion planning process with reinforcement learning and skill chaining. Further developments should focus on testing these hybrid approaches on higher dimensional environments that present difficult exploration challenges together with difficult local control, such as the Ant-Maze MuJoCo benchmark (Tassa and et. al. 2018), and developing methods that use heuristics suited to continuous control in the exploration process, such as Quality-Diversity approaches (Pugh, Soros, and Stanley 2016).

7 Acknowledgements

This work was partially supported by the French National Research Agency (ANR), Project ANR-18-CE33-0005 HUSKI.

References

Achiam, J.; Knight, E.; and Abbeel, P. 2019. Towards Characterizing Divergence in Deep Q-Learning. *arXiv:1903.08894*.

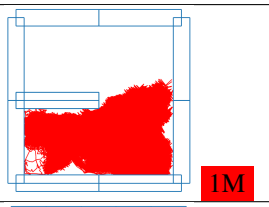
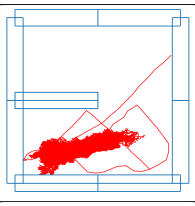
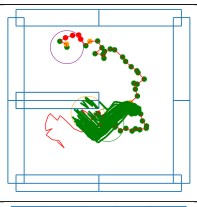
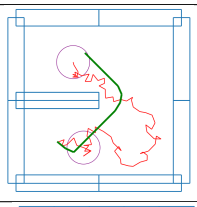
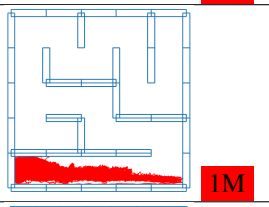
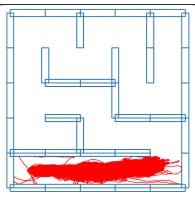
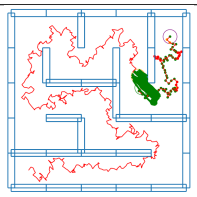
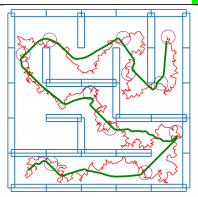
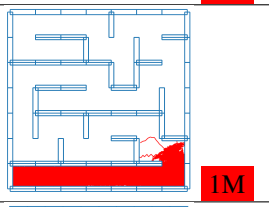
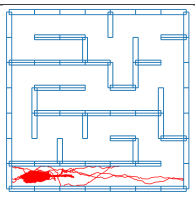
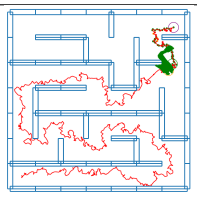
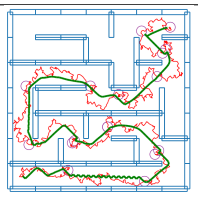
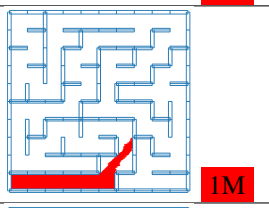
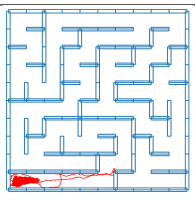
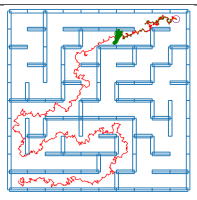
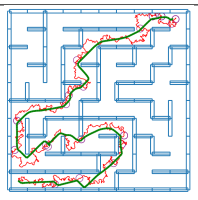
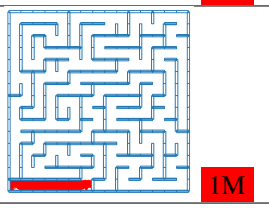
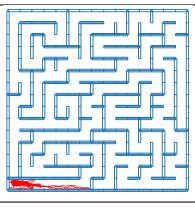
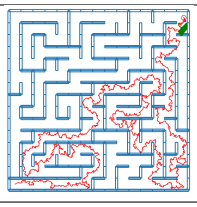
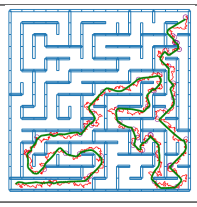
Benureau, F. C. Y., and Oudeyer, P.-Y. 2016. Behavioral Diversity Generation in Autonomous Exploration through Reuse of Past Experience. *Front. Robot. AI* 3.

Burda, Y.; Edwards, H.; Storkey, A.; and Klimov, O. 2018. Exploration by Random Network Distillation. *arXiv:1810.12894*.

Chiang, H.-T. L.; Hsu, J.; Fiser, M.; Tapia, L.; and Faust, A. 2019. RL-RRT: Kinodynamic Motion Planning via Learning Reachability Estimators from RL Policies. *arXiv:1907.04799*.

[2] We tested PBCS without skill chaining with different seeds on small mazes, these results are presented in Appendix S3

Table 1: Results of various algorithms on maze environments. For each test, the number of environment steps performed is displayed with a red background when the policy was not able to reach the target, and a green one when training was successful. In "Vanilla" experiments, the red paths represent the whole area explored by the RL algorithm. In "Backplay" experiments, the trajectory computed in phase 1 is displayed in red, and the "robustified" policy or policy chain is displayed in green. Activation sets A_i are displayed as purple circles. Enlarged images are presented in Fig. S2.

Vanilla		PBCS w/o skill chaining	PBCS
DDPG	TD3	DDPG	DDPG
 1M	 1M	 146k	 321k
 1M	 1M	 372k	 5M
 1M	 1M	 268k	 6M
 1M	 1M	 694k	 8M
 1M	 1M	 175k	 22M

Ciosek, K.; Vuong, Q.; Loftin, R.; and Hofmann, K. 2019. Better Exploration with Optimistic Actor-Critic. *arXiv:1910.12807*.

Colas, C.; Sigaud, O.; and Oudeyer, P.-Y. 2018. GEP-PG: Decoupling Exploration and Exploitation in Deep Reinforcement Learning Algorithms. *arXiv:1802.05054*.

Cully, A., and Demiris, Y. 2017. Quality and Diversity Optimization: A Unifying Modular Framework. *IEEE Transactions on Evolutionary Computation* 1–1.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2019. Go-Explore: a New Approach for Hard-Exploration Problems. *arXiv:1901.10995*.

Erickson, L. H., and LaValle, S. M. 2009. Survivability: Measuring and ensuring path diversity. In *2009 IEEE International Conference on Robotics and Automation*, 2068–2073.

Eysenbach, B.; Gupta, A.; Ibarz, J.; and Levine, S. 2018. Diversity is All You Need: Learning Skills without a Reward Function. *arXiv:1802.06070*.

Faust, A.; Ramirez, O.; Fiser, M.; Oslund, K.; Francis, A.; Davidson, J.; and Tapia, L. 2018. PRM-RL: Long-range Robotic Navigation Tasks by Combining Reinforcement Learning and Sampling-based Planning. *arXiv:1710.03937*.

Florensa, C.; Held, D.; Wulfmeier, M.; Zhang, M.; and Abbeel, P. 2018. Reverse Curriculum Generation for Reinforcement Learning. *arXiv:1707.05300*.

Fournier, P.; Sigaud, O.; Colas, C.; and Chetouani, M. 2019. CLIC: Curriculum Learning and Imitation for object Control in non-rewarding environments. *arXiv:1901.09720*.

Fujimoto, S.; Hoof, H. v.; and Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods. *ICML*.

Fujimoto, S.; Meger, D.; and Precup, D. 2018. Off-Policy Deep Reinforcement Learning without Exploration. *arXiv:1812.02900*.

Goyal, A.; Brakel, P.; Fedus, W.; Singhal, S.; Lillicrap, T.; Levine, S.; Larochelle, H.; and Bengio, Y. 2019. Recall Traces: Backtracking Models for Efficient Reinforcement Learning. *arXiv:1804.00379*.

Hosu, I.-A., and Rebedea, T. 2016. Playing Atari Games with Deep Reinforcement Learning and Human Checkpoint Replay. *arXiv:1607.05077*.

Knepper, R. A., and Mason, M. T. 2009. Path diversity is only part of the problem. In *2009 IEEE International Conference on Robotics and Automation*, 3224–3229.

Konidaris, G., and Barto, A. G. 2009. Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining. In Bengio, Y., and et. al., eds., *Advances in Neural Information Processing Systems 22*. 1015–1023.

Konidaris, G.; Kuindersma, S.; Grupen, R.; and Barto, A. G. 2010. Constructing Skill Trees for Reinforcement Learning Agents from Demonstration Trajectories. In Lafferty, J. D., and et. al., eds., *Advances in Neural Information Processing Systems 23*. 1162–1170.

Lavalle, S. M. 1998. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Iowa State University.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv:1509.02971*.

Matheron, G.; Perrin, N.; and Sigaud, O. 2019. The problem with DDPG: understanding failures in deterministic environments with sparse rewards. *arXiv:1911.11679*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602*.

Morere, P.; Francis, G.; Blau, T.; and Ramos, F. 2020. Reinforcement Learning with Probabilistically Complete Exploration. *arXiv:2001.06940*.

Nair, A.; McGrew, B.; Andrychowicz, M.; Zaremba, W.; and Abbeel, P. 2018. Overcoming Exploration in Reinforcement Learning with Demonstrations. *arXiv:1709.10089*.

Ng, A. Y.; Harada, D.; and Russell, S. J. 1999. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, 278–287.

Osband, I.; Blundell, C.; Pritzel, A.; and Van Roy, B. 2016. Deep Exploration via Bootstrapped DQN. *arXiv:1602.04621*.

Paine, T. L.; Gulcehre, C.; Shahriari, B.; Denil, M.; Hoffman, M.; Soyer, H.; Tanburn, R.; Kapturowski, S.; Rabinowitz, N.; Williams, D.; Barth-Maron, G.; Wang, Z.; and de Freitas, N. 2019. Making Efficient Use of Demonstrations to Solve Hard Exploration Problems. *arXiv:1909.01387*.

Pathak, D.; Agrawal, P.; Efros, A. A.; and Darrell, T. 2017. Curiosity-driven Exploration by Self-supervised Prediction. *arXiv:1705.05363*.

Penedones, H.; Vincent, D.; Maennel, H.; Gelly, S.; Mann, T.; and Barreto, A. 2018. Temporal Difference Learning with Neural Networks - Study of the Leakage Propagation Problem. *arXiv:1807.03064*.

Pugh, J. K.; Soros, L. B.; Szerlip, P. A.; and Stanley, K. O. 2015. Confronting the Challenge of Quality Diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, 967–974. New York, NY, USA: ACM.

Pugh, J. K.; Soros, L. B.; and Stanley, K. O. 2016. Quality Diversity: A New Frontier for Evolutionary Computation. *Front. Robot. AI* 3.

Resnick, C.; Raileanu, R.; Kapoor, S.; Peysakhovich, A.; Cho, K.; and Bruna, J. 2018. Backplay: "Man muss immer umkehren". *arXiv:1807.06919*.

Riedmiller, M.; Hafner, R.; Lampe, T.; Neunert, M.; Degrave, J.; Van de Wiele, T.; Mnih, V.; Heess, N.; and Springenberg, J. T. 2018. Learning by Playing - Solving Sparse Reward Tasks from Scratch. *arXiv:1802.10567*.

Salimans, T., and Chen, R. 2018. Learning Montezuma's Revenge from a Single Demonstration. *arXiv:1812.03381*.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized Experience Replay. *arXiv:1511.05952*.

Schulman, J.; Levine, S.; Moritz, P.; Jordan, M. I.; and Abbeel, P. 2015. Trust Region Policy Optimization. *arXiv:1502.05477*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*.

Stadie, B. C.; Levine, S.; and Abbeel, P. 2015. Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models. *arXiv:1507.00814*.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

Tang, H.; Houthooft, R.; Foote, D.; Stooke, A.; Chen, X.; Duan, Y.; Schulman, J.; De Turck, F.; and Abbeel, P. 2016. #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. *arXiv:1611.04717*.

Tassa, Y., and et. al. 2018. DeepMind Control Suite. *arXiv:1801.00690*.

van Hasselt, H.; Doron, Y.; Strub, F.; Hessel, M.; Sonnerat, N.; and Modayil, J. 2018. Deep Reinforcement Learning and the Deadly Triad. *arXiv:1812.02648*.

Wilson, E. B. 1927. Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association* 22(158):209–212.

Supplemental Materials

S1 Phase 1: Explore Until Solved

Algorithm 4: Exploration algorithm

Input : $s_0 \in S$ the initial environment state
step : $S \times A \to S \times \mathbb{R} \times \mathbb{B}$ the environment step function
iterations $\in \mathbb{N}$ the number of samples to accumulate
Bin : $F \to \mathbb{N}$ a binning function

Output: transitions $\subseteq S \times A \times \mathbb{R} \times \mathbb{B} \times S$ the set of explored transitions

```
1  transitions = ∅
2  b_0 = Bin(s_0)
3  bin_usage[b_0] = 0
4  states_in_bin[b_0] = {s_0}
5  state_usage[s_0] = 0
6  B = B ∪ {b_0}
7  while |transitions| < iterations do
8      chosen_bin = argmin_{b∈B} bin_usage[b]
9      chosen_state = argmin_{s∈states_in_bin[chosen_bin]}
         state_usage[chosen_state]
10     action = random_action()
11     s',reward,terminal = step(chosen_state, action)
12     bin_usage[chosen_bin] ++
13     state_usage[chosen_state] ++
14     transitions = transitions ∪
         (chosen_state,action,reward,terminal,s')
15     if Not terminal then
16         state_usage[s'] = 0
17         b' = Bin(s')
18         if b' ∈ B then
19             states_in_bin[b'] = states_in_bin[b'] ∪
                 {s'}
20         else
21             B = B ∪ {b'}
22             bin_usage[b'] = 0
23             states_in_bin[b'] = {s'}
24         end
25     end
26 end
```

Our proposed phase 1 exploration algorithm maintains a pool of states, which initially contains only the start state. At each step, a selection process described below is used to select a state s from the pool (lines 8 to 9 in Algorithm 4). A random action a is then chosen, and the environment is used to compute a single step from state s using action a (line 11). If the resulting state s' is non-terminal, then it is added to the pool (lines 16 to 23). This process is repeated as long as necessary.

State selection The pool of states stored by the algorithm is divided into square bins of size 0.05. During the state selection process (lines 8 to 9), the least-chosen bin is selected (on line 8), then the least-chosen state from this bin is selected (line 9). In both cases, when several states or bins are tied in the argmin operation, one is selected uniformly randomly from the set of all tied elements.

S2 Choice of PBCS Hyperparameters

PBCS uses three hyperparameters α, β and ϵ.

The parameter α represents the number of consecutive non-improvements of the training performance required to assume training is finished. In our experiments, this value was set to 10, and we summarize here what can be expected if this parameter is set too high or too low.

- Setting α too high results in longer training sessions, in which the policy keeps being trained despite being already successful. The time and sample performance of PBCS is impacted, but the algorithm should still be able to build a policy chain.
- Setting α too low may cause training to stop early. In benign cases, the policy is simply sub-optimal, but in some cases this may lead to the creation of many changepoints, and prevent PBCS from improving at all upon the phase 1 trajectory. If the activation conditions overlap too much, PBCS may output a skill chain that is unable to navigate the environment.

The parameter β represents the number of samples used to evaluate the performance of a skill. In our experiments, this value was set to 50.

- Setting β too high would increase the time and sample complexity of PBCS, but would not impact the output.
- The main risk of setting β too low is that PBCS may incorrectly compute that a skill has a performance of 100%. If this skill is then selected, the output skill chain may be unable to navigate the environment.

The parameter ϵ corresponds to the radius of the targets used during skill chaining.

S3 Choice of Discount Factor γ

The discount factor γ is usually considered to be a parameter of the environment and not the RL algorithm. It controls the decay that is applied when evaluated the contributions of future rewards to a present choice. In our experiments, we tested two values of γ, that are $\gamma = 0.9$ and $\gamma = 0.99$.

DDPG uses a neural network $\hat{Q}$ in order to estimate the state-action value function $Q^\pi(s, a)$ of the current policy π. In the case of deterministic environments, the state-action value function is recursively defined as $Q^\pi(s, a) = R(s, a, s') + \gamma Q(s', \pi(a))$, where $s' = \text{step}(s, a)$.

Therefore, reaching a sparse reward of value 1 after n steps with no reward carries a discounted value of γ^n. This implies that rewards that are reached only after many steps have very little impact on the shape of Q. For instance, with $\gamma = 0.9$ and $n = 50$, $\gamma^n \approx 0.05$. This effectively reduces the magnitude of the training signal used by the actor update

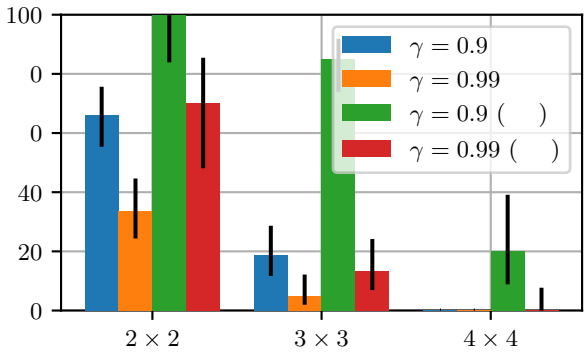


Figure S1: Success rate of PBCS without skill chaining, depending on γ. Bars marked with *(am)* use the variant of DDPG presented in Sect. S3.1. Error bars are computed using Wilson score intervals (Wilson 1927).

of DDPG, reducing the speed of actor updates the farther from the reward. Evidence of this is presented in Sect. S3.1.

With this consideration, it seems that choosing γ very close to 1 solves the problem of exponential decay of the training signal. However, high γ values present their own challenges. The state-action critic approximator $\hat{Q}$ used by DDPG is trained on (s, a, r, s') tuples stored in an experience replay buffer, but as with any continuous approximator, it generalizes the training data to nearby (s, a) couples.

In environments with positive rewards, $\hat{Q}$ can overestimate the value of states: for instance in maze environments, the learned value can be generalized incorrectly and "leak" through walls.

This mechanism is usually counter-balanced by the fact that over-estimated $Q(s, a)$ values can then be lowered. For instance, in our maze environments, hitting a wall generates a training tuple with $s' = s$ and $r = 0$. The update rule of DDPG applied to this tuple yields: $Q(s, a) \leftarrow Q(s, a)(1 + c(\gamma - 1))$ where c is the critic learning rate. Therefore, the closer γ is to 1, the slower over-estimated values will be corrected.

In smaller mazes, our experiments show that reducing gamma increases the performance of PBCS without skill chaining (Fig. S1).

S3.1 Replacing the Actor Update of DDPG

We claim that the lower reward signal obtained with γ values close to 1 affect the actor update of DDPG. We can test this claim by using a variation of DDPG proposed by the authors of (Matheron, Perrin, and Sigaud 2019): we replace the actor update of DDPG with a brute-force approach that, for each sampled state s, computes $\max_a \hat{Q}(s, a)$ using uniform sampling. The performance of this variant is presented in Fig. S1 with green and red bars.

S4 Experimental Setup

Our experiments are conducted in maze environments of various sizes. A maze of size N is described using the following Markov Decision Process:

$$
\begin{aligned}
S &= [0, N] \times [0, N] \\
A &= [-0.1, 0.1] \times [-0.1, 0.1] \\
R(s, a, s') &= \mathbb{1}_{\|s' - \text{target}\| < 0.2} - \mathbb{1}_{[s,s'] \text{ intersects a wall}} \\
\text{step}(s, a) &= \begin{cases} s & \text{if } [s, s + a] \text{ intersects a wall} \\ s + a & \text{otherwise.} \end{cases}
\end{aligned}
$$

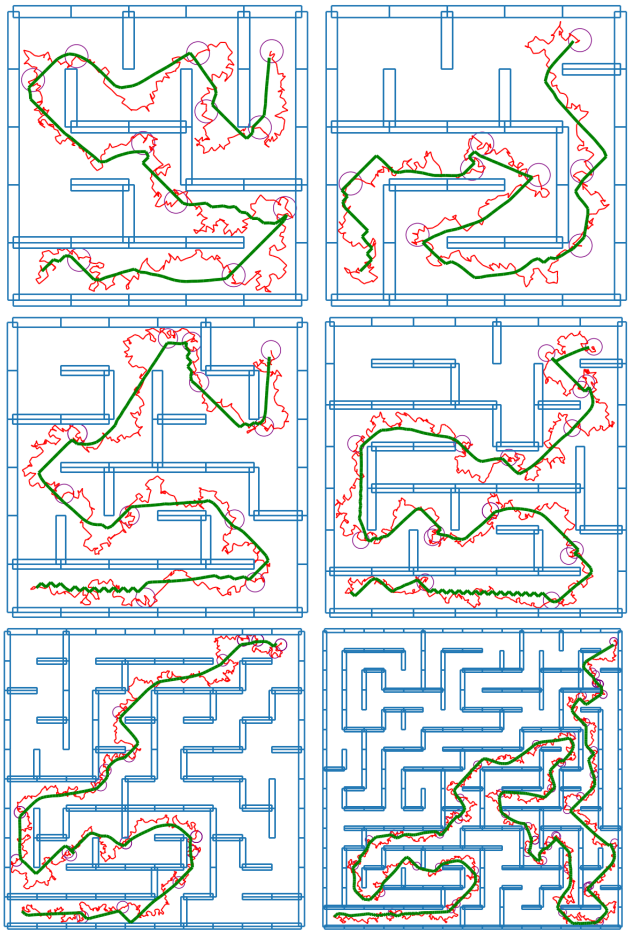
The set of walls is constructed using a maze generation algorithm, and walls have a thickness of 0.1.

The target position is $(N - .5, N - .5)$ when $N > 2$. In mazes of size 2, the target position is $(.5, 1.5)$.

S5 Enlarged Results

A more detailed view of the results of PBCS on mazes of different sizes is presented in Fig. 1.

Figure S2: Enlarged view of the results of PBCS on mazes of different sizes. The trajectory computed in phase 1 is displayed in red, and the "robustified" policy chain is displayed in green. Activation sets A_i are displayed as purple circles.



S6 Need for Resetting in Unseen States

As a reminder, for the Backplay algorithm and our variant, a single trajectory $\tau_0 \dots \tau_T$ is provided, and training is performed by changing the starting point of the environment to various states.

In the original Backplay algorithm, the environment is always reset to a visited state τ_K, where K is an index chosen randomly in a sliding window of $[0, T]$. The sliding window is controlled by hyperparameters, but the main idea is that in the early stages of training, states near T are more likely to be selected, and in later stages, states near 0 are more likely to be used.

However, we found that this caused a major issue when combined with continuous control and the skill chaining process. With skill chaining, the algorithm creates a sequence of activation sets (A_n), and a sequence of policies (π_n) such that when the agent reaches a state in A_n, it switches to policy π_n. Each activation set A_n is a ball of radius ϵ centered around a state τ_K for some K.

The policy needs to be trained not only on portions of the environment that are increasingly long, it also needs to account for the uncertainty of its starting point. When executing the skill chain, the controller switched to policy π_n as soon as the state reaches the activation set A_n, which is $B_\epsilon(\tau_K)$ for some K. Even if A_n is relatively small, we found it caused systematic issues on maze environments, as presented in Fig. S3.

In our variant of the Backplay algorithm, we found it was necessary to train DDPG on starting points chosen randomly in $B_\epsilon(\tau_K)$, to ensure that the policy is trained correctly to solve a portion of the environment with any starting point in this volume.

This also means that we need to reset the environment to unseen states, and can cause problems when these states are unreachable (in our maze examples this is usually because they are inside walls, but in higher dimensions we assume this could be more problematic).

When possible, a solution would be to run the environment backwards from τ_K with random actions to generate these samples (while ensuring that they still lie within $B_\epsilon(\tau_K)$). Another solution, especially in high-dimension environments, would be to run the environment backwards for a fixed number of steps, and use a classifier to define the bounds of A_n, instead of using the L2 distance.

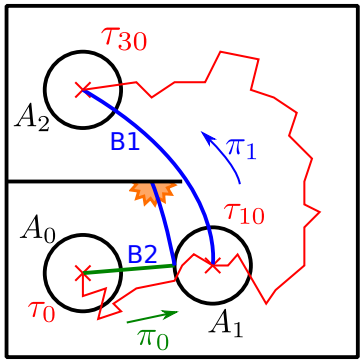


Figure S3: Policy π_1 was trained using starting points $\tau_{30} \dots \tau_{10}$ without any added noise. Therefore, τ_{30} is reachable from τ_{10} using π_1 (trajectory $B2$), but not necessarily form any point in A_1. In maze environments, the optimal policy is usually close to walls, and provides little margin for perturbations. The trajectory $B2$ (that starts in green and ends in blue) results from the execution of the skill chain. The controller switches from π_0 to π_1 as soon as the agent reaches A_1, and then hits the wall (trajectory $B2$). This problem persists even when ϵ is reduced.

Hierarchical Reinforcement Learning in StarCraft II with Human Expertise in Subgoals Selection

Xinyi Xu[*1], **Tiancheng Huang**[2]
Pengfei Wei[1], **Akshay Narayan**[1], **Tze-Yun Leong**[1]
[1]NUS, School of Computing, Medical Computing Lab,
{xuxinyi,weipf,anarayan,leongty}@comp.nus.edu.sg
[2]NTU, School of Computer Science and Engineering,
thuang013@e.ntu.edu.sg

Abstract

This work is inspired by recent advances in hierarchical reinforcement learning (HRL) (Barto and Mahadevan 2003; Hengst 2010), and improvements in learning efficiency from heuristic-based subgoal selection, experience replay (Lin 1993; Andrychowicz et al. 2017), and task-based curriculum learning (Bengio et al. 2009; Zaremba and Sutskever 2014). We propose a new method to integrate HRL, experience replay and effective subgoal selection through an implicit curriculum design based on human expertise to support sample-efficient learning and enhance interpretability of the agent's behavior. Human expertise remains indispensable in many areas such as medicine (Buch, Ahmed, and Maruthappu 2018) and law (Cath 2018), where interpretability, explainability and transparency are crucial in the decision making process, for ethical and legal reasons. Our method simplifies the complex task sets for achieving the overall objectives by decomposing them into subgoals at different levels of abstraction. Incorporating relevant subjective knowledge also significantly reduces the computational resources spent in exploration for RL, especially in high speed, changing, and complex environments where the transition dynamics cannot be effectively learned and modelled in a short time. Experimental results in two StarCraft II (SC2) (Vinyals et al. 2017) minigames demonstrate that our method can achieve better sample efficiency than flat and end-to-end RL methods, and provides an effective method for explaining the agent's performance.

Introduction

Reinforcement learning (RL) (Sutton and Barto 2018) enables agents to learn how to take actions, by interacting with an environment, to maximize a series of rewards received over time. In combination with advances in deep learning and computational resources, the Deep Reinforcement Learning (DRL) (Mnih et al. 2013) formulation has led to dramatic results in acting from perception (Mnih et al. 2015), game playing (Silver et al. 2016), and robotics (Andrychowicz et al. 2020). However, DRL usually requires extensive computations to achieve satisfactory performance. For example, in full-length StarCraft II (SC2) games, AlphaStar (Vinyals et al. 2019) achieves superhuman performance at the expense of huge computational resources[1]. Training flat DRL agents even on minigames (simplistic versions of the full-length SC2 games) requires 600 million samples (Vinyals et al. 2017) and 10 billion samples (Zambaldi et al. 2019) for each minigame, and repeated with 100 different sets of hyper-parameters, approximately equivalent to over 630 and 10,500 years of game playing time respectively. Even with such large number of training samples, DRL agents are not yet able to beat human experts at some minigames (Vinyals et al. 2017; Zambaldi et al. 2019).

We argue that learning a new task in general or SC2 minigames in particular is a two-stage process, viz., learning the fundamentals, and mastering the skills. For SC2 minigames, novice human players learn the minigame fundamentals reasonably quickly by decomposing the game into smaller, distinct and necessary steps. However, to achieve mastery over the minigame, humans take a long time, mainly to practice the precision of skills. RL agents, on the other hand, may take a long time to learn the fundamentals of the gameplay but achieve mastery (stage two) efficiently. This can be observed from the training progress curves in (Vinyals et al. 2017) which shows spikes followed plateaus of reward signals instead of steady and gradual increases.

We want to leverage human expertise to reduce the 'warm-up' time required by the RL agents. The Hierarchical Reinforcement Learning (HRL) framework (Bakker and Schmidhuber 2004; Levy et al. 2019) comprises a general layered architecture that supports different levels of abstractions corresponding to human expertise and agent's skills at the low-level manoeuvres. Intuitively, HRL provides a way for combining the best from human expertise and agent by organizing the inputs from humans at a high level (more abstract) and those from agents at a lower level (more precise). In this work, we extend the HRL framework to incorporate human expertise in subgoal selection. We demonstrate the effects of our methods in mastering SC2 minigames, and present preliminary results on sample efficiency and inter-

*Corresponding author

[1]According to (Vinyals et al. 2019), for each of their 12 agents, they conduct training on 32 TPUs for 44 days.

pretability over the flat RL methods.

The rest of the paper is organized as follows. We briefly outline the background information in the next section. Next, we describe our proposed methodology. Further, we discuss the related works and present our experimental results. We then conclude the paper highlighting opportunities for future work.

Preliminaries

Markov decision process and Reinforcement learning: A Markov decision process (MDP) is a five-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where, $\mathcal{S}$ is the set of states the agent can be in; $\mathcal{A}$ is the set of possible actions available for the agent; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is the reward function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \Delta\mathcal{S}$ is the transition function; and $\gamma \in [0, 1]$ is the discount factor that denotes the usefulness of the future rewards. We consider the standard formalism of reinforcement learning where an agent continuously interacts with a fully observable environment, defined using an MDP. A deterministic policy is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$ and we can describe a sequence of actions and reward signals from the environment. Every episode begins with an initial s_0. At each t, the agent takes an action $a_t = \pi_t(s_t)$, and gets a reward $r_t = \mathcal{R}(s_t, a_t)$. At the same time, s_{t+1} is sampled from $\mathcal{T}(s_t, a_t)$. Over time, the discounted cumulative reward, called *return*, is calculated as: $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_t$. The agent's task is to maximize the expected *return* $\mathbb{E}_{s_0}[R_0|s_0]$. Furthermore, the Q-function (or action-value function) is defined as $Q^\pi(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]$. Assuming an optimal policy $\pi^* : Q^{\pi^*}(s, a) \geq Q^\pi(s, a)\ \forall s \in \mathcal{S}, a \in \mathcal{A}$, for any possible π. All optimal policies have the same Q-function called the *optimal Q-function*, denoted Q^*, satisfying this Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{T}(s,a)}[\mathcal{R}(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')].$$

Q-function Approximators The above definitions enable one possible solution to MDPs: using a function approximator for Q^*. *Deep-Q-Networks* (DQN) (Mnih et al. 2013) and *Deep Deterministic Policy Gradients* (DDPG) (Lillicrap et al. 2016), are such approaches tackling model-free RL problems. Typically, a neural network Q is trained to approximate Q^*. During training, experiences are generated via an *exploration policy*, usually ϵ-greedy policy with the current Q. The experience tuples (s_t, a_t, r_t, s_{t+1}) are stored in a *replay buffer*. Q is trained using gradient descent with respect to the loss $L \coloneqq \mathbb{E}[Q(s_t, a_t) - y_t]^2$, where $y_t = r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$ with experiences sampled from the *replay buffer*.

An *exploration policy* is a policy that describes how the agent interacts with the environment. For instance, a policy that picks actions randomly encourages *exploration*. On the other hand, a *greedy* policy with respect to Q, as in $\pi_Q(s) = \text{argmax}_{a \in \mathcal{A}} Q(s, a)$, encourages exploitation. To balance these, a standard approach of ϵ-greedy (Sutton and Barto 2018) is adopted: with probability ϵ take a random action, and with probability $1 - \epsilon$ take a *greedy* action.

Goal Space $\mathcal{G}$ Schaul et al. (2015) extended DQN to include a goal space $\mathcal{G}$. A (sub)goal can be described with specifically selected states, or via functions such as $f : \mathcal{S} \mapsto [0, 1]$, either a state is a goal or not. Introducing $\mathcal{G}$ modifies the original reward function $\mathcal{R}$ slightly: $\forall g \in \mathcal{G}$, $\mathcal{R}_g : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, $\mathcal{R}(s, a|g) \coloneqq \mathcal{R}_g(s, a)$. At the beginning of each episode, in addition to s_0, the initialization includes a fixed g to create a tuple (s_0, g). Other modifications naturally follow: $\pi : \mathcal{S} \times \mathcal{G} \mapsto \mathcal{A}$, and $Q^\pi(s_t, a_t, g) = \mathbb{E}[\mathcal{R}_t|s_t, a_t, g]$.

Experience Replay Lin (1993) proposed the idea of using 'experiences buffers' to help machines learn. Formally, a single time step experience is defined as a tuple (s_t, a_t, r_t, s_{t+1}) and more generally an experience can be constructed by concatenating multiple consecutive experience tuples.

Curriculum Learning Methods in this framework typically explicitly or implicitly design a series of tasks or goals (with gradually increased difficulties) for the agent to follow and learn, i.e., the curriculum (Bengio et al. 2009; Weng 2020).

StarCraft II SC2 is a real-time-strategy (RTS) game, where players command their units to compete against each other. In an SC2 full-length game, typically players start out by commanding units to collect resources (minerals and gas) to build up their economy and army at the same time. When they have amassed a sufficiently large army, they command these units to attack their opponents' base in order to win. SC2 is currently a very promising simulation environment for RL, due to its high flexibility and complexity and wide-ranging applicability in the fields of game theory, planning and decision making, operations optimization, etc. SC2 minigames, as opposed to full-length games described above, are built-in episodic tutorials where novice players can learn and practice their skills in a controlled and less complex environment. Some relevant skills include collecting resources, building certain army units, etc.

Proposed Methodology

We propose a novel method of integrating the advantages of human expertise and RL agents to facilitate fundamentals learning and skills mastery of a learning task. Our method adopts the principle of *Curriculum Learning* (Bengio et al. 2009) and follows a task-oriented approach (Zaremba and Sutskever 2014). The key idea is to leverage human expertise to simplify the complex learning procedure, by decomposing it into hierarchical subgoals as the curriculum for the agent. More specifically, we factorize the learning task into several successive subtasks indispensable for the agent to complete the entire complex learning procedure. The customized reward function in each subtask implicitly captures the corresponding subgoal. Importantly, these successive subgoals are determined so that they are gradually more difficult to improve learning efficiency (Bengio et al. 2009; Justesen et al. 2018). With defined subgoals, we use the *Experience Replay* technique to construct the experiences to further improve the empirical sample efficiency (Andrychowicz et al. 2017; Bakker and Schmidhuber 2004; Levy et al. 2019). Furthermore, adopting clearly defined subtasks and subgoals enhances the interpretability of the agent's learning progress. In implementation, we customize SC2 minigames to embed human expertise on subgoal information and the criteria to identify and select subgoals during learning. Therefore, the agent learns

the subpolicies and combines them in a hierarchical way. By following a well-defined decomposition of the original minigame into subtasks, we can choose the desired state of a previous subtask to be the starting conditions of the next subtask, thus completing the connection between subtasks.

Hierarchy: Subgoals and Subtasks

Our proposed hierarchy is composed of subgoals, which collectively divide the problem into simpler subtasks that can be solved easily and efficiently. Each subgoal is implicitly captured as the desired state in its corresponding subtask, and we refer to the agent's skills to reach a subgoal its corresponding subpolicy. The rationale behind this is as follows. First, the advantages of human expertise and the agents are complementary to each other in terms of learning and mastering the task. Human players are good at seeing the big picture and thus identifying the essential and distinct steps/skills very quickly. On the other hand, agents are proficient in honing learned skills and maneuvers to a high degree of precision. Second, a hierarchy helps reduce the complexity of search space via divide-and-conquer. Lastly, this method enhances the interpretability of the subgoals (and subpolicies).

Figure 1 illustrates the concept of subgoals and subpolicies with a simple navigation agent. The agent is learning to navigate to the flag post from the initial state s_0. One possible sequence of the states is $s_1, \ldots, s_5$. Therefore, the entire trajectory can be decomposed into subgoals; for instance, Levy et al. (2019) used heuristic-based subgoal selection criteria (in Figure 1 these selected subgoals, $g_0, \ldots, g_4$, are denoted by orange circles). On the other hand, the sequence of red nodes denote subgoals of our method. We highlight that this sequence would constitute a better guided and more efficient exploration path. In addition this sequence is better aligned with the game where some states are the prerequisites for other states (illustrated as the black dashed arrows).

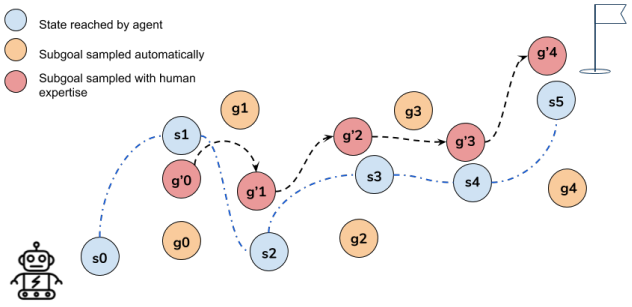


Figure 1: Navigation Agent

Subgoals Selection and Experience Replays

Subgoal Design and Selection. We use the similar method for constructing experiences with a goal space as previous works (Andrychowicz et al. 2017; Levy et al. 2019). However, our method introduces human expertise in constructing the hierarchy and subgoals selection. In (Andrychowicz et al. 2017), the *hindsight experience replay* buffer is

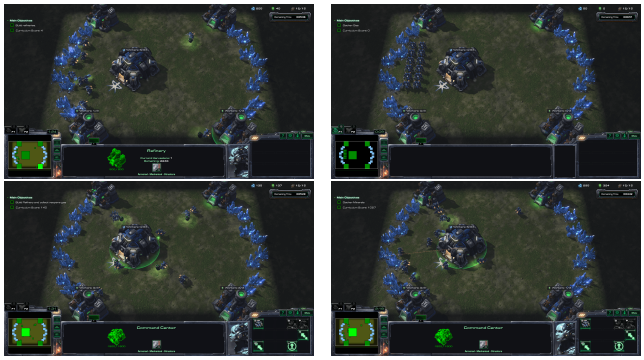


Figure 2: Collect Minerals and Gas. From left to right, top to bottom:(1)-(4): (1) to build refineries; (2) to collect gas with built refineries; (3) both tasks in (1) and (2); (4) all three tasks in (1), (2), (3) and collect minerals.

Figure 3: Build Marines. From left to right, top to bottom:(1)-(4): (1) to build supply depots; (2) to build barracks; (3) to build marines with (1) and (2) already built; (4) all three tasks in (1), (2), (3).

constructed via random sampling from the goal space and concatenating the sampled goals to an *already executed* sequence $\{s_1, \ldots, s_T\}$, hence the name hindsight. The subgoals are initialized with heuristic-based selection and updated according to *hindsight actions*. For example, in Figure 1, given a predetermined subgoal g_0, the agent might not successfully reach it, and instead ends up in s_1. In this case, the subgoal set in hindsight is s_1 (updated from g_0).

Our method distinguishes in that the (sub)goals selection strategy is designed with human expertise, to give a fixed but suitable decomposition of the learning task. Furthermore, we exploit the underlying sequential relationship among the subgoals as in the game some states are the prerequisites for others. Hence, certain actions are required to be performed in order. Furthermore, another reason for introducing human expertise rather than using end-to-end learning alone is that compared with the environments investigated in previous HRL works, SC2 encompasses a significantly larger state-action space that prohibits a sample-efficient end-to-end learning strategy. As a result, our method enjoys an added advantage of interpretability of the selected subgoals.

Subtasks Implementations. We leverage the customizabil-

ity of SC2 minigames to carefully design subtasks to enable training of the corresponding subpolicies, as suggested in (Barto and Mahadevan 2003). We illustrate with the *Collect Minerals and Gas* (CMAG) minigame, as shown and described in Figure 2. There are several distinct and sequential actions the player has to perform to score well: 1. commanding the Space Construction Vehicles (SCVs) - basic units of the game, to collect minerals; 2. having collected sufficient minerals, selecting SCVs to build the gas refinery (a prerequisite building for collecting vespene gas) on specific locations with existing gas wells; 3. commanding the SCVs to collect vespene gas from the constructed gas refinery; 4. producing additional SCVs (at a fixed cost) to optimize the mining efficiency. And there is a fixed time duration of 900 seconds. The challenge of CMAG is that all these actions/subpolicies should be performed in an optimized sequence for best performance. The optimality depends on the order, timing, and the number of repetitions of these actions. For instance, it is important not to under/over-produce SCVs at a mineral site for optimal efficiency. Hence, we implemented the following subtasks: *BuildRefinery*, *CollectGasWithRefineries* and *BuildRefinieryAndCollectGas*. In the first two subtasks, the agent learns the specific subpolicies to build refineries and to collect gas (from built refineries), respectively, while in the last subtask the agent learns to combine them. Based on the same idea, the complete decomposition for CMAG is given by [CMAG, *BuildRefinery*, *CollectGasWithRefineries*, *BuildRefineryAndCollectGas*, CMAG] where the first CMAG trains the agent to collect minerals, and the last CMAG trains it to combine all subpolicies and also 're-introduces' the reward signal for collecting minerals to avoid forgetting (Zaremba and Sutskever 2014). Similarly, for the BuildMarines (BM) minigame, shown in Figure 3, the sequential steps/actions are: 1. commanding the SCVs to collect minerals; 2. having collected sufficient minerals, selecting SCVs to build a supply depot (a prerequisite building for barracks and to increase the supplies limit); 3. having both sufficient minerals and a supply depot, selecting SCVs to build barracks; 4. having minerals, a supply depot and barracks and with current unit count less than the supplies limit, selecting the barracks to train marines. The fixed time duration for BM is 450 seconds. Therefore, we implemented the corresponding subtasks: *BuildSupplyDepots*, *BuildBarracks*, *BuildMarinesWithBarracks* and the complete decomposition for BM is [*BuildSupplyDepots*, *BuildBarracks*, *BuildMarinesWithBarracks*, BM]. Note we do *not* set BM as a first subtask as for CMAG because CMAG contains both reward signals for minerals and gas, so it is an adequate simple task for the agent to learn to collect minerals. However, BM has only the reward signals for training marines, thus too difficult as the first subtask.

Construct Experience Replay for Each Subtask. With the designed subtasks represented by our customized minigames, constructing experience replays is straightforward. For a subtask, a predetermined subgoal g_i is implicitly captured in its customized minigame (e.g., to build barracks, to manufacture SCVs, etc.) using a corresponding reward signal, so that the agent learns to reach g_i. For the immediate subsequent subtask, we set its initial conditions to be the completed subgoal g_i. So, the agent learns to continue on the basis of a completed g_i. It is an implicit process because, when learning to reach subgoal g_{i+1}, the agent does *not* see or interact directly with the reward signal corresponding to g_i. For example, between two ordered subtasks *CollectMinerals* and *BuildRefinery*, the agent learns to collect minerals first and starts with some collected minerals in the latter with the sole objective of learning to build refineries.

Off-policy learning and PPO. Off-policy learning is a learning paradigm where the exploration and learning are decoupled and take place separately. Exploration is mainly used by the agent to collect experiences or 'data points' for its policy function or model. Learning is then conducted on these collected experiences, and Proximal Policy Optimization (PPO) (Schulman et al. 2017) is one such method. Its details are not the focus of this work and omitted here.

Algorithm. We describe the HRL algorithm with human expertise in subgoal selection here. The pseudo-code is given in Algorithm 1. For a learning task, a sequence of subtasks is designed with human expertise to implicitly define the subgoals and we refer to our customized SC2 minigames as subtasks $\Gamma_i, 0 \leq i < m$ for the learning task. We predefine reward thresholds $thresholds \in \mathbb{R}^m$, for all subtasks. As the agent's running average reward is higher than a threshold, this agent is considered to have learnt the corresponding subtask well and will move to the subsequent subtask. We use learner $\mathcal{L}$ to denote the agent and to describe how it makes decisions and takes actions. It can be represented by a deep neural network, and parametrized by $\vec{w}_{\mathcal{L}}$. In addition, we define a sample count c and sample limit n. Sample count c refers to the number of samples the agent has used for learning a subtask. Sample limit n refers to the total number of samples allowed for the agent for the entire learning task, i.e., for all subtasks combined. c and n together are used to demonstrate empirical sample efficiency.

With these definitions and initializations, the algorithm takes the defined sequence of subtasks Γ with corresponding $thresholds$ and initiates learning on these subtasks in the same sequence. During the process, a running average of the agent's past achieved rewards is kept for each subtask, represented by the API call `test()`. For each subtask Γ_i, either the agent completely exhausts its assigned sample limit $\lfloor \frac{n}{m} \rfloor$ or it successfully reaches the $thresholds_i$. If the running average of past rewards $\geq thresholds_i$, the agent completes learning on Γ_i and starts with Γ_{i+1}; the process continues until all subtasks are learned. We follow the *exploration policy* in preliminaries and adopt an ϵ-greedy policy, represented by `explore()` in Algorithm 1.

Related Work

Experience Replay RL has achieved impressive developments in robotics (Singh et al. 2019), strategic games such as Go (Silver et al. 2017), real-time strategy games (Zambaldi et al. 2019; Vinyals et al. 2019) etc. Researchers have attempted in various ways to address the challenge of goal-learning, reward shaping to get the 'agent' to learn to master the task, and yet not overfit to the particular instances of the goals or reward signals. *Experience Replay* (Lin 1993) is a technique to store and re-use past records of executions

Algorithm 1 HRL with Human Expertise in Subgoal Selection

Input: subtasks $\Gamma_i, 0 \leq i < m$
Input: reward thresholds $thresholds \in \mathbb{R}^m$
Input: learner $\mathcal{L}$, parametrized by $\vec{w}_{\mathcal{L}}$
Input: sample count c, sample limit n.
for $0 \leq i < m$ **do**
  $c \leftarrow 0$
  while $c <= \lfloor \frac{n}{m} \rfloor$ **do**
    $experiences \leftarrow \texttt{explore}(\mathcal{L}, \Gamma_i)$
    $c \leftarrow c + |experiences|$
    $\vec{w}'_{\mathcal{L}} \leftarrow \texttt{PPO}(\vec{w}_{\mathcal{L}}, experiences)$ ▷ off-policy
    if $\texttt{test}(\vec{w}'_{\mathcal{L}}) \geq thresholds_i$ **then**
      Break ▷ Go to next subtask
    end if
  end while
end for

(along with the signals from the environment) to train the 'agent', achieving efficient sample usage. Mnih et al. (2013) employed this technique together with Deep-Q-Learning to produce state-of-the-art results in Atari, and subsequently Mnih et al. (2015) confirmed the effectiveness of such approach under the stipulation that the 'agent' only sees what human players would see, i.e., the pixels from the screen and some scoring indices.

Curriculum Learning Bengio et al. (2009) hypothesized and empirically showed that introducing gradually more difficult examples speeds up the online learning, using a manually designed task-specific curriculum. Zaremba and Sutskever (2014) experimentally showed that it is important to mix in easy tasks to avoid forgetting. Justesen et al. (2018) demonstrated that training an RL agent over a simple curriculum with gradually increasing difficulty can effectively prevent overfitting and lead to better generalization.

Hierarchical Reinforcement Learning (HRL) HRL and its related concepts such as *options* (Sutton, Precup, and Singh 1999) *macro-actions* (Hauskrecht et al. 1998), or *tasks* (Li, Narayan, and Leong 2017) were introduced to decompose the problem, usually a Markov decision process (MDP), into smaller sub-parts to be efficiently solved. We refer the readers to (Barto and Mahadevan 2003; Hengst 2010) for more comprehensive treatments. We describe two tracks of related works most relevant to our problem. Bakker and Schmidhuber (2004) proposed a two-level hierarchy, using *subgoal* and *subpolicy* to describe the learning taking place at the lower level of the hierarchy. Levy et al. (2019) further articulated these ideas, and explicitly combined them with *Hindsight Experience Replay* (Andrychowicz et al. 2017) for better sample efficiency and performance. Another similarly inspired approach called *context sensitive reinforcement learning* (CSRL) introduced by Li, Narayan, and Leong (2017) employed the hierarchical structure to enable effective re-use of learnt knowledge of similar (sub)tasks in a probabilistic way. In CSRL, instead of *Experience Replay*, efficient simulations over constructed states are used in learning, able to learn both the tasks, and the environment (the transition and reward functions). CSRL scales well with state space, and is relatively easily parallelizable.

StarCraft II In addition to (Zambaldi et al. 2019), several works addressed some of the challenges presented by SC2. In a real-time strategy (RTS) game such as SC2, the hierarchical architecture is an intuitive solution concept, for its efficient representation and interpretability. Similar but different hierarchies were employed in two other works, where Lee et al. (2018) designed the hierarchy with semantic meaning and from a operational perspective while Pang et al. (2019) forewent explicit semantic meanings for higher flexibility. Both provided promising empirical results on the full-length games against built-in AIs. Instead of full-length SC2 games, our investigation targets the minigames and we propose a way to integrate human expertise, the *Curriculum Learning* paradigm and the *Experience Replay* technique into the learning process.

Different from related works, our work adopts a principle-driven HRL approach with human expertise in the subgoal selection and thus an implicit formulation of a curriculum for the agent, on SC2 minigames in order to achieve empirical sample efficiency and to enhance interpretability.

Experiments

In the experiments, we specifically focus on two minigames, viz., BM and CMAG to investigate the effectiveness of our method. We choose these two because, the discrepancies in the performance between trained RL agents and human experts are the most significant as reported in (Vinyals et al. 2017), suggesting these two are the most challenging for non-hierarchical end-to-end learning approaches. For both CMAG and BM, we have implemented our customized SC2 minigames (subtasks) as described in the proposed methodology section, and we pair them with pre-defined reward thresholds. In our experiments, the decompositions for BM and CMAG are [*BuildSupplyDepots*, *BuildBarracks*, *BuildMarinesWithBarracks*, BM], and [CMAG, *BuildRefinery*, *CollectGasWithRefineries*, *BuildRefineryAndCollectGas*, CMAG], respectively.

Experimental Setup

- *Model Architecture and Hyperparameters*. We follow the model architecture of *Fully Convolutional agent* in (Vinyals et al. 2017) by utilizing an open-source implementation by Ring (2018). We use the hyperparameters listed in Table 1.
- *Training & Testing*. In order to evaluate the empirical sample efficiency of our method, we restrict the total number of training samples to be 10 million. Note this is still significantly fewer than 600 million in (Vinyals et al. 2017) or 10 billion in (Zambaldi et al. 2019). Furthermore, we adopt their practice of training multiple agents to report the best results attained. After training, on the trained model, average and maximum scores over 30 independent episodes are reported.
- *Computing Resource*. CPU: Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz, RAM:64 GB, GPU: GeForce RTX 2080 SUPER 8GB. The training time for a single

model initialization: approximately 1.66 hours for CMAG and 1.5 hours for BM.

Table 1: Hyperparameters

	BM	CMAG
Learning rate	0.0007	0.0007
Batch size	32	32
Trajectory length	40	40
Off-policy learning algorithm	PPO	PPO
Reward thresholds	[7,7,7,2]	[300,5,5,5,500]

Table 2: Average Rewards Achieved

Minigame	SC2LE	DRL	Ours	Human Expert
CMAG	3,978	5,055	478.5(527)	7,566
BM	3	123	6.7(6.24)	133

Table 3: Maximum Rewards Achieved

Minigame	SC2LE	DRL	Ours	Human Expert
CMAG	4,130	unreported	1825	7,566
BM	42	unreported	22	133

Table 4: Training Samples Required

Minigame	SC2LE	DRL	Ours	Human Expert
CMAG	6e8	1e10	1e7	N.A
BM	6e8	1e10	3.4e6	N.A

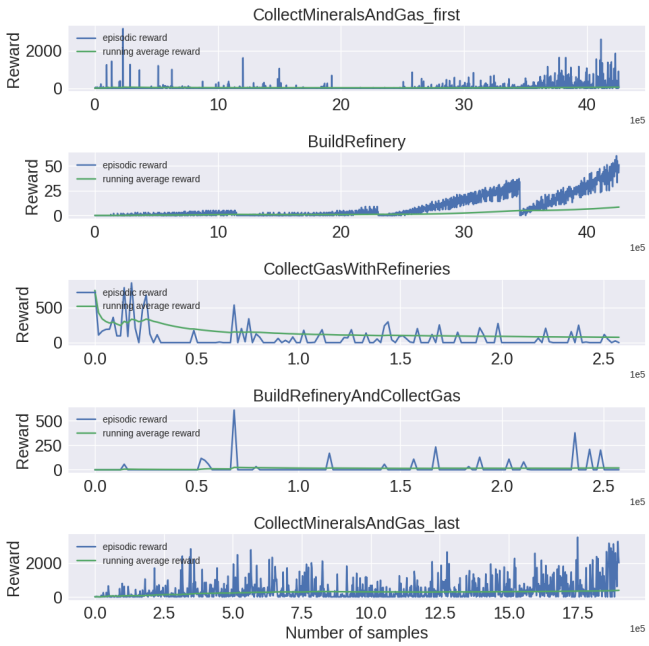


Figure 4: Collect Minerals And Gas learning curve.

Discussion

Our experimental results demonstrate similar trends to those shown in (Vinyals et al. 2017). The variance observed in final performance achieved can be quite large, over different hyperparameter sets, different or same model parameter initializations and other stochasticity involved in learning. For Tables 2 and 3, the higher the values the better. For Table 4, the lower the values the better. Among the 5 agents for BM, the best performing agent can achieve an average reward of 6.7 during testing, while the worst performing agent can barely achieve 0.1. Note that the average reward of 6.7 is twice more than the average reward of the best performing agent (3) reported in (Vinyals et al. 2017) for BM. In addition, our method allows for an in-depth investigation into the agent's learning curves to identify which part of the learning was not effective and led to the sub-optimal final performance. We compare the best (average 6.7) and worst (average 0.1) agents based on their subgoal learning curves, and we find that the best agent is learning effectively across all subgoals. From Figure 5, the learning curves in all subtasks show consistent progress with more samples, where the learning curves of the worst agent show substantially less progress, often flat at zero with very rare spikes, as shown in Figure 6. Especially for the *BuildBarracks* subtask, the agent's learning is ineffective and it only occasionally stumbles upon the correct actions of building barracks at random and receives a corresponding reward signal. Alternatively, the comparison between the running average rewards for these two agents clearly demonstrates that learning for the best agent on the *BuildBarracks* subtask is significantly more effective. The performance on this subtask also affects the final subtask *BuildMarines* since without knowing how to build barracks, the agent cannot take the action of producing marines even if it has learnt this subpolicy. We believe such interpretability and explainability provided by our method are helpful in understanding and improving the learning process and the behavior of the agent.

On the other hand, the experimental results in CMAG show slightly less success. We believe this can be attributed to the difference in the setting of learning. In BM, the agent has to learn distinct skills and how to execute them in sequence in order to perform well, with relatively less emphasis on the degree of mastery of these skills. However, in CMAG the agent's mastery of the skills including mining minerals and gas directly and critically affects its final score, viz., total amount of minerals and gas collected. It means that the agent has to be able to perform the skills well, i.e., optimize with respect to time and manufacturing cost, which in itself can be a separate and more complex learning task. Another experimental difficulty for CMAG lies in the reward scales because the subtasks for collecting minerals and gas have high reward ceilings (as high as several thousand), while those for building the gas refineries have comparatively low reward ceilings (less than one hundred). Due to this large difference in the scales of the reward signals between subtasks, the learning on the subtasks is even more difficult and can be unbalanced.

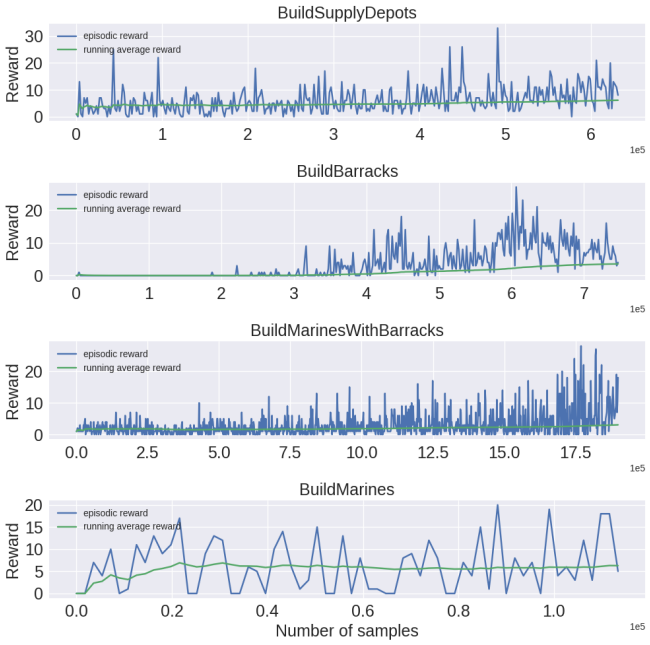


Figure 5: Build Marines learning curve (best agent).

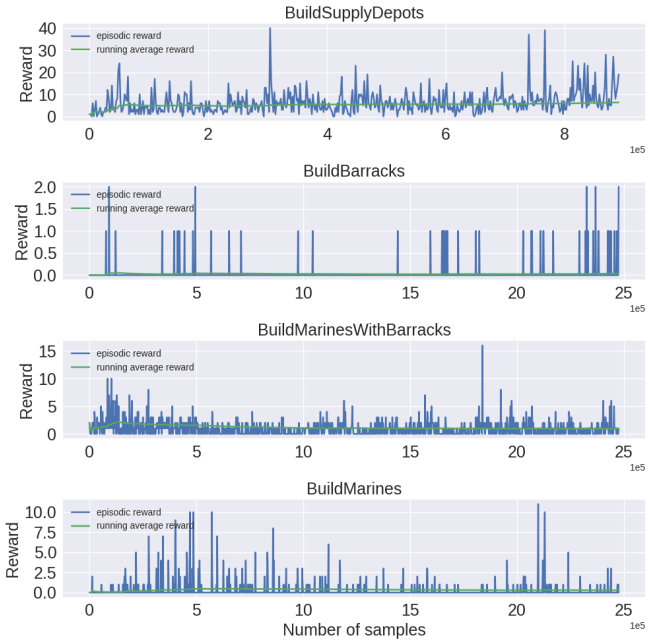


Figure 6: Build Marines learning curve (worst agent).

Conclusion & Future Work

In this work, we examined the SC2 minigames and proposed a way to introduce human expertise to an HRL framework. By designing customized minigames to facilitate learning and leveraging the effectiveness of hierarchical structures in decomposing complex and large problems, we empirically showed that our approach is sample-efficient and enhances interpretability. This initial work invites several exploration directions: developing more efficient and effective ways of introducing human expertise; a more formal and principled state representation to further reduce the complexity of the state space (goal space) with theoretical analysis on its complexity; and a more efficient learning algorithm to pair with the HRL architecture, *Experience Replay* and *Curriculum Learning*.

Acknowledgments

This work was partially supported by an Academic Research Grant T1 251RES1827 from the Ministry of Education in Singapore and a grant from the Advanced Robotics Center at the National University of Singapore.

References

Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Pieter Abbeel, O.; and Zaremba, W. 2017. Hindsight experience replay. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc. 5048–5058.

Andrychowicz, M.; Baker, B.; Chociej, M.; Józefowicz, R.; McGrew, B.; Pachocki, J.; Petron, A.; Plappert, M.; Powell, G.; Ray, A.; Schneider, J.; Sidor, S.; Tobin, J.; Welinder, P.; Weng, L.; and Zaremba, W. 2020. Learning dexterous in-hand manipulation. *International Journal of Robotics Research* 39(1):3–20.

Bakker, B., and Schmidhuber, J. 2004. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8*, 438–445.

Barto, A. G., and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems* 13(1–2):41–77.

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, 41–48. New York, NY, USA: Association for Computing Machinery.

Buch, V.; Ahmed, I.; and Maruthappu, M. 2018. Artificial intelligence in medicine: Current trends and future possibilities. *British Journal of General Practice* 68:143–144.

Cath, C. 2018. Governing artificial intelligence: Ethical, legal and technical opportunities and challenges. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences* 376:20180080.

Hauskrecht, M.; Meuleau, N.; Kaelbling, L. P.; Dean, T.; and Boutilier, C. 1998. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, 220–229. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hengst, B. 2010. Hierarchical reinforcement learning. In Sammut, C., and Webb, G. I., eds., *Encyclopedia of Machine Learning*. Boston, MA: Springer US. 495–502.

Justesen, N.; Torrado, R. R.; Bontrager, P.; Khalifa, A.; Togelius, J.; and Risi, S. 2018. Illuminating generalization in deep reinforcement learning through procedural level generation. In *NeurIPs Workshop on Deep Reinforcement Learning*.

Lee, D.; Tang, H.; Zhang, J. O.; Xu, H.; Darrell, T.; and Abbeel, P. 2018. Modular architecture for starcraft II with deep reinforcement learning. In Rowe, J. P., and Smith, G., eds., *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2018, November 13-17, 2018, Edmonton, Canada*, 187–193. AAAI Press.

Levy, A.; Konidaris, G. D.; Jr., R. P.; and Saenko, K. 2019. Learning multi-level hierarchies with hindsight. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Li, Z.; Narayan, A.; and Leong, T. Y. 2017. An efficient approach to model-based hierarchical reinforcement learning. *31st AAAI Conference on Artificial Intelligence, AAAI 2017* 3583–3589.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*.

Lin, L.-J. 1993. *Reinforcement learning for robots using neural networks*. Ph.D. Dissertation, Carnegie Mellon University.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Pang, Z.-J.; Liu, R.-Z.; Meng, Z.-Y.; Zhang, Y.; Yu, Y.; and Lu, T. 2019. On Reinforcement Learning for Full-Length Game of StarCraft. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4691–4698.

Ring, R. 2018. Reaver: Modular deep reinforcement learning framework. https://github.com/inoryy/reaver.

Schaul, T.; Horgan, D.; Gregor, K.; and Silver, D. 2015. Universal value function approximators. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, 1312–1320. JMLR.org.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *CoRR* abs/1707.06347.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–489.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354–359.

Singh, A.; Yang, L.; Finn, C.; and Levine, S. 2019. End-to-end robotic reinforcement learning without reward engineering. In Bicchi, A.; Kress-Gazit, H.; and Hutchinson, S., eds., *Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019*.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book55 Hayward Street Cambridge MA United States, second edition.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1–2):181–211.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J. P.; Schrittwieser, J.; Quan, J.; Gaffney, S.; Petersen, S.; Simonyan, K.; Schaul, T.; van Hasselt, H.; Silver, D.; Lillicrap, T. P.; Calderone, K.; Keet, P.; Brunasso, A.; Lawrence, D.; Ekermo, A.; Repp, J.; and Tsing, R. 2017. Starcraft II: A new challenge for reinforcement learning. *CoRR* abs/1708.04782.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnevets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782):350–354.

Weng, L. 2020. Curriculum for reinforcement learning. *lilianweng.github.io/lil-log*.

Zambaldi, V. F.; Raposo, D.; Santoro, A.; Bapst, V.; Li, Y.; Babuschkin, I.; Tuyls, K.; Reichert, D. P.; Lillicrap, T. P.; Lockhart, E.; Shanahan, M.; Langston, V.; Pascanu, R.; Botvinick, M.; Vinyals, O.; and Battaglia, P. W. 2019. Deep reinforcement learning with relational inductive biases. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Zaremba, W., and Sutskever, I. 2014. Learning to execute. *CoRR* abs/1410.4615.

Symbolic Network: Generalized Neural Policies for Relational MDPs

Sankalp Garg[1], **Aniket Bajpai**[1], **Mausam**[1]
[1]Indian Institute of Technology Delhi
{sankalp262199, quantum.computing96}@gmail.com, mausam@cse.iitd.ac.in

Abstract

A Relational Markov Decision Process (RMDP) is a first-order representation to express all instances of a single probabilistic planning domain with possibly unbounded number of objects. Early work in RMDPs outputs generalized (instance-independent) first-order policies or value functions as a means to solve *all* instances of a domain at once. Unfortunately, this line of work met with limited success due to inherent limitations of the representation space used in such policies or value functions. Can neural models provide the missing link by easily representing more complex generalized policies, thus making them effective on all instances of a given domain?

We present SYMNET, the first neural approach for solving RMDPs that are expressed in the probabilistic planning language of RDDL. SYMNET trains a set of shared parameters for an RDDL domain using training instances from that domain. For each instance, SYMNET first converts it to an instance graph and then uses relational neural models to compute node embeddings. It then scores each ground action as a function over the first-order action symbols and node embeddings related to the action. Given a new test instance from the same domain, SYMNET architecture with pre-trained parameters scores each ground action and chooses the best action. This can be accomplished in a single forward pass *without any retraining* on the test instance, thus implicitly representing a neural generalized policy for the whole domain. Our experiments on nine RDDL domains from IPPC demonstrate that SYMNET policies are significantly better than random and sometimes even more effective than training a state-of-the-art deep reactive policy from scratch.

1 Introduction

A Relational Markov Decision Process (RMDP) (Boutilier, Reiter, and Price 2001) is a first-order, predicate calculus-based representation for expressing instances of a probabilistic planning domain with a possibly unbounded number of objects. An RMDP *domain* has object types, relational state predicate and action symbols that are applied over objects, first order transition templates that specify probabilistic effects associated with action symbols, and a first-order reward structure. A domain *instance* additionally specifies a set of objects and a start state, thus defining a ground MDP with a known start state (Kolobov, Mausam, and Weld 2012)). *Relational* planners aim to produce a single *generalized* policy that can yield a ground policy for *all* instances of the domain, with little instance-specific computation. *Domain-independent* planners are representation-specific, but domain-agnostic, making them applicable to all domains expressible in the language. In this paper, we design a domain-independent relational planner.

RMDP planners, in their vision, expect to scale to very large problem sizes by exploiting the first-order structures of a domain – thereby reducing the curse of dimensionality. Traditional RMDP planners attempted to find a generalized *first-order* value function or policy using symbolic dynamic programming (Boutilier, Reiter, and Price 2001), or by approximating them via a function over first-order basis functions (e.g., (Guestrin et al. 2003; Sanner and Boutilier 2009)). Unfortunately, these methods met with rather limited success, for e.g., no relational planner participated in International Probabilistic Planning Competition (IPPC)[1] after 2006, even though all competition domains were relational. We believe that this lack of success may be due to the inherent limitations in the representation power of a basis function-based representation. Through this work, we wish to revive the research thread on RMDPs and explore if neural models could be effective in representing these first-order functions.

We present **Sym**bolic **Net**Work (SYMNET), the first domain-independent neural relational planner that computes generalized policies for RMDPs that are expressed in the symbolic representation language of RDDL (Sanner 2010). SYMNET outputs its generalized policy via a neural model whose all parameters are specific to a domain, but tied among all instances of that domain. So, on a new test instance, the policy can be applied out of the box using pre-trained parameters, i.e., without any retraining on the test instance. SYMNET is domain-independent because it converts an RDDL domain file (and instance files) completely automatically into neural architectures, without any human intervention.

SYMNET architecture uses two key ideas. First, it visualizes each state of each domain instance as a graph, where

[1]http://www.icaps-conference.org/index.php/Main/Competitions

nodes represent the *object tuples* that are valid arguments to some relational predicate. An edge between two nodes indicates that an action causes predicates over these two nodes to interact in the instance. The values of predicates in a state act as features for corresponding nodes. SYMNET then learns node (and state) embeddings for these graphs using graph neural networks. Second, SYMNET learns a neural network to represent the policy and value function over this graph-structured state. To learn these in an instance-independent way, we recognize that most ground actions are a first-order action symbol applied over some object tuple. SYMNET scores such ground actions as a function over the action symbol and the relevant embeddings of object tuples. After training all model parameters using reinforcement learning over training instances of a domain, SYMNET architecture can be applied on any new (possibly larger) test problem without any further retraining.

We perform experiments on nine RDDL domains from IPPC 2014 (Grzes, Hoey, and Sanner 2014). Since no planner exists that can run without computation on a given instance, we compare SYMNET to random policies (lower bound) and policies trained from scratch on the test instance. We find that SYMNET obtains hugely better rewards than random, and is quite close to the policies trained from scratch – it even outperforms them in 28% instances. Overall, we believe that our work is a step forward for the difficult problem of domain-independent RMDP planning. We release the code of SYMNET for future research.[2]

2 Background and Related Work

2.1 Probabilistic Planning

Markov Decision Process (MDP): A (ground) finite-horizon MDP (Bellman 1957; Puterman 1994) with a known start state is formalized as a tuple $< S, A, T, R, H, s_0, \gamma >$, where S is the set of states, A is the set of actions, T is the transition model $S \times A \times S \to [0, 1]$, R is the reward model $S \times A \times S \to \mathbb{R}$, H is the horizon and s_0 is the start state, and γ, the discount factor.

Relational Markov Decision Process (RMDP): An RMDP $< \mathcal{C}, \mathcal{SP}, \mathcal{A}, O, \mathcal{T}, \mathcal{R}, H, s_0, \gamma >$ is a first-order representation of a factored MDP (Boutilier, Reiter, and Price 2001), expressed via objects, predicates and functions. Here, $\mathcal{C}$ is a set of classes (types), $\mathcal{SP}$ is the set of state predicate symbols. $\mathcal{A}$ is a set of action predicate symbols, O represents a set of domain objects, each associated with single type from $\mathcal{C}$. It is a first order representation because different sets of objects O can construct different ground MDPs.

Each predicate symbol is declared to take as argument a tuple of object types. A predicate symbol (action symbol) applied over a type-consistent tuple of object variables forms a state variable (respectively, ground action). A ground-state s is, thus, a complete assignment of all predicate symbols $\mathcal{SP}$ applied on all type-consistent object tuples from O (also denoted by $\mathcal{SP}_O$). Similarly, the set of all ground actions (A) can be defined as $\mathcal{A}_O$ – all-action symbols applied on all type-consistent object tuples. We also denote the ground state space S by $\mathcal{P}(\mathcal{SP}_O))$, where $\mathcal{P}$ denotes the powerset. Transition and reward models for an RMDP are defined at the schema level through different languages, e.g., PPDDL (Younes et al. 2005) and, our focus, RDDL (Sanner 2010).

Research in Relational MDPs explores ways to represent and construct first-order (generalized) value functions or policies, which can be used directly on a new test instance. Example representations for these include regression trees (Mausam and Weld 2003), decision lists (Fern, Yoon, and Givan 2006), extensions of algebraic decision diagrams (Joshi and Khardon 2011), and linear combinations of basis functions (Guestrin et al. 2003; Sanner and Boutilier 2005). We believe a reason for limited success of RMDP algorithms is the inherent limitation of these representations. We study the use of deep neural models for representing such generalized functions. To the best of our knowledge, we are the first to develop relational planners for domains expressed in RDDL.

Relational Dynamic Decision Language (RDDL): RDDL has been the language of choice for the last three IPPCs. It divides $\mathcal{SP}$ into non-fluent ($\mathcal{NF}$) and fluent ($\mathcal{F}$) symbols. Non-fluents are the state variables that do not change with time in a given instance, but may be different across problem instances. Fluents represent state variables that change with time (due to actions or natural dynamics). RDDL splits an RMDP into two separate files, one for the whole domain (that has types, predicates, transitions, and rewards), and the other for the instance (that has objects, non-fluent values, and fluent values for initial state). RDDL uses additive rewards, total reward is sum of local rewards collected for satisfying different properties in a state. It factors transition function via an underlying DBN semantics. There exist algorithms that convert an RDDL instance into ground DBN (Sanner 2010).

Running Example: We use a simplified Wildfire domain as our example. It has a grid where each cell may have fuel, causing it to burn. The goal is to have the least damage to the grid by either putting out the fire or cutting out the fuel supply. The DBN for the domain is show in Figure 1.

There are two classes, $\mathcal{C} = \{x_{pos}, y_{pos}\}$: x and y coordinate of the grid cell. Domain has two fluent symbols $\mathcal{F} = \{burning, out\text{-}of\text{-}fuel\}$, representing the current burning state and the fuel state of the cell. Both fluent symbols take a cell (x, y) as its arguments. The non-fluents represent costs and topology, $\mathcal{NF} = \{CostTgtBurn, CostNTgtBurn, Neighbour, Target\}$. The non fluent symbol $Neighbour$ takes four arguments (x, y, x', y'), since it defines the topology of the grid. $Target$ has arguments (x, y). $\mathcal{A} = \{put\text{-}out, cut\text{-}out, finisher\}$. First two action symbols take arguments (x, y) – they put out fire and cut out fuel supply at a cell. There is one global action $finisher$, which puts out fire in all the cells simultaneously. Reward (negative) in each time step adds $CostTgtBurn$ for each target cell that is burning and $CostNTgtBurn$ for each non-target cell that is burning. In a problem instance, say there are three objects $O = \{x1, x2, y1\}$. This implies a problem with two cells $(x1, y1)$, and $(x2, y1)$. Say the target cell is $(x1, y1)$ and that these are connected, i.e., $Neighbour(x1, y1, x2, y1) = 1$.

[2] https://github.com/dair-iitd/symnet

2.2 Reinforcement Learning

Reinforcement Learning (RL) refers to planning problems without known transition and rewards, necessitating learning from experience. State of the art approaches for RL are neural, which approximate policy and value functions through deep neural models. We use the Asynchronous Advantage Actor-Critic (A3C) (Mnih et al. 2016) as our underlying RL algorithm. A3C uses two neural networks 1) θ_π to represent the policy (mapping from a state to distribution over actions) and 2) θ_V to estimate the state value (long term discounted reward starting in a state). Policy parameters are optimized to prefer an action that increases a state's advantage function – difference between its value when taking that action and its overall value. Value parameters optimize the MSE loss between observed and predicted long-term rewards.

2.3 Graph Neural Networks

Graph Neural Networks input a graph and learn latent space embeddings for each node, based on the its individual features and the local connectivity structure. Examples include Graph Convolution Networks (GCN) (Kipf and Welling 2017), and Graph Attention Networks (GAT) (Velickovic et al. 2017). We use GAT, which computes a node embedding by using a weighted attention for each of neighbouring nodes. Specifically output node embedding $\overline{v}_i{}' = \sigma(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in N_i}\alpha_{ij}^k \mathbf{W}^k\overline{v}_j)$, where $\overline{v}_i$ is the input feature of node v_i, N_i are its neighbours, $\mathbf{W}^k$ is a trainable weight matrix, k is the multi-head hyperparameter and $\alpha_{ij}^k = softmax_j(f(\mathbf{W}^k\overline{v}_i, \mathbf{W}^k\overline{v}_j))$ is the normalized self attention coefficient for any non-linear function (f), which in our implementation is LeakyReLU (Xu et al. 2015).

2.4 Transfer Learning for Probabilistic Planning

There having been several classical (Taylor and Stone 2009; Sorg and Singh 2009; Atkeson and Schaal 1997) and neural (Parisotto, Ba, and Salakhutdinov 2015; Matiisen et al. 2017; Garnelo, Arulkumaran, and Shanahan 2016; Higgins et al. 2017) approaches for transfer learning in RL. Recent work studies transfer learning for symbolic planning problems, e.g., Groshev et al. (2018) for deterministic planning problems. ASNets, Action-Schema Networks (Toyer et al. 2018; Shen et al. 2019), tackle a problem similar to ours but for goal-oriented subset of PPDDL. While an RDDL instance can be converted automatically into propositional PPDDL, an RDDL domain cannot always be converted into relational PPDDL – hence we cannot directly compare against ASNets. Issakkimuthu et al. (2018) devise a neural framework to learn a policy for (ground) RDDL MDPs from scratch. Their constraint on non-transferability is due to the fixed size of fully connected layers in the neural network. TORPIDO achieves transfer across RDDL problem instances of the same domain (Bajpai, Garg, and Mausam 2018); it can only transfer over *equi-sized* problems due to its fixed size decoder.

Our previous work TRAPSNET is closest to SYMNET, as it can transfer to different-sized instances of an RMDP (Garg, Bajpai, and Mausam 2019). It constructs a graph, which uses single object as nodes, and non-fluent based edge. It encodes each node in embedding space and computes the score for a ground action based on the applied action template, and object embedding. However, TRAPSNET makes the restrictive assumptions that the domains have exactly one binary non-fluent, and all the rest are unary fluents or non-fluents, and each action symbol is parameterized by *exactly* one object. These assumptions don't hold in several RDDL domains.

3 Problem Formulation

Given an RMDP domain $D =< \mathcal{C}, \mathcal{SP}, \mathcal{A}, \mathcal{T}, \mathcal{R}, H, \gamma >$ expressed in RDDL, we wish to learn a generalized policy π^D, which can be applied to all instances of D and maximizes the discounted sum of expected rewards over a finite horizon H. Given a test problem instance $I_t =< O, s_0 >$ from D, this generalized policy can yield an instance-specific policy $\pi^D(I_t) : \mathcal{P}(\mathcal{SP}_O) \rightarrow \mathcal{A}_O$, without any training on I_t. The RMDP learning problem can be seen in terms of multi-task learning over several problem instances in D: given N randomly selected problem instances I_1, I_2, ..., I_N (possibly of different sizes) from D, we wish to learn the weights ϕ of a neural network, such that $\pi^D(I_i; \phi)$ is a good (high-reward) policy for problem instance I_i. A good generalized policy is one which, without training, achieves high reward values on the new instance I_t.

4 The SYMNET Framework

We now present SYMNET's architecture for training a generalized policy for a given RMDP domain. We follow existing research to hypothesize that for any instance of a domain, we can learn a representation of the current state in a latent space and then output a policy in latent space, which is decoded into a ground action. To achieve this, SYMNET uses three modules: (1) problem representation, which constructs an instance graph for every problem instance, (2) representation learning, which learns embeddings for every node in the instance graph, and for the state, and (3) policy decoder, which computes a value for every ground action, outputting a mixed policy for a given state. All parameters of representation learning and policy learning modules are shared across all instances of a domain. SYMNET's full architecture is shown in Figure 2.

4.1 Problem Representation

We follow TRAPSNET, in that we continue the general idea of converting an instance into an instance graph and then learning a graph encoder to handle different-sized domains. However, the main challenge for a general RMDP, one that does not satisfy the restricted assumptions of TRAPSNET, is in defining a coherent graph structure for an instance. The first key question is what should be a node in the instance graph. TRAPSNET's approach was to use a single object as nodes, as all fluents (and actions) in its domains took single objects as arguments. This may not work for a general RMDP since it's fluents and actions may take several objects as arguments. Secondly, how should edges be defined. Edges represent the interaction between nodes. TRAPSNET defined them based on the one binary non-fluent in its domain. A general RMDP may not have any non-fluent symbol or may have many (possibly higher-order) non-fluents.

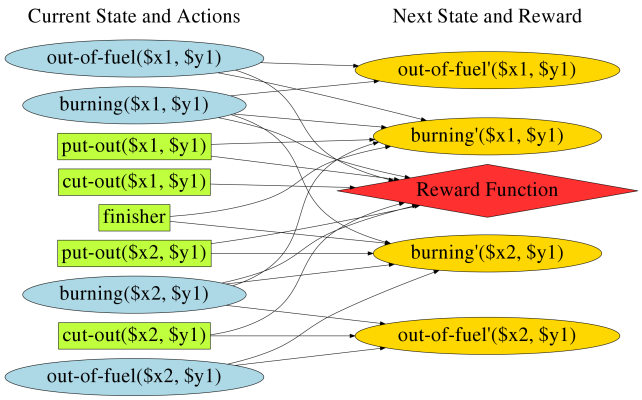


Figure 1: DBN for a modified wildfire problem.

Last but not least, the real domain-independence for SYMNET can be achieved only when it parses an RDDL domain file without any human intervention. This leads to a novel challenge of reconciling multiple different ways in RDDL to express the same domain. In our running example, connectivity structure between cells may be defined using non-fluents $y\text{-}neighbour(y, y')$, $x\text{-}neighbour(x, x')$, or using a quarternary non-fluent $neighbour(x, y, x, y')$. Since both these representations represent the same problem, an ideal desideratum is that the graph construction algorithm leads to the same instance graph in both cases. But, this is a challenge since the corresponding RDDL domains may look very different. While, in general, this problem seems too hard to solve, since it is trying to judge logical equivalence of two domains, SYMNET attempts to achieve the same instance graphs in case the equivalence is within non-fluents.

To solve these problems, we make the observation that dynamics of an RDDL instance ultimately compile to a ground DBN with nodes as state variables (fluent symbols applied on object tuples) and actions (action symbols applied on object tuples).[3] DBN exposes a connectivity structure that determines which state variables and actions directly affect another state variable. It additionally has conditional probability tables (CPTs) for each transition. Figure 1 shows an example of a DBN for our running example instance. Here, left column is for current time step, and right for the next one. The edges represent which state and action variables affect the next state-variable. We note that the ground DBN does not expose non-fluents since its values are fixed, and their dependence can be compiled directly into CPTs.

SYMNET converts a ground DBN to an instance graph. It constructs a node for every unique *object tuple* that appears as an argument in any state variable in the DBN. Moreover, two nodes are connected if the state variables associated with two nodes influence each other in the DBN through some action. This satisfies all our challenges. First, it goes beyond an object as a node, but only defines those nodes that are likely important in the instance. Second, it defines a clear semantics of edges, while maintaining its intuition of "directly influences." Finally, it can handles some variety of non-fluent representations for the same domain. Since the DBN does not even expose non-fluent state variables, and compiles them away, the same instancs encoded with different non-fluent representations often yield yield same ground DBNs and thus the same instance graphs.

Construction of Instance Graph: We now formally describe the conversion of a DBN into a directed instance graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges. G is composed of $\mathcal{K} = |\mathcal{A}| + 1$ disjoint subgraphs $G_j = (V_j, E_j)$. Intuitively, each graph G_j has information about influence of each individual action symbol $\mathbf{a}_j \in \mathcal{A}$. $G_{\mathcal{K}}$ represents the influence of the full set $\mathcal{A}$, and also the natural dynamics. In our example $\mathcal{K} = 4$ since we have three action symbols: put-out, cut-out and finisher.

To describe the formal process, we define three analogous sets: O_f, O_{nf} and O_a. O_f represents the set of all object tuples that act as a valid argument for any fluent symbol. O_{nf} and O_a are analogous sets for non-fluent and action symbols. In our running example, $O_f = \{(x1, y1), (x2, y1)\}$, $O_{nf} = \{(x1, y1), (x2, y1), (x1, y1, x2, y1), (x2, y1, x1, y1)\}$, and $O_a = \{(x1, y1), (x2, y1)\}$. Nodes in the instance graph associate with object tuples. We use o_v to denote the object tuple associated with node v. SYMNET converts a DBN into an instance graph as follows:

1. The distinct object tuples in fluents form the nodes of the graph, i.e. $V_j = \{v | o_v \in O_f\}, \forall j$. For the example, each $V_j =$ different copies of $\{(x1, y1), (x2, y1)\}$.
2. We add an edge between two nodes in G_j if some state variables corresponding to them are connected in the DBN through $\mathbf{a}_j$. Formally, $E_j(u, v) = 1$, if $\exists f, g \in F, \exists o_a \in O_a, j \in \{1, \ldots, |\mathcal{A}|\}$ s.t. the transition dynamics (T^f) for state variable $g'(o_v)$ and action $\mathbf{a}_j(o_a)$ depend on state variable $f(o_u)$ or $f'(o_u)$.For the running example, there is no edge between $(x1, y1)$ and $(x2, y1)$ since cut-out, put-out or finisher's effects on one cell do not depend on any other cell.
3. We add an edge between two nodes in $G_{\mathcal{K}}$ if some state variables corresponding to them are connected in the DBN (possibly through natural dynamics). I.e., $E_{\mathcal{K}}(u, v) = 1$, if $\exists f, g \in F$ s.t. there is an edge from $f(o_u)$ (or $f'(o_u)$) and $g'(o_v)$ in the DBN. For the example, $E_4((x1, y1), (x2, y1)) = 1$ as there is an edge between $burning(x1, y1)$ and $burning'(x2, y1)$ since fire propagates to neighboring cells through natural dyanamics. Similarly, $E_4((x2, y1), (x1, y1)) = 1$.
4. As every node influences itself, self loops are added on each node. $E(v, v) = 1, \forall v \in V$.

For each node $v \in V$, we additionally construct a feature vector ($h(v)$) which consists of fluent feature vector ($h^f(v)$) and non-fluent feature vector ($h^{nf}(v)$), such that $h = concat(h^f, h^{nf})$. The feature vector for all nodes for the same object tuple is the same. The feature vector is constructed as follows:

1. The fluent features for each node are obtained from the state of the problem instance. The values of state variables corresponding to a node are added as feature to that node. Whenever a fluent symbol cannot take a node as an argument, we add zero as the feature for it. Formally,

[3] done automatically using code from https://github.com/ssanner/rddlsim

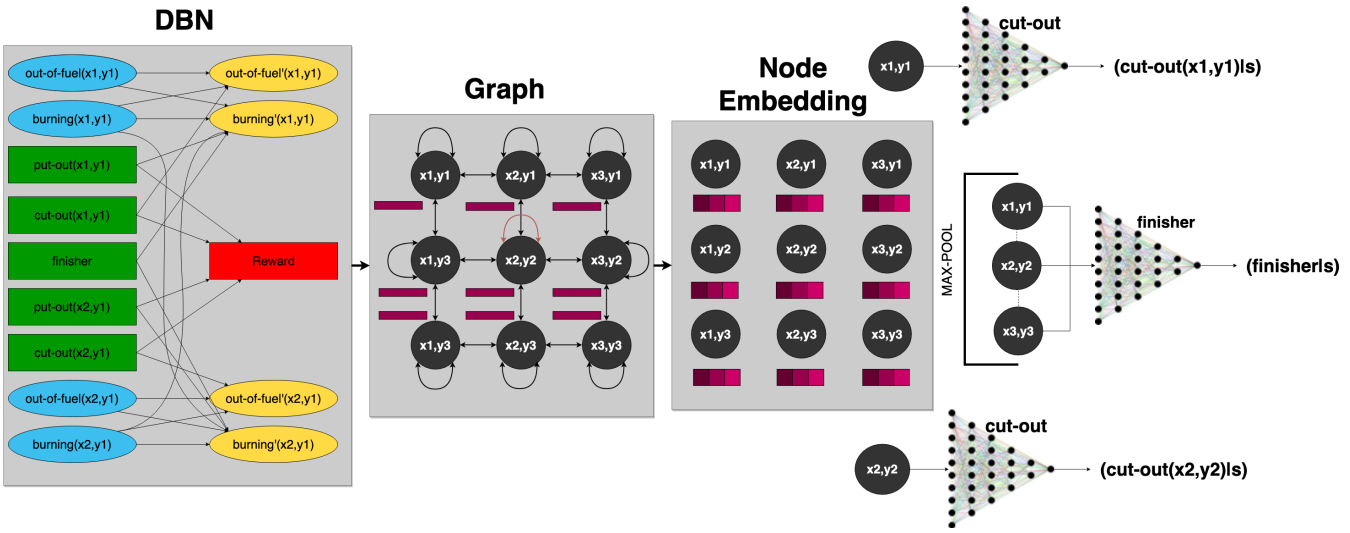


Figure 2: Policy network for SYMNET demonstrated on 2×1 wildfire domain. Fully Connected Network is used in Action Decoder.

$h^f(v)_i = g_i(o_v)$ if $g_i \in \mathcal{F}, v \in V$ and o_v is an argument of g_i, otherwise, $h^f(v)_i = 0, \forall i = 1 \ldots |\mathcal{F}|$. For the running example, we have two state-fluents. Hence, $h^f((x1, y1)) = [burning(x1, y1), out\text{-}of\text{-}fuel(x1, y1)]$.

2. The non-fluent feature vector for each node is obtained from the RDDL file. The values of non-fluents defined on the node, and additionally any unary non-fluents where the argument intersects the node are added as the features for the node. The default value is obtained from the domain file while the specific value (if available) is obtained from the instance file. Formally, $h^{nf}(v)_i = g_i(o_{nf})$ if $g_i \in \mathcal{NF}, v \in V, o_{nf} \in O_{nf}, ((o_v = o_{nf}) \vee (|o_{nf}| = 1 \wedge o_{nf} \subset o_v))$, otherwise, $h^f(v)_i = 0, \forall i = 1 \ldots |\mathcal{NF}|$. In our example, $h^{nf}((x1, y1)) = [target(x1, y1)]$.

We note that the size of feature vector on each node depends on the domain, but is independent of the number of objects in the instance – there are a constant number of feature values per state predicate symbol. This allows variable-sized instances of the same domain to use the same representation.

4.2 Representation Learning

SYMNET runs GAT on the instance graph to obtain node embeddings $\overline{v}$ for each node $v \in V$, It then constructs tuple embedding for each object tuple by concatenating node embeddings of all associated nodes. Formally, let $O_V = \{o_v | v \in V\}$. For $o \in O_V$, the tuple embedding $\overline{o} = \text{concat}(\overline{v})$, over all v s.t. $o_v = o$. SYMNET also computes a state embedding $\overline{s}$ by taking a dimension-wise max over all tuple embeddings, i.e., $\overline{s} = MaxPool_{o \in O_V}(\overline{o})$.

4.3 Policy Decoder

SYMNET maps latent representations $\overline{o}$ and $\overline{s}$ into a state value $V(s)$ (long-term expected discounted reward starting in state s) and mixed policy $\pi(s)$ (probability distribution over all ground actions). This is done using a value decoder and a policy decoder, respectively.

There are several challenges in designing a (generalized) policy decoder. First, the action symbols may take multiple objects as arguments. Second, and more importantly, action symbols may even take those object tuples as arguments that do not correspond to any node in the instance graph. This will happen if an object tuple (in O_a) is not an argument to any fluent symbol, i.e., $\exists o_a$ s.t. $o_a \in O_a \wedge o_a \notin O_f$. Adding these object tuples as nodes in the instance graph may not work, since we won't have any natural features for those nodes.

In response, we design a novel framework for policy and value decoders. The decoders consist of fully connected layers, the input to which are a subset of the tuple embeddings $\overline{o}$. SYMNET uses the following rules to construct decoders:

1. The number of decoders is constant for a given domain and is equal to the number of distinct action symbols ($|\mathcal{A}|$). For the running example, three different decoders for each policy and value decoding are constructed, namely $cut\text{-}out, put\text{-}out$ and $finisher$.
2. The input to a decoder is the state embedding $\overline{s}$ concatenated with embeddings of object tuples corresponding to the state variables affected by the action in the DBN. In running example, $put\text{-}out(x1, y1)$ action takes only the tuple embedding of $(x1, y1)$ as input. However, the number of state-variables being affected by a ground action might vary across instances of the same domain. For example, the $finisher$ action affects all cells. To alleviate this, we use size-independent max pool aggregation over the embeddings of all affected tuple embeddings to create a fixed-sized input.
3. Decoder parameters are specific to action symbols and not to ground actions. In running example, $put\text{-}out(x1, y1)$ will be scored using embedding of $(x1, y1)$; similarly, for $(x2, y1)$. But, both scorings will use a single parameter set specific to $put\text{-}out$.

Table 1: $\alpha_{symnet}(0)$ values of SYMNET. Bold values represent over 90% the score of max performance.

	Instance	5	6	7	8	9	10
Domain	AA	**0.93 ± 0.01**	**0.94 ± 0.01**	**0.94 ± 0.01**	**0.92 ± 0.02**	**0.95 ± 0.03**	**0.91 ± 0.05**
	CT	0.87 ± 0.16	0.78 ± 0.14	**1.00 ± 0.07**	**0.98 ± 0.13**	**0.99 ± 0.04**	**1.00 ± 0.05**
	GOL	**0.96 ± 0.06**	**1.00 ± 0.05**	0.65 ± 0.05	0.83 ± 0.03	**0.95 ± 0.04**	0.64 ± 0.08
	Nav	**0.99 ± 0.01**	**1.00 ± 0.01**	**0.99 ± 0.01**	**1.00 ± 0.01**	**1.00 ± 0.02**	**1.00 ± 0.02**
	ST	**0.91 ± 0.05**	0.84 ± 0.02	0.86 ± 0.05	0.85 ± 0.05	0.81 ± 0.02	0.89 ± 0.03
	Sys	**0.96 ± 0.03**	**0.98 ± 0.02**	**0.98 ± 0.02**	**0.97 ± 0.02**	**0.99 ± 0.01**	**0.96 ± 0.03**
	Tam	**0.92 ± 0.07**	**1.00 ± 0.12**	**0.98 ± 0.06**	**1.00 ± 0.12**	**1.00 ± 0.12**	**0.95 ± 0.06**
	Tra	0.85 ± 0.18	**0.93 ± 0.06**	0.88 ± 0.21	0.74 ± 0.17	**0.94 ± 0.12**	0.87 ± 0.13
	Wild	**0.99 ± 0.01**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.01**	**1.00 ± 0.01**

4. The policy decoder computes scores of all ground actions, which are normalized using softmax to output the final policy in a state. For I_t, the highest probability action is selected as the final action.
5. All value outputs are summed to give the final value for that state. This modeling choice reflects the additive reward aspect of many RDDL domains.

4.4 Learning

While construction of SYMNET architecture is heavily dependent on the RDDL domain and instance files, actual training is done via model-free reinforcement learning approach of A3C (Mnih et al. 2016). RL learns from interactions with environment – SYMNET simulates the environment using RDDL-specified dynamics. Use of model-based planning algorithms for this purpose is left as future work. We formulate training of SYMNET as a multi-task learning problem (see Section 3), so that it generalizes well and does not overfit on any one problem instance. The parameters for the state encoder, policy decoder, and value decoder are learned using updates similar to that in A3C. SYMNET's loss function for the policy and value network is the same as that in the A3C paper (summed over the multi-task problem instances).

As constructed, SYMNET's number of parameters is independent of the size of the problem instance. Hence, the same network can be used for problem instances of any size. After the learning is completed, the network represents a generalized policy (or value), since it can be directly used on a new problem instance to compute policy in a single forward pass.

5 Experiments

Our goal is to estimate the effectiveness of SYMNET out-of-the-box policy for a new problem in a domain. Unfortunately, there are no available transfer algorithms for general RDDL RMDPs. So, we first compare it against a random policy, because that is the best we can do currently with no time to train. To further understand the overall quality of the generalized policy, we also compare it against several neural models that train from scratch on the test instance. We also compare it against state-of-the-art online planner PROST (Keller and Eyerich 2012).

5.1 Domains and Experimental Setting

We show all our results on nine RDDL domains used IPPC 2014: Academic Advising (AA), Crossing Traffic (CT), Game of Life (GOL), Navigation (NAV), Skill Teaching (ST), Sysadmin (Sys), Tamarisk (Tam), Traffic (Tra), and Wildfire (Wild). We describe the domains, number of state fluents & non-fluents, action fluents in supplementary material. The RL agent is trained to learn the generalized policy on smaller sized instances. We use IPPC problem instances 1, 2, and 3 of each domain for the multi-task training of SYMNET network. A3C uses a discounted parameter of 0.99. In the spirit of domain-independent planning, we use the *same* hyperparameters for each domain. The embedding module for GAT uses a neighborhood of 1 and an output feature size of 6. We then use a fully connected layer of output 20 dimensions to get an embedding from each of the tuple embedding outputs by GAT. All layers use a leaky ReLU activation and a learning rate of 10^{-3}. We train the network using RMSProp (Ruder 2016) on a single Nvidia K40 GPU. SYMNET is trained for each domain for twelve hours (4 hours for each instance).

5.2 Comparison Algorithms and Metrics

As there does not exist any previous method for learning over Relational RDDL MDPs, we can only compare against a random policy. However, this experiment can only show the difference from a random policy, but cannot evaluate the overall goodness of the generalized policy. For that, we compare against several (potentially upper bound) policies that are not directly comparable to SYMNET in their experimental settings. For our first such experiment, we use TORPIDO as the state-of-the-art deep reactive policy. Note that we do not use their transfer method, but train the network from scratch on the problem instance. This is because it can only transfer across equi-sized instances. Still, it is an upper bound as TORPIDO trained on the test instance is compared against SYMNET trained on other smaller instances, but not the test instance. Similarly, we also compare against SYMNET architecture itself, trained from scratch on the test instance (named SYMNET-s). The main difference between TORPIDO and SYMNET architectures is that TORPIDO has a much higher capacity since it models each ground action explicitly. On three domains where TRAPSNET is applicable, we also compare against TRAPSNET policies out of the box. Finally, we also compare against the state-of-the-art *online* planner,

Table 2: Comparison of SYMNET against SYMNET-s (SYM) architecture trained from scratch and TORPIDO (TOR) architecture trained from scratch. We compare out-of-the-box SYMNET to others after 12 hours of training. INF is used when SYM or TOR achieved minimum possible reward and hence SYMNET was infinitely better.

Domain	SYM	TOR	Domain	SYM	TOR	Domain	SYM	TOR	Domain	SYM	TOR
AA 5	1.09	0.99	GOL 5	**1.35**	**1.49**	ST 5	1.11	0.94	Tam 5	**INF**	**2.33**
AA 6	1.78	0.95	GOL 6	**1.57**	**1.69**	ST 6	1.21	0.90	Tam 6	**27.71**	**8.13**
AA 7	1.21	0.98	GOL 7	1.08	0.76	ST 7	1.10	0.87	Tam 7	**17.81**	**4.83**
AA 8	1.31	0.97	GOL 8	2.22	0.87	ST 8	1.14	0.90	Tam 8	**2.74**	**15.56**
AA 9	1.39	0.95	GOL 9	**1.86**	**1.31**	ST 9	1.13	0.81	Tam 9	**24.94**	**13.07**
AA 10	1.32	0.93	GOL 10	1.25	0.68	ST 10	1.30	0.95	Tam 10	**2.35**	**7.99**
CT 5	**1.34**	**1.39**	Nav 5	**10.84**	**INF**	Sys 5	**1.03**	**2.89**	Tra 5	1.78	0.86
CT 6	**INF**	**1.56**	Nav 6	**INF**	**INF**	Sys 6	**1.33**	**1.20**	Tra 6	**1.56**	**1.39**
CT 7	**1.13**	**1.12**	Nav 7	**INF**	**INF**	Sys 7	**1.56**	**2.45**	Tra 7	**3.28**	**1.13**
CT 8	**1.55**	**1.23**	Nav 8	**INF**	**INF**	Sys 8	**1.46**	**1.60**	Tra 8	1.13	0.81
CT 9	**1.35**	**1.16**	Nav 9	**INF**	**INF**	Sys 9	**1.38**	**1.17**	Tra 9	**2.50**	**1.08**
CT 10	**1.22**	**4.99**	Nav 10	**INF**	**INF**	Sys 10	**1.18**	**1.50**	Tra 10	**1.53**	**1.86**
Wild 5	**1.03**	**1.13**	Wild 7	**1.03**	**1.13**	Wild 9	**1.01**	**13.14**			
Wild 6	**1.01**	**1.01**	Wild 8	**1.00**	**1.09**	Wild 10	**34.80**	**11.19**			

Table 3: Comparison of TRAPSNET with SYMNET on three domains as published in (Garg, Bajpai, and Mausam 2019). Label: AA - Academic Advising, GOL - Game Of Life, Sys - Sysadmin

	Instance	5	6	7	8	9	10
Domain	AA	**1.12**	**1.17**	**1.12**	**1.27**	**1.26**	**1.40**
	GOL	0.96	**1.04**	0.69	**1.00**	0.97	**1.50**
	Sys	**1.01**	**1.55**	**1.33**	**1.39**	**1.21**	**1.17**

PROST.

After training algorithm alg for t hours, we simulate its output policy 200 times (for H steps each) from the start state. We average the discounted rewards to estimate the expected long term discounted reward of that policy, denoted by $V_{alg}(t)$. To be able to compare across domains and problems and reward ranges, we report a normalized metric $\alpha_{alg}(t) = \frac{V_{alg}(t)-V_{min}}{V_{max}-V_{min}}$ where V_{min} and V_{max} are the minimum, and the maximum expected discounted rewards obtained at any time by any of the four comparison algorithms on a given instance. This number lies between 0 and 1, with 1 being the best-found reward, and 0 being the random policy's reward. All algorithms are trained independently 5 times and the average result is reported. During training from scratch, all networks start with a random policy and hence have their $\alpha(0)$ values as 0. However, that is not true for SYMNET as it is pre-trained on the domain. To compare against other training approaches directly, we compute $\beta_{alg}(t) = \frac{\alpha_{symnet}(0)}{\alpha_{alg}(t)}$. A value higher than 1 suggests that SYMNET out-of-the-box outperformed alg trained for t hours, and less than 1 implies SYMNET performed worse.

5.3 Results

Comparison against Random Policy: We report the values of $\alpha_{symnet}(0)$ in Table 1. Since the random policy is 0, we notice that on all six problem instances from the nine domains, SYMNET performs enormously better than random. We highlight the instances where our method achieves over 90% of the max reward obtained by any algorithm for that instance. We see that SYMNET with no training achieves over 90% the max reward on 40 instances and over 80% in 50 out of 54 instances. We also show that our method performs the best out-of-the-box in 28 instances. This is our main result, and it highlights that SYMNET takes a major leap towards the goal of computing generalized policies for the whole RMDP domain, and can work on a new instance out of the box.

Comparison against Training from Scratch: We now compare SYMNET against the expected discounted rewards obtained by TORPIDO and SYMNET-s, when they are trained from scratch for 12 hours on the test problem. We note that these numbers are not directly comparable, since in one case, the model has been trained on other instances of the domain, but not trained on the test problem at all, and in the other case the models are trained from scratch on the test. That said, this comparison is likely a good indicator of the absolute performance of SYMNET.

Table 2 reports the values for $\beta_{torpido}(12)$ and $\beta_{symnet-s}(12)$. We notice that, surprisingly, SYMNET policy with no training is better than both methods on several instances. Against SYMNET trained from scratch, it is better on all instances, although its edge over TORPIDO is limited to 37 out of 54. We hypothesize that this excellent performance is due to the multi-task learning aspect of SYMNET, where it is able to reach some generalized policy of a domain that is not found on the specific instance even after training for 12 hours.

In 17 out of 54 instances, SYMNET lags behind TORPIDO , which is not surprising, since TORPIDO has much higher capacity, as discussed earlier. We also notice that the performance of TORPIDO is no better than random for

Table 4: Comparison of PROST with SYMNET. INF is used when PROST returned a policy equal to or worse than a random policy.

	Domain	AA	CT	GOL	Nav	ST	Sys	Tam	Tra	Wild
Instance	5	**2.13**	0.76	0.48	**1.16**	0.95	**1.13**	0.61	0.60	**1.39**
	6	**2.14**	0.44	0.57	**1.87**	0.86	**1.24**	0.96	0.65	**INF**
	7	**2.18**	0.62	0.33	**6.42**	0.86	**1.13**	0.70	0.61	**INF**
	8	**1.79**	0.37	0.39	**45.46**	0.90	**1.50**	0.79	0.51	**INF**
	9	**1.46**	0.74	0.44	**101.23**	0.78	**1.21**	0.83	0.75	**INF**
	10	**1.46**	0.37	0.30	**INF**	0.93	**1.42**	0.84	0.64	**1.49**

Navigation. We attribute this to the sparse and late reward obtained in large instances of this domain, which makes it difficult for TORPIDO to learn a good policy. Because of the late rewards, TORPIDO is not able to reach the goal state at all in 12 hours of training, and hence is not able to improve on the random policy. SYMNET trains well on small instances where path to goal is short and generalizes well. In GOL, SYMNET performs worse, because the nature of policy changes significantly in large instances (e.g. requiring new patterns to survive) which cannot be learned in smaller instances at all.

Comparison against TRAPSNET: While TRAPSNET is not applicable in many RMDPs, still, we can compare it with SYMNET on some domains. We compare these on three domains that follow the unary fluents and binary non-fluents constraint: Academic Advising, Game of Life, and SysAdmin. We report $\beta_{trapsnet}(0)$ in Table 3. It shows that SYMNET outperforms TRAPSNET on 15 out of 18 instances, is comparable on 2 instances and worse on 1 instance. We attribute the success of SYMNET over TRAPSNET to the action-symbol specific graphs (G_j), which likely help learn better action dependencies in the embeddings.

Comparison against ASNets: Even after significant efforts, we were not able to compare against ASNets, which solves a similar problem for PPDDL domains. Converting an RDDL domain to PPDDL enumerates all the ground state-variables and loses the RMDP structure. This leads to different domain files for different instances for the same problem domain, due to which ASNets is unable to train. We also tried writing a domain file manually for a few domains, but were not successful due to the unavailability of floating non-fluent values, and due to non-additive reward structure in PPDDL.

Comparison against PROST: Finally, we compare against PROST. PROST is a state-of-the-art *online* planner, i.e., it performs interleaved planning and execution, as it builds a new search tree before taking every action, based on the specific state reached. On the other hand, SYMNET outputs an offline policy, which does not need much computation for deciding the next action. Offline and online policies are two very different settings, and these results are *not directly comparable*. Nonetheless, we report $\frac{V_{symnet}(0)-V_{min}}{V_{prost}-V_{min}}$. The code of PROST is obtained from the official repository and we use its default settings for this comparison.[4]

We compare our policy with PROST on all 9 domains, shown in Table 4. We see that on four domains SYMNET achieves a much better performance than PROST. This is rather surprising to us that even after substantial lookahead from the current state, PROST is still not able to compute a good policy. For example, in both Navigation and Wildfire, the rewards are sparse and distant, and PROST is often unable to reach the goal in its planning horizon. In other five domains, PROST is substantially better than SYMNET. This suggests that SYMNET policies are not close to optimal, and further research is needed for making them even stronger. This also points to the future possibility of applying a combination of SYMNET and PROST for the offline setting, not unlike the use of Monte-Carlo Tree Search with deep neural networks in AlphaGo (Silver et al. 2016).

Overall, we find that SYMNET's generalized policies out-of-the-box are enormously better than random, and can frequently beat other deep neural models trained from scratch on the test instance. However, comparison with PROST suggests that SYMNET policies are not close to optimal and further research is needed to make them even better.

6 Conclusion and Future Work

We present the first neural-method for obtaining a generalized policy for Relational MDPs represented in RDDL. Our method, named SYMNET, converts an RDDL problem instance into an instance graph, on which a graph neural network computes state embeddings and embeddings for important object tuples. These are then decoded into scores for each ground action. All parameters are tied and size-invariant such that the same model can work on problems of varying sizes. In our experiments, we train SYMNET on small problems of a domain and test them on larger problems to find that they out-of-the-box perform hugely better than random. Even when compared against training deep reactive policies from scratch, SYMNET without training perform better or at par in over half the problem instances.

Our work is an attempt to revive the thread on Relational MDPs and the attractive vision of generalized policies for a domain. However, ours is only one of the first steps. Further investigation is needed to assess how far are SYMNET's generalized policies from optimal. We strongly believe that there may be even better architectures that could learn near-optimal generalized policies, and the need for retraining or interleaving planning and execution could be rendered unnecessary. We release all our software for use by the research community at https://github.com/dair-iitd/symnet.

[4] https://github.com/prost-planner/prost

Acknowledgements

We would like to thank Scott Sanner for an extremely insightful discussion on DBNs, which enabled us to work out a solution to convert an RMDP into a graph using DBNs. We also thank Vishal Sharma, Gobind Singh, Parag Singla and the anonymous reviewers for their comments on various drafts of the paper. This work is supported by grants from Google, Bloomberg, IBM and 1MG, Jai Gupta Chair Fellowship, and a Visvesvaraya faculty award by Govt. of India. We thank Microsoft Azure sponsorships, and the IIT Delhi HPC facility for computational resources.

References

Atkeson, C. G., and Schaal, S. 1997. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997*, 12–20.

Bajpai, A.; Garg, S.; and Mausam. 2018. Transfer of deep reactive policies for mdp planning. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc. 10965–10975.

Bellman, R. 1957. A Markovian Decision Process. *Indiana University Mathematics Journal*.

Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic dynamic programming for first-order mdps. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, 690–700.

Fern, A.; Yoon, S. W.; and Givan, R. 2006. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *J. Artif. Intell. Res.* 25:75–118.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size independent neural transfer for rddl planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 631–636.

Garnelo, M.; Arulkumaran, K.; and Shanahan, M. 2016. Towards deep symbolic reinforcement learning. *CoRR* abs/1609.05518.

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *ICAPS*.

Grzes, M.; Hoey, J.; and Sanner, S. 2014. International Probabilistic Planning Competition (IPPC) 2014. In *ICAPS*.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing plans to new environments in relational mdps. In *IJCAI*, 1003–1010.

Higgins, I.; Pal, A.; Rusu, A. A.; Matthey, L.; Burgess, C.; Pritzel, A.; Botvinick, M.; Blundell, C.; and Lerchner, A. 2017. DARLA: improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 1480–1490.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training deep reactive policies for probabilistic planning problems. In *ICAPS*.

Joshi, S., and Khardon, R. 2011. Probabilistic relational planning with first order decision diagrams. *Journal of Artificial Intelligence Research* 41:231–266.

Keller, T., and Eyerich, P. 2012. PROST: probabilistic planning based on UCT. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Kolobov, A.; Mausam; and Weld, D. S. 2012. A theory of goal-oriented mdps with dead ends. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*, 438–447.

Matiisen, T.; Oliver, A.; Cohen, T.; and Schulman, J. 2017. Teacher-student curriculum learning. *CoRR* abs/1707.00183.

Mausam, and Weld, D. S. 2003. Solving relational MDPs with first-order machine learning. In *ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 1928–1937.

Parisotto, E.; Ba, L. J.; and Salakhutdinov, R. 2015. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR* abs/1511.06342.

Puterman, M. 1994. *Markov Decision Processes*. John Wiley & Sons, Inc.

Ruder, S. 2016. An overview of gradient descent optimization algorithms.

Sanner, S., and Boutilier, C. 2005. Approximate linear programming for first-order mdps. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, 509–517.

Sanner, S., and Boutilier, C. 2009. Practical solution techniques for first-order mdps. *Artif. Intell.* 173(5-6):748–788.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Shen, W.; Trevizan, F.; Toyer, S.; Thiébaux, S.; and Xie, L. 2019. Guiding Search with Generalized Policies for Probabilistic Planning. In *Proc. of 12th Annual Symp. on Combinatorial Search (SoCS)*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Sorg, J., and Singh, S. P. 2009. Transfer via soft homomorphisms. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, 741–748.

Taylor, M. E., and Stone, P. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10:1633–1685.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2017. Graph attention networks. *CoRR* abs/1710.10903.

Xu, B.; Wang, N.; Chen, T.; and Li, M. 2015. Empirical evaluation of rectified activations in convolutional network. *CoRR* abs/1505.00853.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res.* 24:851–887.

Appendix

A Domain Description

We describe the details of the domains presented in the IPPC 2011 and IPPC 2014. The statistics for state fluents ($\mathcal{F}$), non-fluents ($\mathcal{NF}$) and Action ($\mathcal{A}$) for all the domains are show in the Table 5 and Table 6. UP represent $\mathcal{F}$, $\mathcal{NF}$ and $\mathcal{A}$ without parameters, Unary represents $\mathcal{F}$, $\mathcal{NF}$ and $\mathcal{A}$ with a single parameter and multiple represents $\mathcal{F}$, $\mathcal{NF}$ and $\mathcal{A}$ with more than one parameter. Table 7 lists the instance specific number of objects, state variables and action variables for the domain. The domains $1, 2, 3$ are used for training, 4 for validation and $5, 6, 7, 8, 9, 10$ for testing.

Academic Advising

The academic advising domain represents a student at a university trying to complete his/her degree. Some courses are required to be completed to obtain the final degree. Each course is either a basic course or may have prerequisites. The probability of passing a course depends on the number of prerequisites completed (a fixed probability if no prerequisite). The goal is to complete the degree as soon as possible.

Crossing Traffic

Crossing Traffic is represents a robot in a grid, with obstacles at a random grid cell at any time. The obstacles (car) start at any cell randomly and move left. The robot aims to plan its path from the starting grid cell to the goal cell while avoiding obstacles.

Game of Life

Game of Life domain is represented as a grid where each cell can either be dead or alive. The goal is to keep as many cells alive as possible. The probability of cell death depends on the number of neighbors alive at a particular time, which is non-linear in the number of neighbors alive.

Navigation

Navigation represents a robot in a grid world where the aim is to reach a goal cell as quickly as possible. The probability of the robot dying in a particular cell is different, which is specified in the instance file.

Skill Teaching

Skill Teaching domain represents a teacher trying to teach a skill to students. Each student has a mastery level in a particular skill. Some skills have pre-conditions, which increase the probability of learning a particular skill. The skill is taught using either hints or multiple-choice questions. The goal is to answer as many questions as possible by the student by learning the required skill.

Sysadmin

Sysadmin domain represents computers connected in a network. The probability of a computer shutting down on its own depends on the number of turned-on neighboring computers. The agent can either turn on a computer or leave it as it is. The goal is to maximize the number of computers at a particular time.

Table 5: The statistics related to the domains listing the number of UP (Un-Paramataried), Unary and Muiltiple Action ($\mathcal{A}$) for each domain.

Domain	UP-$\mathcal{A}$	Unary-$\mathcal{A}$	Multiple-$\mathcal{A}$
Academic Advising	0	1	0
Crossing Traffic	4	0	0
Game of Life	0	0	1
Navigation	4	0	0
Skill Teaching	0	2	0
Sysadmin	0	1	0
Tamarisk	0	2	0
Traffic	0	1	0
Wildfire	0	0	2

Tamarisk

Tamarisk domain represents invasive species of plants (Tamarisk) trying to take over native plant species. The plants spread in any direction and try to destroy the native plant species. The agent can either eradicate Tamarisk in a cell or restore the native plant species, each having a different reward. The goal is to minimize the cost of eradication and restoration of the native plant species.

Traffic

Traffic domain models the traffic on the road with roads connecting at various intersections. Each road intersection has two traffic light signals combinations of which yield different traffic movement. The agent aims to control the traffic signal (only on the forward sequence) to control the traffic.

Wildfire

The wildfire domain represents a forest catching fire. The direction of fire spreading depends on the direction of the wind and also the type of fuel at that point (e.g., grass or wood, etc.). The agent can either choose to put down the fire or cut off the fuel even before the fire happens. The goal is to prevent as many cells as possible, and more reward is provided to protect high priority cells.

B Variation of $\alpha_{\text{SymNet}}(0)$ with neighbourhood

To inspect the importance of the neighborhood information in learning a generalized policy for the domains, we perform the study of the neighborhood parameter variation. In the Figure 3, we show the variation of $\alpha_{\text{SymNet}}(0)$ with neighbourhood. From the Figure, we observe that message passing for the neighborhood of size 1 yields the best results for most domains, and hence we reported the results with neighborhood 1 in the main paper. In general, we observe that the value of $\alpha_{\text{SymNet}}(0)$ first increases and then decreases.

For most instances, the $\alpha_{\text{SymNet}}(0)$ is less for neighborhood 0 compared to neighborhood 1, showing that the information regarding the neighbors is necessary for learning a better policy. For example, in domain academic advising,

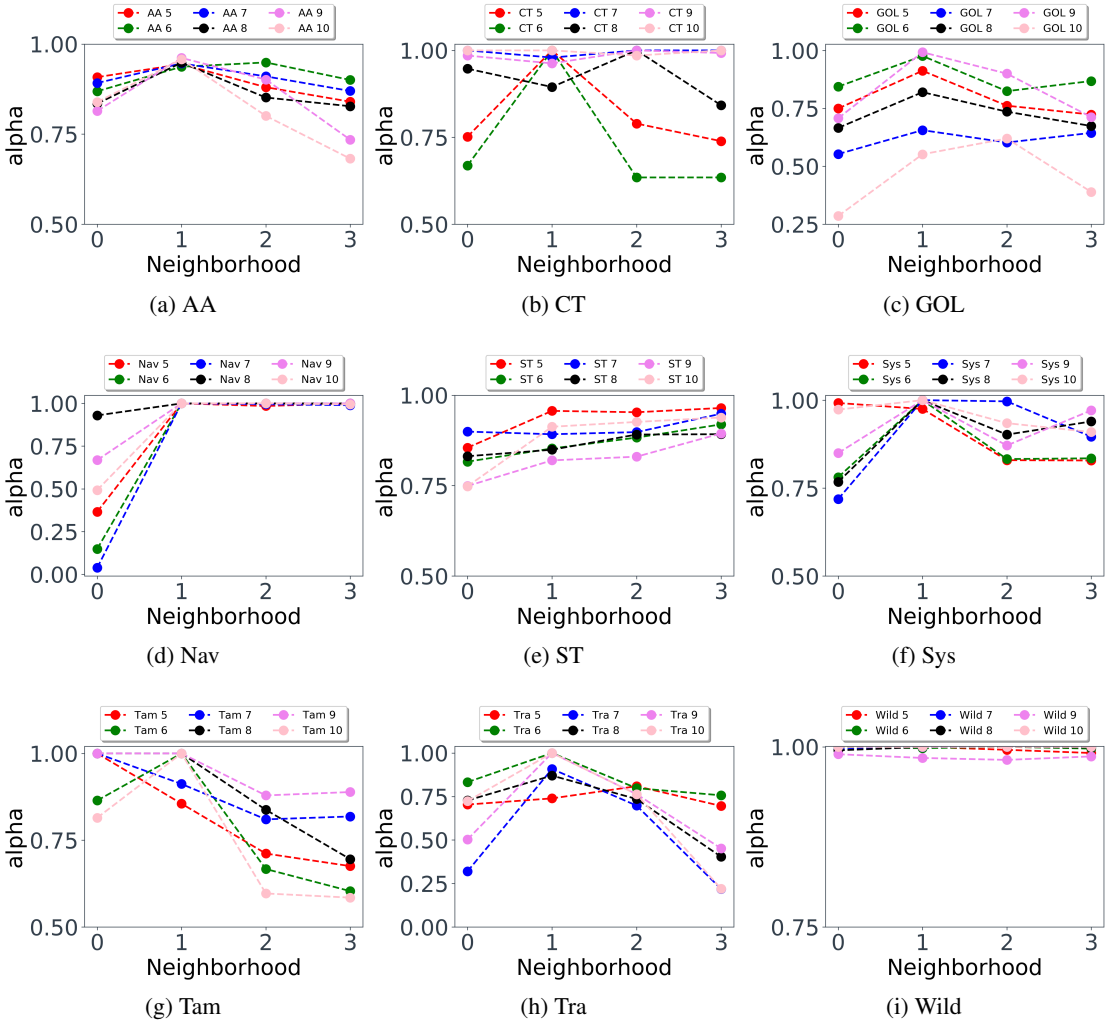
Table 6: The statistics related to the domains listing the number of UP (Un-Paramataried), Unary and Multiple State Fluents ($\mathcal{F}$) and Non-Fluents ($\mathcal{NF}$) for each domain.

Domain	UP-$\mathcal{F}$	UP-$\mathcal{NF}$	Unary-$\mathcal{F}$	Unary-$\mathcal{NF}$	Multiple-$\mathcal{F}$	Multiple-$\mathcal{NF}$
Academic Advising	0	1	2	5	0	1
Crossing Traffic	0	1	0	4	2	5
Game of Life	0	0	0	0	1	2
Navigation	0	0	0	4	1	6
Skill Teaching	0	0	6	7	0	1
Sysadmin	0	2	1	0	0	1
Tamarisk	0	17	2	0	0	2
Traffic	0	0	3	3	0	3
Wildfire	0	4	0	0	2	2

the neighborhood 1 aggregates information about the prerequisites for the courses and then prioritizes the courses to take. A similar trend is observed in domain skill teaching, where the information about the pre-condition for the skill plays an important role in learning the skills. For some domains like navigation, neighborhood information is absolutely critical for planning the next move which can be observed from very low values of $\alpha_{\text{SymNet}}(0)$ from Figure 3(d). Other domains like wildfire are not affected a lot by neighborhood a lot. This is because the margin between the minimum and maximum rewards is large, and the generalized policy outputs rewards close to the maximum value, which decreases the variation in the value of $\alpha_{\text{SymNet}}(0)$. As we increase the value of neighborhood to 2 and 3, the value of $\alpha_{\text{SymNet}}(0)$ tends to fall down for most instances. We hypothesize that the agent overfits to instance-specific policies for the instances it is trained on and hence fails to generalize.

Table 7: The statistics related to the domain instances listing the number of Objects, State Variables and Action Variables for all the instances of the domains. We also give the number of node and edges in the instance graph $G_{\mathcal{K}}$ (i.e. the graph constructed from the DBN). Domain $1, 2, 3$ are used for training, 4 for validation and $5, 6, 7, 8, 9, 10$ for testing.

Domain	#Objects	#State Vars	#Action Vars	#Nodes	#Edges	Domain	#Objects	#State Vars	#Action Vars	#Nodes	#Edges
AA 1	10	20	11	10	26	ST 1	2	12	5	2	4
AA 2	10	20	11	10	32	ST 2	2	12	5	2	4
AA 3	15	30	16	15	38	ST 3	4	24	9	4	16
AA 4	15	30	16	15	51	ST 4	4	24	9	4	16
AA 5	20	40	21	20	53	ST 5	6	36	13	6	36
AA 6	20	40	21	20	63	ST 6	6	36	13	6	36
AA 7	25	50	26	25	63	ST 7	7	42	15	7	49
AA 8	25	50	26	25	93	ST 8	7	42	15	7	49
AA 9	30	60	31	30	75	ST 9	8	48	17	8	64
AA 10	30	60	31	30	101	ST 10	8	48	17	8	64
CT 1	9	12	5	9	39	Sys 1	10	10	11	10	24
CT 2	9	12	5	9	39	Sys 2	10	10	11	10	38
CT 3	16	24	5	16	77	Sys 3	20	20	21	20	58
CT 4	16	24	5	16	77	Sys 4	20	20	21	20	77
CT 5	25	40	5	25	127	Sys 5	30	30	31	30	86
CT 6	25	40	5	25	127	Sys 6	30	30	31	30	111
CT 7	36	60	5	36	189	Sys 7	40	40	41	40	118
CT 8	36	60	5	36	189	Sys 8	40	40	41	40	156
CT 9	49	84	5	49	263	Sys 9	50	50	51	50	150
CT 10	49	84	5	49	263	Sys 10	50	50	51	50	196
GOL 1	9	9	10	9	49	Tam 1	12	16	9	8	40
GOL 2	9	9	10	9	49	Tam 2	16	24	9	12	90
GOL 3	9	9	10	9	49	Tam 3	15	20	11	10	52
GOL 4	16	16	17	16	100	Tam 4	20	30	11	15	117
GOL 5	16	16	17	16	100	Tam 5	18	24	13	12	64
GOL 6	16	16	17	16	100	Tam 6	24	36	13	18	144
GOL 7	25	25	26	25	169	Tam 7	21	28	15	14	76
GOL 8	25	25	26	25	169	Tam 8	28	42	15	21	171
GOL 9	25	25	26	25	169	Tam 9	24	32	17	16	88
GOL 10	30	30	31	30	196	Tam 10	32	48	17	24	198
Nav 1	12	12	5	12	55	Tra 1	28	32	5	28	84
Nav 2	15	15	5	15	71	Tra 2	28	32	5	28	84
Nav 3	20	20	5	20	99	Tra 3	40	44	5	40	120
Nav 4	30	30	5	30	155	Tra 4	40	44	5	40	120
Nav 5	30	30	5	30	151	Tra 5	52	56	5	52	156
Nav 6	40	40	5	40	209	Tra 6	52	56	5	52	156
Nav 7	50	50	5	50	267	Tra 7	64	68	5	64	192
Nav 8	60	60	5	60	311	Tra 8	64	68	5	64	192
Nav 9	80	80	5	80	429	Tra 9	76	80	5	76	228
Nav 10	100	100	5	100	547	Tra 10	76	80	5	76	228
Wild 1	9	18	19	9	48	Wild 6	25	50	51	25	156
Wild 2	9	18	19	9	43	Wild 7	30	60	61	30	180
Wild 3	16	32	33	16	87	Wild 8	30	60	61	30	172
Wild 4	16	32	33	16	97	Wild 9	36	72	73	36	238
Wild 5	25	50	51	25	156	Wild 10	36	72	73	36	230



(a) AA (b) CT (c) GOL

(d) Nav (e) ST (f) Sys

(g) Tam (h) Tra (i) Wild

Figure 3: Variation of $\alpha_{\text{SYMNET}}(0)$ with neighbourhood. [Larger is better]

Safe Learning of Lifted Action Models

Brendan Juba[1], Hai S. Le[1], Roni Stern[2,3]
[1]Washington University in St. Louis, 1 Brookings Dr., St. Louis, MO, USA, {bjuba,hsle}@wustl.edu
[2]Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA, USA, rstern@parc.com
[3]Ben Gurion University of the Negev, Be'er Sheva, Israel, sternron@post.bgu.ac.il

Abstract

Creating a domain model, even for classical, domain-independent planning, is a notoriously hard knowledge-engineering task. A natural approach to solve this problem is to use learning, but learning approaches frequently do not provide guarantees of safety: variously, actions may fail or may not lead to the desired outcome. In some domains such failures are not acceptable, due to the cost of failure or inability to replan online after failure. In such settings, all learning must be done offline, based on some observations collected, e.g., by some other agents or a human. Through this learning, the task is to generate a plan that is guaranteed to be successful. This is called the model-free planning problem. Prior work proposed an algorithm for solving the model-free planning problem in classical planning. However, they were limited to learning grounded domains, and thus they could not scale. We generalize this prior work and propose the first safe model-free planning algorithm for lifted domains. We prove the correctness of our approach, and provide a statistical analysis showing that the number of trajectories needed to solve future problems with high probability is linear in the potential size of the domain model. We also present experiments demonstrating that our approach scales favorably in practice.

Introduction

In classical domain-independent planning, a *domain model* is a model of the environment and how the acting agent can interact with it. The domain model is given in a formal planning description language such as STRIPS (Fikes and Nilsson 1971) or the Planning Domain Definition Language (PDDL) (McDermott et al. 1998). Domain-independent planning algorithms (planners) use the domain model to generate a plan for achieving a given goal condition from a given initial state. Creating a domain model, however, is a notoriously hard knowledge-engineering task.

To overcome this modeling problem, a variety of learning methods have been proposed. The best-known formulation is Reinforcement Learning (RL). In RL, an agent collects observations about the world by performing actions and observing their outcomes. The RL agent then uses these observations to decide how to act in the future. RL techniques have proven to be effective in a variety of domains, especially for low-level control tasks. However, RL generally does not consider the possibility of "failing," except insofar as the reward is sub-optimal. Similarly, most offline approaches that aim to learn a world model from past observations allow generating failing actions (Amir and Chang 2008).[1] In some domains, this is not a problem and the agent simply incorporates such experiences and updates its internal model to improve future executions. In other domains, however, execution failure *must* be avoided and only *safe* actions are allowed. This occurs when execution failure is too costly, or the agent cannot replan due to limited computational capabilities. The problem of finding a *safe* plan, i.e., a plan that will not fail, without having a domain model, is called *safe model-free planning* (Stern and Juba 2017). Instead of a domain model, in safe model-free planning the planning agent is given a set of *trajectories* from plans that were executed in the past in the same domain (e.g., by a different agent or a human).

Stern and Juba (2017) proposed a sound algorithm for safe model-free planning, i.e., an algorithm that generates plans that do not fail, provided that the environment is actually captured by a (grounded) STRIPS model. However, their algorithm is not complete, i.e., it may not return a plan for a solvable planning problem. Nevertheless, they bounded the probability of encountering such problems given a number of trajectories quasi-linear in the number of actions.

Their positive result is limited to *grounded* domain models, that is, domains that are not defined by *lifted*, i.e., parameterized, actions and fluents. It is possible to generate a grounded domain model from a given lifted domain model and problem. But, the size of the resulting grounded domain model can be arbitrarily larger than the lifted domain model. In particular, a single lifted action can yield a number of grounded actions that grow polynomially with the number of objects in the domain, with the number of parameters of the lifted action as its exponent. This significantly limits the applicability of Stern and Juba's algorithm.

In this work, we generalize their approach and propose an algorithm that efficiently solves safe model-free planning problems for lifted domains. The key component of this approach is an algorithm that learns a *safe action model*, which is a model of the agent's possible actions that is consistent with the underlying, unknown, domain model. We call this

[1]They also did not consider lifted action models as we do.

algorithm *Safe Action Model (SAM)* Learning.

Two versions of SAM learning are presented. The first may be used when the mapping from lifted fluents to grounded fluents can be inverted. We prove that this version is sound, and when the actions and fluents have bounded arity, we can guarantee that the action model is sufficient with high probability after observing a number of trajectories that is linear in the possible size of the lifted model. Importantly, the number of trajectories needed depends only on the size of this lifted model, and is independent of the number of objects in the domain, in contrast to Stern and Juba's algorithm. We also observed efficient learning experimentally on small planning benchmarks. Finally, we discuss a more general version of SAM learning, for the case where the mapping from lifted fluents to grounded fluents cannot be inverted.

Background and Problem Definition

We define a classical planning **domain** by a tuple $\langle T, O, \mathcal{F}, \mathcal{A}, M \rangle$ where T is a set of types, O is a set of objects, $\mathcal{F}$ is a set of lifted fluents, $\mathcal{A}$ is a set of lifted actions, and M is an action model for $\mathcal{A}$.

Every object $o \in O$ is associated with a type $t \in T$ denoted *type*(o). For example, in the logistics domain from the International Planning Competition (IPC) (McDermott 2000) there are types *truck* and *location* and there may be objects t_1 and t_2 that represent two different trucks and two objects l_1 and l_2 that represent two different locations.

Lifted and Grounded Literals

A *lifted fluent* F is a relation over a list of types. These types are called the *parameters* of F and denoted by *params*(F). For example, in the logistics domain $at(?truck, ?location)$ is a lifted fluent that represents some truck $(?truck)$ is at some location $(?location)$. A *binding* of a lifted fluent F is a function b : *params*$(F) \rightarrow O$ mapping every parameter of F to an object in O of the same type. A *grounded fluent* f is a pair $\langle F, b \rangle$ where F is a lifted fluent and b is a binding for F. To *ground* a lifted fluent F with a binding b means to create a relation over the objects in the image of b that match the relation over the corresponding parameters. We call this relation a *grounded fluent* or simply a fluent, and denote it by f. In our logistics example, for $F = at(?truck, ?location)$ and $b = \{?truck : truck1, ?location : loc1\}$ the corresponding grounded fluent f is $at(truck1, loc1)$. A *state* of the world is a set of grounded fluents. The term *literal* refers to either a fluent or its negation. The definitions of binding, lifted, and grounded fluent transfer naturally to literals.

Lifted and Grounded Actions

A lifted action $A \in \mathcal{A}$ is a pair $\langle$*name*, *params*$\rangle$ where *name* is a string and *params* is a list of types, denoted *name*(A) and *params*(A), respectively. The action model M for a set of actions $\mathcal{A}$ is a pair of functions pre_M and eff_M that map every action in $\mathcal{A}$ to its preconditions and effects. To define the preconditions and effects of a lifted action, we first define the notion of a *parameter-bound literal*. A *parameter binding* of a lifted literal L and an action A is a function $b_{L,A}$: *params*$(L) \rightarrow$ *params*(A) that maps every parameter of l to a parameter in A. A *parameter-bound literal* l for the lifted action A is a pair of the form $\langle L, b_{L,A} \rangle$ where b is a parameter binding of L and A. $pre_M(A)$ and $eff_M(A)$ are sets of parameter-bound literals for A.

A *binding* of a lifted action A is defined like a binding of a lifted fluent, i.e., a function b : *params*$(A) \rightarrow O$. A *grounded action* a is a tuple $\langle A, b \rangle$ where A is a lifted action and b_A is a binding of A. The preconditions of a grounded action a according to the action model M, denoted $pre_M(a)$, is the set of grounded literals created by taking every parameter-bound literal $\langle L, b_{L,A} \rangle \in pre_M(A)$ and grounding L with the binding $b_A \circ b_{L,A}$. The effects of a grounded action a, denoted $eff_M(a)$, are defined in a similar manner. The grounded action a can be applied in a state s iff $pre_M(a) \subseteq s$. The outcome of applying a to a state s according to action model M, denoted $a_M(s)$, is a new state that contains all literals in $eff_M(a)$ and all the literals in s such that their negation is not in $eff_M(a)$. Formally:

$$a_M(s) = \{l | l \in s \wedge \neg l \notin eff_M(a) \vee l \in eff_M(a)\} \qquad (1)$$

We omit M from $a_M(s)$ when it is clear from the context. The outcome of applying a sequence of grounded actions $\pi = (a^1, \dots a^n)$ to a state s is the state $s' = a_n(\cdots a_1(s) \cdots)$. A sequence of actions $a_1, \dots, a_n$ can be applied to a state s if for every $i \in 1, \dots, n$ the action a_i is applicable in the state $a_{i-1}(\cdots a_1(s) \cdots)$.

Definition 1 (Trajectory)**.** *A trajectory* $T = \langle s_0, a_1, s_1, \dots a_n, s_n \rangle$ *is an alternating sequence of states* $(s_0, \dots, s_n)$ *and actions* $(a_1, \dots, a_n)$ *that starts and ends with a state.*

The trajectory created by applying π to a state s is the sequence $\langle s_0, a_1, \dots, a_{|\pi|}, s_{|\pi|} \rangle$ such that $s_0 = s$ and for all $0 < i \leq |\pi|$, $s_i = a_i(s_{i-1})$. In the literature on learning action models (Wang 1994, 1995; Stern and Juba 2017; Walsh and Littman 2008), it is common to represent a trajectory $\langle s_0, a_1, \dots, a_{|\pi|}, s_{|\pi|} \rangle$ as a set of triples $\{ \langle s_{i-1}, a_i, s_i \rangle \}_{i=1}^{|\pi|}$. Each triple $\langle s_{i-1}, a_i, s_i \rangle$ is called an *action triplet*, and the states s_{i-1} and s_i are referred to as the pre- and post- state of action a_i. We denote by $\mathcal{T}(a)$ the set of all action triplets in the trajectories in $\mathcal{T}$ that include the action a. $\mathcal{T}(A)$ is similarly defined for all action triplets that contain actions that are groundings of A.

A classical planning **problem** is a tuple $\langle D, s_I, s_g \rangle$ where D is a classical planning domain, s_I is the start state, i.e., the state of the world before planning, and s_g is a set of grounded literals that define when the goal has been found. A solution to a planning problem is a sequence of grounded actions that can be applied to s_I and if applied to s_I results in a state s' that contains all the grounded literals in s_g. Such a sequence of grounded actions is called a *plan*. The trajectory of a plan starts with s_I and ends with a goal state s_G (where $s_g \subseteq s_G$). The *safe model-free planning* problem (Stern and Juba 2017) is defined as follows.

Definition 2 (Safe model-free planning)**.** *Let* $\Pi = \langle \langle T, O, \mathcal{F}, \mathcal{A}, M^* \rangle, s_I, s_g \rangle$ *be a classical planning problem and let* $\mathcal{T} = \{\mathcal{T}^1, \dots, \mathcal{T}^m\}$ *be a set of trajectories for other planning problems in the same domain. The input to a safe*

model-free planning algorithm is the tuple $\langle T, O, s_I, s_g, \mathcal{T} \rangle$ *and the desired output is a plan* π *that is a solution to* Π*. We denote this safe model-free planning problem as* $\Pi_{\mathcal{T}}$.

The main challenge in this problem is that the problem-solver – the agent – needs to find a sound plan for a planning problem but it is not given the set of fluents, actions, and action model of the domain ($\mathcal{F}$, $\mathcal{A}$, and M^*, respectively). We assume that when the agent observes a grounded action $a = \langle A, b_a \rangle$, it is able to discern that a is the result of grounding A with b_a. Similarly, if it observes a state with a grounded fluent $f = \langle F, b_f \rangle$, it is able to discern that f is the result of grounding F with b_f.

Conservative Planning in Grounded Domains

Our approach for solving the model-free planning problem in lifted domains builds on the conservative planning approach proposed by Stern and Juba (2017) for grounded domains. Thus, we first recall their approach.

Inference Rules for Grounded Domains

In a grounded domain, a state is a set of literals, and so are the preconditions and effects of all actions. That is, there is no notion of lifted literals of actions.

First, we define the notion of a consistent action model following the semantics of classical planning.

Definition 3 (Consistent Action Model)**.** *An action model* M *is consistent with a set of trajectories* $\mathcal{T}$ *if for every action triplet* $\langle s, a, s' \rangle \in \mathcal{T}(a)$ *it holds that:*
1. *All preconditions are satisfied:* $\forall l \in pre(a) \forall s : l \in s$
2. *All effects are satisfied:* $\forall l \in eff(a) \forall s' : l \in s'$
3. *Frame axioms hold:* $\forall (l \notin eff(a) \wedge l \notin s) \rightarrow l \notin s'$

The contrapositives of the conditions in the above definition can be interpreted as inference rules as follows.

Observation 1 (Inference rules for grounded domains)**.** *For any action triplet* $\langle s, a, s' \rangle$ *it holds that:*
- *Rule 1 [not a precondition].* $\forall l \notin s : l \notin pre(a)$
- *Rule 2 [not an effect].* $\forall l \notin s' : l \notin eff(a)$
- *Rule 3 [must be an effect].* $\forall l \in s' \setminus s : l \in eff(a)$

So, Rule 1 states that a literal that is not in a pre-state cannot be a precondition. Rule 2 states that a literal that is not in a post-state cannot be an effect. Rule 3 states that a literal that is in the post-state but not in the pre-state, must be an effect. Since this is just a restatement of the definition of a consistent action model, these rules precisely characterize the action models that are consistent with a given set of traces.

Definition 4 (Safe Action Model)**.** *An action model* M' *is safe with respect to an action model* M *iff for every state* s *and grounded action* a *it holds that*

$$pre_{M'}(a) \subseteq s \rightarrow \Big(pre_{M}(a) \subseteq s \wedge a_{M'}(s) = a_{M}(s) \Big) \qquad (2)$$

Definition 4 says that if a is applicable in s according to a safe action model (M'), then (1) a is also applicable in s according to the action model M, and (2) the state resulting by applying a to s is the same according to both action models.

We say that an action model is safe if it is a safe action model w.r.t. the real action model M^*. Observe that any plan generated by a planner given a safe action model must also be a sound plan according to M^*. The *conservative planning* algorithm (Stern and Juba 2017) for model-free planning is based on this observation. In conservative planning, we first learn from the given set of trajectories an action model M that is safe w.r.t. M^*, and then apply an off-the-shelf planner to generate plans using M. To implement this algorithm, Stern and Juba (2017) proposed an algorithm for learning a safe action model, that we refer to as the *Safe Action-Model (SAM) Learning* algorithm.

SAM Learning for Grounded Domains

SAM Learning works as follows. First, it assumes every action a has all literals as its preconditions and no literals as its effects. Then, it iterates over every action triplet in $\mathcal{T}(a)$ and applies the rules in Observation 1 to remove incorrect preconditions and to add effects.

Theorem 1 (SAM Learning is sound (Stern and Juba 2017))**.** *SAM learning produces a safe action model.*

The main limitation of using a safe action model to generate plans is that it may be more restrictive than the real action model (M^*). That is, there may be states in which an action a is applicable according to M, but not according to the safe action model. Consequently, there may be planning problems that are solvable with M, but not with the safe action model. Thus, if an action model M' is safe w.r.t. another action model M then in some sense it is *weaker*. Next, we complement Theorem 1 by showing that every safe action model that is consistent with the given trajectories must be weaker than the action model returned by SAM learning.

Clearly, in the fully observable deterministic world of classical planning, every action model that is not consistent with the given set of trajectories is false. Moreover, there exists a trajectory in the domain in which using such a false action model will yield a failure. Thus, the set of consistent action models must contain the real action model.

Theorem 2 (SAM Learning is complete)**.** *Let* M_{SAM} *be the action model created by SAM learning given the set of trajectories* $\mathcal{T}$*. Every action model* M' *that is consistent with* $\mathcal{T}$ *and safe w.r.t. the real action model* M^* *is also safe with respect to* M_{SAM}.

Proof. Consider an action model M' that is consistent with $\mathcal{T}$ and safe w.r.t. M^*. Let a be an action and s be a state such that a is applicable in s according to M', i.e., $pre_{M'}(a) \in s$. Since M' is safe w.r.t. M^*, $\Big(pre_{M^*}(a) \subseteq s \wedge a_{M'}(s) = a_{M^*}(s) \Big)$ By construction of M_{SAM}, if a literal l is a precondition of a according to M_{SAM}, then it has appeared in the pre-state of all action triplets in $\mathcal{T}(a)$. Thus, there exists a consistent action model in which l is a precondition of a and this action model may be the real model. Therefore, since M' is safe it follows that $pre_{M'}(a) \subseteq pre_{M_{SAM}}(a)$, and thus a is applicable in s according to M_{SAM}, i.e., $pre_{M_{SAM}}(a) \in s$. Since M_{SAM} is safe, $a_{M_{SAM}}(s) = a_{M^*}(s) = a'_M(s)$. □

Theorem 2 says that every action-model learning algorithm is bound to either return an unsafe action model or

return a action model model that is weaker than the action model returned by SAM learning. However, the action model returned by SAM may still be weaker than the real action model, and consequently, conservative planning for model-free planning is bound to be sound but incomplete – it generates plans that are sound but it may fail to generate plans for some solvable planning problems.

A statistical analysis showed that under some assumptions, the number of trajectories SAM learning needs to learn a safe action model that can solve most problems is quasilinear in the number of actions in the domain (Stern and Juba 2017). However, the number of grounded actions in a *lifted domain* can be quite large: the number of grounded actions that are groundings of a single lifted action grows polynomially with the number of objects in the domain (exponentially in the number of parameters). However, in a lifted domain, the real action model is assumed to be defined only over lifted actions. This enables us to generalize SAM learning across multiple groundings of the same lifted action, eliminating the dependence on the number of objects in the number of trajectories needed to learn a useful safe action model. We describe this in the next section.

Conservative Planning for Lifted Domains

In this section, we describe a conservative planning approach for safe model-free planning in lifted domains, which is based on a novel generalization of SAM learning to lifted domains. To describe our SAM learning algorithm for lifted domains, we denote by $\mathit{bindings}(b_A, b_L)$ the set of all parameter bindings $b_{L,A}$ that satisfy the following

$$b_A \circ b_{L,A} = b_L. \tag{3}$$

Inference Rules for Lifted Domains

The core of our algorithm is the following generalization of Observation 1, defining what observing an action triplet with a grounded action $\langle A, b_A \rangle$ entails for the lifted action A.

Observation 2. *For any action triplet* $\langle s, \langle A, b_A \rangle, s' \rangle$

- *Rule 1 [not a precondition].* $\forall \langle L, b_L \rangle \notin s$:

$$\forall b \in \mathit{bindings}(b_A, b_L) : \langle L, b \rangle \notin \mathit{pre}(A) \tag{4}$$

- *Rule 2 [not an effect].* $\forall \langle L, b_L \rangle \notin s'$:

$$\forall b \in \mathit{bindings}(b_A, b_L) : \langle L, b \rangle \notin \mathit{eff}(A) \tag{5}$$

- *Rule 3 [an effect].* $\forall \langle L, b_L \rangle \in s' \setminus s$:

$$\exists b \in \mathit{bindings}(b_A, b_L) : \langle L, b \rangle \in \mathit{eff}(A) \tag{6}$$

For much of this paper, we make the following assumption:

Definition 5 (Injective Action Binding)**.** *In every grounded action* $\langle A, b_A \rangle$*, the binding* b_A *is an injective function, i.e., every parameter of* A *is mapped to a different object.*

Under this assumption, for every pair of bindings b_L and b_A there exists a unique $b_{L,A}$ that satisfies Eq. 3. This binding is obtained by inverting b_A, i.e.,

$$\mathit{bindings}(b_A, b_L) = \{(b_A)^{-1} \circ b_L\}. \tag{7}$$

This simplifies the inference rules given in Observation 2. In particular, the "an effect" rule (Rule 1) becomes

$$\forall \langle L, b_L \rangle \in s' \setminus s : \left\langle L, (b_A)^{-1} \circ b_L \right\rangle \in \mathit{eff}(A). \tag{8}$$

Algorithm 1: Safe Action-Model (SAM) Learning

Input : $\Pi_{\mathcal{T}} = \langle T, O, s_I, s_g, \mathcal{T} \rangle$
Output: An action model that is safe w.r.t. the action model that generated $\mathcal{T}$

1 $\mathcal{A}' \leftarrow$ all lifted actions observed in $\mathcal{T}$
2 **foreach** *lifted action* $A \in \mathcal{A}'$ **do**
3 $\quad \mathit{eff}(A) \leftarrow \emptyset$
4 $\quad \mathit{pre}(A) \leftarrow$ all parameter-bound literals
5 $\quad$ **foreach** $(s, \langle A, b_A \rangle, s') \in \mathcal{T}(A)$ **do**
6 $\quad\quad$ **foreach** $\langle L, b_{L,A} \rangle \in \mathit{pre}(A)$ **do**
7 $\quad\quad\quad$ **if** $\langle L, b_A \circ b_{L,A} \rangle \notin s$ **then**
8 $\quad\quad\quad\quad$ Remove $\langle L, b_{L,A} \rangle$ from $\mathit{pre}(A)$
9 $\quad\quad$ **foreach** $\langle L, b_L \rangle \in s' \setminus s$ **do**
10 $\quad\quad\quad$ $b_{L,A} \leftarrow \left\langle L, (b_A)^{-1} \circ b_L \right\rangle)$
11 $\quad\quad\quad$ Add $\langle L, b_{L,A} \rangle$ to $\mathit{eff}(A)$
12 **return** $(\mathit{pre}, \mathit{eff})$

SAM Learning for Lifted Domains

We now present our SAM Learning algorithm for lifted domains in Algorithm 1. For every lifted action A observed in some trajectory, we initially assume that A has no effects and all possible parameter-bound literals are its preconditions (line 4 in Algorithm 1).[2] Then, for every action triplet $(s, \langle A, b_A \rangle, s')$ with this lifted action, we remove from the preconditions of A every parameter-bound literal $\langle L, b_{L,A} \rangle$ that is not satisfied in the current pre-state (Rule 2 in Observation 2). Then, for every grounded literal $\langle L, b_L \rangle$ that holds in the post-state s' and not in s, we add a corresponding effect to A (Rule 1 in Observation 2). Note that Rule 3 in Observation 2 is not needed since we initialize the set of effects of every action to be an empty set.

Safety Property

We extend the notion of a *safe action model* to lifted domains as follows. An action model M in a lifted domain is safe iff every grounded action defined by M satisfies Eq. 2. This definition preserves the property that a safe action model is an action model that enables generating plans that are guaranteed to be sound w.r.t. M^*. We show next that *SAM* Learning for lifted domains indeed returns a safe action model.

Theorem 3. *Given the injective action binding assumption, SAM Learning (Algorithm 1) creates a safe action model.*

Proof. We first show by induction on the iterations of the loop in lines 5–11 that on every iteration

$$\mathit{pre}_{M^*}(A) \subseteq \mathit{pre}(A) \text{ and } \mathit{eff}(A) \subseteq \mathit{eff}_{M^*}(A) \tag{9}$$

where M^* is the correct action model. Prior to the first iteration, the preconditions of all lifted actions A are all possible parameter-bound literals, so $\mathit{pre}(A)$ must be a subset of $\mathit{pre}_{M^*}(A)$. (Note that this includes every parameter-bounded fluent *and* its negation.) Similarly, the effects are

[2] It is possible to initialize the preconditions of every lifted action to the pre-state of one of the action triplets in which it is used.

Action	Params	Precond.	Effects
Move	?tr - truck ?from - location ?to - location	at(tr, from)	at(tr, to), not(at(tr, from))
Load	?pkg - package ?tr - truck ?loc - location	at(tr, loc) at(pkg, loc)	on(pkg, tr), not(at(pkg, loc))
Unload	?pkg - package ?tr - truck ?loc - location	at(tr, loc), on(pkg, tr)	not(on(pkg,tr), at(pkg, loc)

Table 1: The parameters, preconditions, and effects of the actions in our simple logistics example.

set to $\emptyset$, which is surely a subset of $\mathit{eff}_{M^*}(A)$. The changes made to $\mathit{pre}(A)$ and $\mathit{eff}(A)$ in subsequent iterations are encapsulated in lines 8 and 11 in Algorithm 1. Line 8 is a direct application of Rule 1 ("not a precondition") from Observation 2, and thus $\mathit{pre}(A)$ is still a subset of pre_{M^*}. Similarly, line 11 is an application of Rule 3 ("an effect") in the same observation, given that $\mathit{bindings}(b_A, b_L)$ consists of a single parameter binding due to the injective binding assumption. This completes the induction.

Let $(\mathit{pre}, \mathit{eff})$ be the action model returned by Algorithm 1 (line 12). From the induction above (Eq. 9) it immediately follows that for every grounded action $\langle A, b_A \rangle$ and state s, if $\mathit{pre}(\langle A, b_A \rangle) \subseteq s$ then $\mathit{pre}_{M^*}(\langle A, b_A \rangle) \subseteq s$. From the induction above, all the parameter-bound literals in $\mathit{eff}(A)$ are indeed effects of A. Finally, consider any parameter-bound literal $\langle L, b_{L,A} \rangle$ that is an effect of A but is absent from $\mathit{eff}(A)$, i.e., every $\langle L, b_{L,A} \rangle \in \mathit{eff}_{M^*}(A) \setminus \mathit{eff}(A)$. By construction of eff, this can only occur if this parameter-bound literal was true in all pre-states of groundings of A in all the available trajectories. Consequently, $\langle L, b_{L,A} \rangle$ must be in $\mathit{pre}(A)$. Therefore, every grounded literal in the post-state of applying $\langle A, b_A \rangle$ in s (i.e., $\langle A, b_A \rangle_{M^*}(s)$) is either in $\mathit{eff}(\langle A, b_A \rangle)$ or $\mathit{pre}(\langle A, b_A \rangle)$. □

Consider the following simple logistics problem. There are five objects: one *truck* object (tr), one *package* object (pkg), and three *locations* objects (A, B, and C). *at(?truck, ?location)* and *on(?truck, ?package)* are lifted fluents representing that the truck is in the location and the package is on the track, respectively. There are three possible actions: *Move*, *Load*, and *Unload*. Table 1 lists the parameters, preconditions, and effects of these actions. Now, assume we are given three trajectories T_1, T_2, and T_3. T_1 starts with the truck and the package at location A, and performs two move actions: Move(tr, A, B) and Move(tr, B, C). T_2 starts in the same state, but performs Load(pkg, tr, A) and Move(tr, A, B). T_3 starts with the truck at location A and the package at location B, and performs Move(tr, A, B), Load(pkg, tr, B), Move(tr, B, C), and Unload(pkg, tr, C). Given only the first trajectory T_1, the action model returned by SAM Learning already contains the correct action model for the lifted Move action, since the only grounded fluents that can be bound to the parameters of the grounded action Move(tr, A, B) are at(tr, A) and not(at(tr, B)) in the pre-state, and at(tr, B) and not(at(tr, A)) in the post-state. In contrast, SAM Learning for grounded domains will not know anything about the preconditions and effects of the grounded action Move(tr, B, C) unless it is also given the trajectory T_3. Similarly, given the second trajectory T_2, the action model returned by SAM Learning contains the correct action model for the lifted Load action, since the only grounded fluents that can be bound to the parameters of the grounded action Load(pkg, tr, A) are at(tr, A), at(pkg, A), and not(on(pkg, tr)) in the pre-state and at(tr, A), not(at(pkg, A)), and on(pkg, tr)) in the post-state. In fact, given T_1, T_2, and T_3, SAM Learning is able to learn the correct action model for this domain. Note that since there are 10 grounded actions in this domain (four Move actions and three Load and Unload actions), SAM Learning for grounded domains will require at least 10 trajectories to learn an action model with all of the actions.

Sample Complexity Analysis

Planning with a safe action model is a sound approach for safe model-free planning, since every plan it outputs is a sound plan according to the real action model. However, it is not complete: a planning problem may be solvable with the real action model, but not the learned one. As in prior work on safe model-free planning (Stern and Juba 2017), we can bound the likelihood of facing such a problem as follows.

Let $\mathcal{P}_D$ be a probability distribution over solvable planning problems in a domain D. Let $\mathcal{T}_D$ be a probability distribution over pairs $\langle P, T \rangle$ given by drawing a problem P from $\mathcal{P}(D)$, using a sound and complete planner to generate a plan for P, and setting T to be the trajectory from following this plan.[3] Let $\mathit{arity}(F, t)$ and $\mathit{arity}(A, t)$ be the number of type-t parameters of the lifted fluent F and action A.

Theorem 4. *Under the injective binding assumption, given* $m \geq \frac{1}{\epsilon}(2 \ln 3 \sum_{\substack{F \in \mathcal{F} \\ A \in \mathcal{A}}} \prod_{t \in T} \mathit{arity}(A, t)^{\mathit{arity}(F,t)} + \ln \frac{1}{\delta})$ *trajectories sampled from* $\mathcal{T}_D$*, with probability at least* $1 - \delta$ *SAM learning for lifted domains (Algorithm 1) returns a safe action model* M_{SAM} *such that a problem is drawn from* $\mathcal{P}_D$ *that is not solvable with* M_{SAM} *with probability at most* ϵ*.*

Theorem 4 guarantees that with high probability ($\geq 1 - \delta$) SAM Learning returns an action model that will only fail to solve a given problem with low probability ($\leq \epsilon$), given a number of example trajectories linear in the size of the models. Indeed, there are $\mathit{arity}(A, t)^{\mathit{arity}(F,t)}$ ways to bind the parameters of F of type t to the parameters of A, and hence $\prod_{t \in T} \mathit{arity}(A, t)^{\mathit{arity}(F,t)}$ ways of binding all of the parameters of F to parameters of A. The preconditions and effects are sets of these parameter-bound fluents. For example, in our simple logistics example with two binary fluents and three ternary actions, the load and unload actions have a single argument of each type; only the move action has two arguments of the same type (location). The only fluents that have location arguments are the at fluents, which have arity one with respect to locations. Thus, guaranteeing $\epsilon = \delta = 5\%$ requires only 324 trajectories. The rest of this section is devoted to establishing Theorem 4.

Definition 6 (Adequate)**.** *An action model* M *is* ϵ-adequate *if, with probability at most* ϵ*, a trajectory* T *sampled from* $\mathcal{T}_D$

[3] The planner need not be deterministic.

contains an action triplet $\langle s, a, s' \rangle$ where s does not satisfy $pre_M(a)$.[4]

Lemma 1. *The action model returned by SAM Learning (Algorithm 1) given m trajectories (as specified in Theorem 4) is ϵ-adequate with probability at least $1 - \delta$.*

Proof. Consider any action model M_B that may be returned by SAM Learning but is not ϵ adequate. By definition, the probability of drawing a trajectory from $\mathcal{T}_D$ that is inconsistent with M_B is at least ϵ. Thus, the probability of drawing m samples that are consistent with M_B is at most

$$(1 - \epsilon)^m \leq e^{-m \cdot \epsilon}. \tag{10}$$

M_B can only be returned if this occurs. For our choice of m,

$$e^{-m \cdot \epsilon} \leq e^{-(\ln 3L + \ln \frac{1}{\delta})} = \frac{\delta}{3^L} \tag{11}$$

where

$$L = 2 \sum_{\substack{F \in \mathcal{F} \\ A \in \mathcal{A}}} \prod_{t \in T} arity(A, t)^{arity(F,t)}$$

Let B be the set of action models that are not ϵ-adequate. By a union bound over B, the probability that SAM Learning will return an action model that is not ϵ-adequate is at most $\frac{|B|\delta}{3^L}$. For each parameter-bound fluent, each precondition or effect will either contain that fluent, or its negation, or neither of them. Hence, the number of possible action models is 3^L. Since B is a set of action models, we have that the size of B is at most 3^L. Therefore, the probability that SAM Learning will return an action model that is not ϵ-adequate is at most δ. □

Proof of Theorem 4. Let M be an action model returned by SAM Learning given m samples. Thus, M is a safe action model (Theorem 3) and it is ϵ adequate (Lemma 1). Consider a problem P drawn from $\mathcal{P}(D)$, and its corresponding pair $\langle P, T \rangle$ from $\mathcal{T}(D)$. Since M is ϵ-adequate, with probability at least $1 - \epsilon$, for every action triplet $\langle s, a, s' \rangle \in T$ a is applicable in s, that is, $pre_M(a) \subseteq s$. Since M is a safe action model, we have that $a_M(s) = a_{M^*}(s) = s'$. Thus, with probability at least $1 - \epsilon$ the trajectory T is consistent with the learned action model M, and therefore P can be solved with M □

Multiple Action Bindings

When the injective action-binding assumption does not hold, multiple action parameters are bound to the same object and thus $(b_A)^{-1}$ is not defined. As a result, when SAM Learning infers an effect (Rule 1 in Observation 2) it cannot generalize it to be a unique effect of the corresponding lifted action, as done in line 10 in Algorithm 1. This poses a challenge to learning a safe action model, as the information that can be inferred from observing action triplets can be complex.

For example, consider a lifted action $A(x, y)$. Suppose x and y are associated with the same type and o is an object of that type. Given the action triplet $\langle \{\}, A(o, o), \{L(o)\} \rangle$, the agent can infer that $L(o)$ is an effect of the grounded action $A(o, o)$. However, the agent cannot accurately infer the effect of the lifted action $A(x, y)$: it can be either $\{L(x)\}$, $\{L(y)\}$, or both. Concretely, if o_1 and o_2 are two different objects from the same type as o, the agent cannot determine if applying $A(o_1, o_2)$ will result in a state with $\{L(o_1)\}$, $\{L(o_2)\}$, or $\{L(o_1), L(o_2)\}$. Consequently, any safe action model must not enable groundings of A that bind x and y to different objects, unless $L(x)$ and $L(y)$ both already hold.

Now, assume the agent is also given the action triplet $\langle \{L(o_1)\}, A(o_1, o_2), \{L(o_1)\} \rangle$. The pre- and post-state are the same, so in Algorithm 1 we cannot learn any new effects of A from this triplet. However, we can infer that $L(o_2)$ is not an effect of the grounded action in this triplet. Consequently, the parameter-bound literal $L(y)$ cannot be an effect of the lifted action A. Thus, this second action triplet does provide useful information: it allow us to infer that the lifted action $A(x, y)$ has a parameter-bound effect $L(x)$. Next, we describe Extended SAM Learning, which is able to capture the above form of inference and is applicable to cases where the injective action-binding assumption does not hold.

Extended SAM Learning

Extended SAM (E-SAM) learning works in two stages. First, it creates for every lifted action A two Conjunctive Normal Form (CNF) formulas, denoted $CNF_{pre}(A)$ and $CNF_{eff}(A)$, that describe a set of constraints for a safe action model. Then E-SAM learning generates a safe action model based on these CNFs.

Safe Action Model Constraints $CNF_{pre}(A)$ uses atoms of the form IsPre($\langle L, b_{L,A} \rangle$), which specify that $\langle L, b_{L,A} \rangle$ is a precondition L in a safe action model. Similarly, $CNF_{eff}(A)$ uses atoms of the form IsEff($\langle L, b_{L,A} \rangle$), which specify that $\langle L, b_{L,A} \rangle$ is an effect of L in a safe action model.

Initially, $CNF_{pre}(A)$ and $CNF_{eff}(A)$ represent that all possible parameter-bound literals are preconditions and there are no effects. Then, E-SAM learning iterates over every action triplet (s, a, s') in the given set of trajectories in which a is a grounding of A. For every such triplet, it applies the inference rules in Observation 2 as follows.

Every parameter-bound literal $\langle L, b_{L,A} \rangle$ such that $\langle L, b_A \circ b_{L,A} \rangle$ is not in the pre-state cannot be a precondition (Rule 1). So, we remove IsPre($\langle L, b_{L,A} \rangle$) from CNF_{pre} for such parameter-bound literals. Similarly, every parameter-bound literal $\langle L, b_{L,A} \rangle$ such that $\langle L, b_A \circ b_{L,A} \rangle$ is not in the post-state cannot be an effect (Rule 2). So, we add ¬IsEff($\langle L, b_{L,A} \rangle$) to CNF_{eff} for such parameter-bound literals. Finally, every grounded literal $\langle L, b_L \rangle$ in $s' \setminus s$ must be an effect. So, we add to CNF_{eff} the disjunction over all parameter-bound literals $\langle L, b_A \circ b_{L,A} \rangle$ that satisfy $\langle L, b_A \circ b_{L,A} \rangle = \langle L, b_L \rangle$ (Rule 3). Once the given trajectories have been processed by the algorithm, we simplify both CNFs by applying unit propagation and removing subsumed clauses.

Proxy Actions The main challenge in creating a safe action model from the generated CNFs is the disjunction in CNF_{eff}, which represents uncertainty w.r.t to the effects of

[4] An action model may not contain any information about some action a. For the purpose of safe planning this is equivalent to an action model in which the precondition to a can never be satisfied.

action. To address this, we create a safe action model with a set of proxy actions that ensure every action is only applicable when we know its effects. This is done as follows.

If an action has only unit clauses, we have a single action with the effects indicated by the positive literals. Otherwise, we create a proxy action for all subsets of the non-subsumed non-unit clauses. (The number of proxy actions is thus exponential in the number of non-unit clauses.) In this proxy action, we identify all of the parameters that appear together in a clause in the subset; if multiple clauses share any variables, we identify all of the parameters across the clauses. Each proxy action has the following set of preconditions and effects: every unit clause in the CNF and every clause in the corresponding subset specifies an effect of the proxy action. For the subset of clauses not chosen for this proxy action, the proxy action has the corresponding literals as additional preconditions, in addition to the preconditions of the original SAM Learning action model. Every plan generated by the action model created by the resulting action model is translated to a plan without proxy actions by replacing them with the actions for which they were created. Algorithm 2 lists the complete pseudocode of E-SAM learning.

Theoretical Properties The E-SAM Learning action model satisfies the same properties as the original model (assuming injective bindings), captured in Theorems 3 and 2:

Theorem 5. *The E-SAM Learning action model is safe.*

Proof. For each of the proxy actions, for every effect, at least one of the parameter-bound literals for the identified parameters is an effect of the true action. Furthermore, the preconditions ensure that the rest of the uncertain effects are already present in the pre-state. The post-state of the proxy action is thus identical to that of the true action when its precondition is satisfied. Likewise, the proxy actions have preconditions that are only stronger than the actual precondition. Eq. 2 therefore holds. The rest of the claim now follows from the argument in Theorem 3. □

Recall that a *prime implicate* is a clause that is entailed by a formula for which no subclause is also entailed. CNF_{eff} consists of precisely these prime implicates.

Lemma 2. *All prime implicates of CNF_{eff} are derived by unit propagation.*

Proof. Note that the clauses created by Rule 3 contain only positive literals, and negative literals are only created by Rule 1 and 2, which create unit clauses. Hence, unit propagation is sufficient to capture all possible cut inferences (a.k.a., resolution inferences) from these clauses. By the completeness of resolution for prime implicates (see, e.g., (Brachman and Levesque 2004, Chapter 13, Exercise 1)), all of the prime implicates of CNF_{eff} can be derived by applications of cut. In turn, therefore, unit propagation can also derive all of the prime implicates of CNF_{eff}. □

Theorem 6. *Every action model M' that is consistent with $\mathcal{T}$ and safe w.r.t. the real action model M^* is also safe with respect to the extended SAM Learning action model.*

Algorithm 2: Extended SAM Learning

Input : $\Pi_{\mathcal{T}} = \langle T, O, s_I, s_g, \mathcal{T} \rangle$
Output: $(\mathit{pre}, \mathit{eff})$ for a safe action model

$\mathcal{A}' \leftarrow$ all lifted actions observed in $\mathcal{T}$
foreach *lifted action* $A \in \mathcal{A}'$ **do**
  $(CNF_{\mathit{pre}}, CNF_{\mathit{eff}}) \leftarrow$ ExtractClauses(A, $\mathcal{T}(A)$)
  $CNF^1_{\mathit{eff}} \leftarrow$ all unit clauses in CNF_{eff}
  SurelyEff$\leftarrow \{l \mid \text{IsEff(l)} \in CNF^1_{\mathit{eff}}\}$
  SurelyPre$\leftarrow \{l \mid \text{IsPre}(l) \in CNF_{\mathit{pre}}\}$
  /* Create proxy actions for non-unit effects clauses */
  $CNF_{\mathit{eff}} \leftarrow CNF_{\mathit{eff}} \setminus CNF^1_{\mathit{eff}}$
  foreach $S \in \mathit{Powerset}(CNF_{\mathit{eff}})$ **do**
    $\mathit{pre}(A_S) \leftarrow$ SurelyPre; $\mathit{eff}(A_S) \leftarrow$ SurelyEff
    foreach $C_{\mathit{eff}} \in CNF_{\mathit{eff}} \setminus S$ **do**
      foreach $\mathit{IsEff}(l) \in C_{\mathit{eff}}$ **do**
        Add l to $\mathit{pre}(A_S)$
    MergeObjects$\big(S, \mathit{pre}(A_S), \mathit{eff}(A_S)\big)$
return $(\mathit{pre}, \mathit{eff})$

Proof. Let M' be an action model that is consistent with $\mathcal{T}$ and safe w.r.t. M^*, and consider any transition $\langle s, \langle A, b\rangle, s'\rangle$ permitted by M'. Consider the set S of literals in $s' \setminus s$ that do not correspond to unit clauses in the CNF created by E-SAM Learning, and the set $\bar{S}$ of literals that are the groundings under b of the effects in the non-unit clauses created by E-SAM learning that are not in $s' \setminus s$. Recall, Observation 2 characterizes the set action models consistent with $\mathcal{T}$ and by Lemma 2, no sub-clause of the CNF created by E-SAM learning is entailed by the rules of Observation 2. Therefore, for every literal of every non-unit clause of this CNF, there exists an action model consistent with $\mathcal{T}$ in which that literal is the only satisfied literal of the clause. (Otherwise, a strictly smaller clause would be entailed.) Therefore, for each literal $l \in S$, since M' is safe w.r.t. M^*, all of the parameters of A in some clause for this effect must be bound to the objects necessary to obtain l as the corresponding effect. Thus, b must be consistent with at least one of the proxy actions A_{proxy}. Furthermore, since the literals in $\bar{S}$ may be effects of $\langle A, b\rangle$, if they are not in $s' \setminus s$, they must be in s, so the preconditions of A_{proxy} are satisfied as well. Since the E-SAM Learning action model is safe by Theorem 5, the post-state of A_{proxy} is therefore equal to that obtained by the true action model, which is in turn also equal to s' since M' is also safe. A is therefore an application of A_{proxy}, and we see that the use of A in M' is safe with respect to the set of proxy actions in the E-SAM Learning action model. □

Experiments

To evaluate the performance of SAM Learning for lifted domains (Algorithm 1), we performed experiments on two simple IPC domains: *N-Puzzle* (3×3, 3 predicates, 1 action) and *Blocksworld* (8 blocks, 5 predicates, 4 actions). For each domain, we generated 10 trajectories with 10 actions each by taking random actions (all distinct lifted actions of a domain appear at least once in each trajectory). Then, we ran

SAM Learning on these trajectories and obtained a safe action model. As a baseline, we used FAMA (Aineto, Celorrio, and Onaindia 2019), which is a modern algorithm for learning an action model from trajectories. Note that FAMA has no safety guarantee. For N-Puzzle, both methods correctly learned the action model after observing a single $\langle s, a, s' \rangle$ triple in all 10 trials. For Blocksworld, Table 2 lists the number of state-action-state triples needed to learn a correct action model over 10 runs, where each run processed the trajectories in a random order.

# $\langle s, a, s' \rangle$	6	7	8	9	$\cdots$	13	14	15	16	17
SAM	1	1	6	2	0	0	0	0	0	0
FAMA	0	0	0	0	0	3	1	3	1	2

Table 2: # trials in which SAM Learning and FAMA learned the true Blocksworld action model with a given # of triples.

In all cases, SAM learning was able to recover a correct action model using fewer (s, a, s') triplets than FAMA. Note that once SAM Learning finds a correct model, it will not add more literals to the preconditions or effects, since SAM only removes literals that are not satisfied in the pre-state and adds literals that switch values between pre and post-states. Meanwhile, FAMA might add irrelevant literals to the preconditons or effects as it processes more transitions.

The code for SAM learning implementation and experiments is available at https://github.com/hsle/sam-learning.

Related Work

A variety of notions of *safety* have been considered in RL, for example capturing the ability to reliably return to a home state (Moldovan and Abbeel 2012) or avoiding undesirable states (which are often identified with negative "reward") (Turchetta, Berkenkamp, and Krause 2016; Wachi et al. 2018) while learning about the environment.

But, these approaches to safe exploration require some kind of strong prior knowledge, either in the form of beliefs about the transition model or knowledge that the safety levels follow a Gaussian process model. Such assumptions are reasonable in the low-level motion planning tasks where RL excels, but they do not suit the kind of discrete, high-level problems typically considered in domain-independent planning. In addition, in these works safety is soft constraint that an algorithm aims to maximize, while in our case safety is a hard constraint.

Conclusion and Future Work

In this work, we presented the Safe Action Model Learning algorithm for lifted domains. SAM Learning for lifted domains is guaranteed to return an action model that produces sound plans, even without knowledge of the domain. A theoretical analysis shows that the number of trajectories needed to learn an action model that will solve a given problem with high probability is linear in the potential size of the action model. This approach is suitable for most domains in current planning benchmarks, where the effects of actions are trivial unless the action parameters are bound to different objects. We also discussed how to adapt our algorithm to the case where this assumption does not hold. In the future, we aim to extend safe action-model learning to domains with partial observability and stochasticity. We will also examine its performance on more complicated IPC domains.

Acknowledgments

This research is partially funded by NSF award IIS-1908287 and BSF grant #2018684 to Roni Stern.

References

Aineto, D.; Celorrio, S.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence* 275. doi:10.1016/j.artint.2019.05.003.

Amir, E.; and Chang, A. 2008. Learning Partially Observable Deterministic Action Models. *J. Artif. Intell. Res. (JAIR)* 33: 349–402.

Brachman, R. J.; and Levesque, H. J. 2004. *Knowledge Representation and Reasoning*. Elsevier.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4): 189–208.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2): 13.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical report, AIPS '98 - The Planning Competition Committee.

Moldovan, T. M.; and Abbeel, P. 2012. Safe exploration in Markov decision processes. In *Proceedings of the 29th International Conference on Machine Learning*, 1451–1458.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *the International Joint Conference on Artificial Intelligence (IJCAI)*, 4405–4411.

Turchetta, M.; Berkenkamp, F.; and Krause, A. 2016. Safe exploration in finite markov decision processes with gaussian processes. In *Advances in Neural Information Processing Systems*, 4312–4320.

Wachi, A.; Sui, Y.; Yue, Y.; and Ono, M. 2018. Safe exploration and optimization of constrained mdps using gaussian processes. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Walsh, T. J.; and Littman, M. L. 2008. Efficient learning of action schemas and web-service descriptions. In *AAAI*, volume 8, 714–719.

Wang, X. 1994. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 335–340.

Wang, X. 1995. Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, 549–557.

Reinforcement Learning for Planning Heuristics

Patrick Ferber[1,2] and **Malte Helmert**[1] and **Jörg Hoffmann**[2]

University of Basel[1]
Switzerland
{firstname}.{lastname}@unibas.ch

Saarland Informatics Campus, Saarland University[2]
Saarbrücken, Germany
{lastname}@cs.uni-saarland.de

Abstract

Informed heuristics are essential for the success of heuristic search algorithms. But, it is difficult to develop a new heuristic which is informed on various tasks. Instead, we propose a framework that trains a neural network as heuristic for the tasks it is supposed to solve.

We present two reinforcement learning approaches to learn heuristics for fixed state spaces and fixed goals. Our first approach uses *approximate value iteration*, our second approach uses *searches* to generate training data. We show that in some domains our approaches outperform previous work, and we point out potentials for future improvements.

Introduction

A key component in classical planning is heuristic search (Bonet and Geffner 2001). A search algorithm like A^* (Hart, Nilsson, and Raphael 1968) or *greedy best-first search* uses an *heuristic* as guidance to the goal. The heuristic estimates for every visited state its distance to the goal. The closer those estimates are to the true goal distance, the faster we expect the search algorithm to find a solution. Countless researchers designed heuristics for optimal (e.g., Helmert and Domshlak 2009; Helmert et al. 2014; Haslum et al. 2007) and satisficing (e.g., Hoffmann and Nebel 2001; Richter and Westphal 2010; Domshlak, Hoffmann, and Katz 2015) planning or invented ways to combine the power of multiple heuristics (e.g., Röger and Helmert 2010; Seipp 2017).

Instead of designing search algorithms or heuristics independently of the tasks to solve, another line of research develops algorithms that can be adapted for different tasks.

For example, offline portfolio algorithms learn a schedule which describes a planner order and a time limit per planner. To solve a new task, the planners are executed in their order with their time limits (e.g., Helmert et al. 2011; Seipp 2018). Online portfolio algorithms learn a mapping that decides for a given task which planner to use (Sievers et al. 2019; Ma et al. 2020). Gomoluch et al. (2020) apply reinforcement learning to learn how to modify a running search algorithm depending on some statistics. Their algorithm switches among others between best-first search, local search, random walks.

We also use reinforcement learning, but we do not change the behavior of the search algorithm. Instead, we learn a heuristic. Arfaee, Zilles, and Holte (2010) learn to combine multiple feature heuristics into a new heuristic. Iteratively, they use the feature heuristics and their learned heuristics to solve a set of task. From every solved task, the states along the solution together with their estimates of the feature heuristics and their estimated goal distance are saved. These estimates are used to improve the learned heuristic. If they are not able to solve sufficiently many new task, they generate training tasks by regressing from the goal. This is possible, because in their domains regression produce complete assignments and quickly leads to random states.

Agostinelli et al. (2019) learn heuristic functions using reinforcement learning with approximate value iteration. They also generate training states using random walks from the goal. They evaluate training states by minimizing over the heuristic estimates of the states' successors. Like previously, in their domains regression produces complete states and random walks quickly lead to random states (especially in the Rubik's Cube domain).

Ferber, Helmert, and Hoffmann (2020b) take another route. They use progression from some seed task to generate training states. They use an arbitrary heuristic search algorithm to solve sampled states and every state encountered along a plan is stored for training. The authors train their heuristics using supervised learning. Contrary to the previous two approaches they evaluate their technique on domains for which regression does not produce complete assignments.

Like Agostinelli et al. (2019) and Ferber, Helmert, and Hoffmann (2020b), we learn heuristics for fixed state spaces and fixed goals. We present two approaches that use reinforcement learning to train heuristics on tasks of the *International Planning Competition* (IPC). On some domains we already outperform previous approaches, and we identified important future steps that will further improve our performance. The paper is organized as follows. First, we provide some background on planning and reinforcement learning. Next, we present how we train our heuristics. Then, we evaluate our approach on IPC tasks. And finally, we resume our results and present our next steps.

Background

We work on planning tasks in *Finite-Domain Representation* (FDR) (Bäckström and Nebel 1995). An FDR task Π is defined as a quad-tuple $\langle V, A, I, G\rangle$. V is a set of finite-domain variables. Every variable v has a domain $dom(v)$ that contains all values assignable to it. A *state* assigns to every variable exactly one value. A fact is a $\langle var, val\rangle$ pair with $val \in dom(var)$. Two facts can be *mutually exclusive* (mutex), i.e. they cannot be part of the same state. A is a set of actions. Every action $a \in A$ is defined as $\langle pre_a, \mathit{eff}_a\rangle$ and has a cost associated. Both, pre_a and eff_a are partial assignments to V. An action a is applicable in a state s if $pre_a \subseteq s$. Applying action a in state s leads to a new state s' with $s' = \{v \mapsto \mathit{eff}_a[v] \mid v \in V \text{ and } v \in \mathit{eff}_a\} \cup \{v \mapsto s[v] \mid v \in V \text{ and } v \notin \mathit{eff}_a\}$. This is also called progression. I is the initial state and G is a partial assignment which describes the goal of the task. A state s is a goal state if $s \subseteq G$. A *plan* is a sequence of actions $\langle a_1, a_2, \ldots, a_n\rangle$ such that applying one action after another leads from the initial state to a goal state.

In this paper we do not only use progression, but also regression. A partial state p is regressable with an action a if, firstly, $\mathit{eff}_a \cap p \neq \emptyset$, secondly, there is no $v \in V$ such that $v \in \mathit{eff}_a$ and $v \in p$ and $\mathit{eff}_a[v] \neq p[v]$, and thirdly, there is no $v \in V$ such that $v \notin \mathit{eff}_a$ and $v \in pre_a$ and $v \in p$ and $pre_a[v] \neq p[v]$. The result of regressing the partial state p with the action a is defined as $\{var \mapsto val \mid var \mapsto val \in p \text{ and } var \notin \mathit{eff}_a\} \cup \{var \mapsto val \mid var \mapsto val \in pre_a\}$ (Alcázar et al. 2013).

Sometimes planning and machine learning use the same notations, but with different meanings. We annotate variables with ˜, if we use their machine learning meaning. We use reinforcement learning to learn a value function $\tilde{V} : S \mapsto \mathbb{R}$ that assigns every state a value. One technique to learn this function is *approximate value iteration* (AVI) (Bertsekas and Tsitsiklis 1996). We start with an arbitrary function $\tilde{V}_0$ and iteratively improve it using Equation 1.

$$\tilde{V}_{n+1} = \tilde{\mathcal{A}}\mathcal{T}\tilde{V}_n \tag{1}$$

$\mathcal{T}$ denotes the Bellman operator and is defined in Equation 2.

$$\mathcal{T}(s) = \max_a \left[\mathcal{R}^a_s + \delta \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{s,s'} \tilde{V}(s')\right] \tag{2}$$

For a given state the Bellman operator provides an estimation of the expected total reward given the current value function $\tilde{V}_i$. We use the simplification of Agostinelli et al. (2019) shown in Equation 3. Our rewards $\mathcal{R}$ depend only on the action and can be replaced by the negated action cost. We do not need a weighted sum over possible successors, because our actions produce exactly one successor. We set δ to 1 such that the value function learns to estimate the remaining cost to the goal. All our rewards are negative (assuming planning tasks with non-negative costs). Therefore, we change the maximization to a minimization of negative rewards.

$$\mathcal{T}(s) = \min_a \left[cost(a) + \tilde{V}(s')\right] \tag{3}$$

$\tilde{\mathcal{A}}$ represents an approximation method that incorporates the sampled states and their values estimated by $\mathcal{T}$ and returns a new value function.

Training

We use reinforcement learning to train value functions that approximate the optimal heuristic for an FDR task Π. As approximation method $\tilde{\mathcal{A}}$ we use supervised learning. Our networks are residual network (He et al. 2016) with two dense layers followed by one residual block containing two dense layers and a single output neuron. Each dense layer contains 250 neurons. All neurons use the *ReLU* activation function. The inputs of our networks are states represented as fixed size Boolean vectors. We associate every entry of the input vector with a fact of Π. For all facts that are part of the input state we set their vector entries to 1. All other entries are set to 0. We train the network using the *mean squared error* as loss function and the *adam* optimizer with its default parameters (Kingma and Ba 2015). To prevent performance instabilities during training, we update the model for the sample generation after at least 50 epochs have passed and the mean squared error is below 0.1.

Because generating a training batch of 250 samples takes longer than training on that batch, we use experience replay. The data generating process pushes all samples into a *first-in-first-out* buffer with a maximum size of 25,000. In each training epoch we choose uniformly 250 samples from the buffer. This allows us to train multiple times on the same - recent - samples and to decouple the training from the data generation.

We run the data generation in four independent processes. Algorithm 1 provides an overview of them. Each process calls GENERATE_DATA and samples $\langle state, value\rangle$ pairs until we terminate it. First the process checks if a new value function is available. If yes, it loads the new value function. Then, the process samples a new state from the state space of Π using either SAMPLE_PROGRESSION or SAMPLE_REGRESSION. It evaluates the state using either EVALUATE_LOOKAHEAD or EVALUATE_SEARCH. EVALUATE_SEARCH can return multiple $\langle state, value\rangle$ pairs for each sampled state. Each $\langle state, value\rangle$ pair will be stored for training. Depending on some conditions, the process updates the parameters for the sampling methods.

SAMPLE_PROGRESSION starts at the initial state of the task Π, and performs a random walk for *walk_length* steps using progression. At each step it chooses a random applicable action which does not undo the previous action. SAMPLE_REGRESSION starts at the goal of Π, and performs a random walk for *walk_length* steps using regression. At each step it chooses a random regressable action which again does not undo the previous action. Unlike the progression walk, the regression walk ends with a partial assignment. We randomly complete the partial assignment to a state. Therefore, we assign every unassigned variable a value of its domain. We use the translator of Fast Downward (Helmert 2009) to

Algorithm 1 Generate Training Data

```
1: function SAMPLE_PROGRESSION(Π, walk_length)
2:     s, s' ← Π.I, None
3:     for i = 1..walk_length do
4:         a ← choose({a | a ∈ A, applicable(s, a) ∧
                          s' ≠ apply(s, a)})
5:         s, s' ← apply(s, a), s
6:     return s
7: function SAMPLE_REGRESSION(Π, walk_length)
8:     p, p' ← Π.G, None
9:     for i ← 1..walk_length do
10:        a ← choose({a | a ∈ A, regressable(p, a) ∧
                          p' ≠ regress(p, a)})
11:        p, p' ← regress(p, a), p
12:    return make_complete_assignment(p)
13: function EVALUATE_LOOKAHEAD(Π, s, Ṽ, lookahead)
14:    curr, succs ← [⟨s, 0⟩], []
15:    for i ← 1..lookahead do
16:        for s', c' ∈ curr do
17:            if is_goal(Π, s') then
18:                succs.insert(⟨s', c'⟩)
19:                continue
20:            for a ∈ {a | a ∈ A, applicable(Π, s', a)} do
21:                s'' ← apply(s', a)
22:                c'' ← c' + cost(Π, a)
23:                succs.insert(⟨s'', c''⟩)
24:        curr, succs ← succs, []
25:    c ← min({c' + (0 if is_goal(s') else Ṽ(s')) |
                s', c' ∈ curr})
26:    return [⟨s, c⟩]
27: function EVALUATE_SEARCH(Π, s, Ṽ, search)
28:    try
29:        plan ← search(Π, s, Ṽ)
30:        c ← sum([cost(a) | a ∈ plan])
31:        evals ← [⟨s, c⟩]
32:        for a ∈ plan do
33:            s, c ← apply(s, a), c − cost(Π, a)
34:            evals.insert(⟨s, c⟩)
35:        return evals
36:    exceptTIMEOUT, UNSOLVABLE
37:        return []
38: function GENERATE_DATE(Π, min_walk, max_walk, ls)
39:    while true do
40:        if value_function_outdated() then
41:            Ṽ ← load_value_function()
42:        s ← sample_X(Π, rnd(min_walk, max_walk))
43:        evals ← evaluate_Y(Π, s, Ṽ, ls)
44:        if s, v ∈ evals then
45:            store(s, v)
46:        if CONDITION(s, v) then
47:            min_walk ← update_min_walk_length()
48:            max_walk ← update_max_walk_length()
49:
```

identify some mutexes of the task Π and enforce that none of them are violated.

To label the sampled state, we use either EVALUATE_LOOKAHEAD or EVALUATE_SEARCH. EVALUATE_LOOKAHEAD is an adaption of the simplified Bellman operator in Equation 3. Instead of considering only the direct successors of the current state, the function considers the n-step successors. If it finds a goal state during the n-step successor exploration, then it will not further explore the successors of this state. This is possible, because we want to learn the distance to the *closest* goal. Any successor of a goal state is further afar from us than the goal state itself. Every n-step successor is evaluated by adding up the action cost to reach it with its estimate of the value function. If a successor is a goal state, then the optimal value function would assign it to 0. Thus, we evaluate goal states with their action costs only. EVALUATE_SEARCH evaluates a state by executing a - potentially suboptimal - search on the state. If the search finds a plan, then it stores all states along the plan for training. Their associated values are the summed action costs from their position in the plan to the goal.

We use 2 different configuration to generate training data in our experiments. The first configuration samples all states using SAMPLE_REGRESSION and a random walk length between 0 and 300. It evaluates states using EVALUATE_LOOKAHEAD with a lookahead of 2. We call this configuration *approximate value iteration* (AVI). The second configuration samples for the first 10 hours with SAMPLE_REGRESSION and afterwards uses both SAMPLE_REGRESSION and SAMPLE_PROGRESSION. To evaluate the sampled states it uses EVALUATE_SEARCH. As search engine we use *greedy best-first search* with a timeout of 10s. In the beginning the value function is not good enough to solve sampled states far away from the goal. Therefore, the sampling starts with a walk length between 0 and 5. We double the maximum random walk length, if EVALUATE_SEARCH finds a plan for more than 95% of the sampled states. We observed that at some point further increasing the random walk length does not sample states further away from the goal, but just wastes computational time. Thus, we double at most 8 times the maximum walk length. We call this configuration *sampling search* (SaSe).

We run the training - including data generation - for 28 hours on 4 cores of an Intel Xeon E5-2600 processor with 3.8 GB of memory. For training the neural networks we used the Keras framework (Chollet 2015) with Tensorflow (Abadi et al. 2015) as back-end. We implemented the data generation and all searches in Fast Downward (Helmert 2006), and used Lab (Seipp et al. 2017) to setup our experiments.

Experiments

We train neural networks as heuristics to solve different tasks from the same state space and with the same goal. We evaluate our training procedure on the domains Ferber, Helmert, and Hoffmann (2020b) used. They selected a subset of tasks which they deemed hard enough to be interesting, but also easy enough for them to generate training data. We call their task selection *moderate tasks*. Because the data

Domain	AVI	SaSe	SL	Lama
blocks	0.0	0.0	**98.0**	96.8
depots	17.7	39.7	64.3	**98.7**
grid	51.0	86.0	74.0	**97.0**
npuzzle	1.0	1.5	0.0	**97.8**
pipesworld-notankage	29.8	50.4	92.8	**97.2**
rovers	25.8	35.8	12.5	**98.0**
scanalyzer-opt11-strips	83.3	33.3	77.7	**97.7**
storage	47.5	**71.5**	22.0	37.5
transport-opt14-strips	69.0	70.5	**98.0**	97.5
visitall-opt14-strips	13.0	30.7	0.7	**95.0**
Average	33.8	41.9	54.0	**91.3**

Table 1: Coverage of LAMA and greedy best-first search with heuristics trained using approximate value iteration (AVI), sampling search (SaSe), or supervised learning (SL) on the moderate tasks.

generation is not a bottleneck in our method, we also consider the harder tasks they skipped. We call these *hard tasks*. For every task selected, we have 50 different initial states for testing. We use the test states provided for the moderate tasks by Ferber, Helmert, and Hoffmann (2020b). For the other tasks, we generate new test states in the same way they did. We start at the initial state of the original task and perform a 200 step forward random walk.

For every task, we train a network using the approximate value iteration (AVI) configuration and a network using the sampling search (SaSe) configuration. To solve the test tasks, we use our neural networks as heuristics in an eager greedy best-first search. Exploratory experiments have shown that the eager version of greedy best-first search performs better with our heuristic than the lazy version. We run each search for 10 hours with a memory limit of 3.8 GB on a single core of an Intel Xeon Silver 4114. We use the first iteration of *LAMA* (Richter and Westphal 2010) as baseline. On the moderate tasks we also compare to the networks of Ferber, Helmert, and Hoffmann (2020b) used in a greedy best-first search. Because they used supervised learning, we call this baseline *supervised learning* (SL).

All code, benchmarks, and experimental results are online available (Ferber, Helmert, and Hoffmann 2020a).

Moderate Tasks

Table 1 shows the coverage of our configurations (AVI, SaSe) against the supervised learning (SL) and the LAMA baseline on the moderate tasks. On average LAMA outperforms all other techniques. Given enough time to generate its training data (400 hours) the supervised training approach solves more tasks than our *current* approach. From our two approaches, the sampling search configuration outperforms the approximate value iteration approach.

A more detailed view shows that whether an approach works well or not depends on the domain. There are some domains on which the supervised learning approach works well, but our reinforcement learning approach does not work at all (e.g. Blocksworld) and some domains where the reinforcement learning works better than the supervised learning(e.g. VisitAll). In Storage we outperform not only the supervised learning approach, but also LAMA.

Domain	AVI	SaSe	Lama
depots	15.1	6.9	**80.6**
grid	0.0	0.0	**90.0**
npuzzle	0.0	0.0	**84.0**
pipesworld-notankage	1.4	25.1	**68.7**
rovers	0.1	0.8	**97.7**
scanalyzer-opt11-strips	34.0	3.3	**98.7**
storage	18.8	**26.5**	11.0
visitall-opt14-strips	0.0	36.0	**98.0**
Average	8.7	12.3	**78.6**

Table 2: Coverage of LAMA and greedy best-first search with heuristics trained using approximate value iteration (AVI), sampling search (SaSe) on the hard tasks.

Figure 1 shows for each domain how the coverage increases over time. We see that LAMA quickly reaches its coverage limit. The supervised learning approach takes a bit longer. The two reinforcement learning approaches require the most time until they converge to their final coverage.

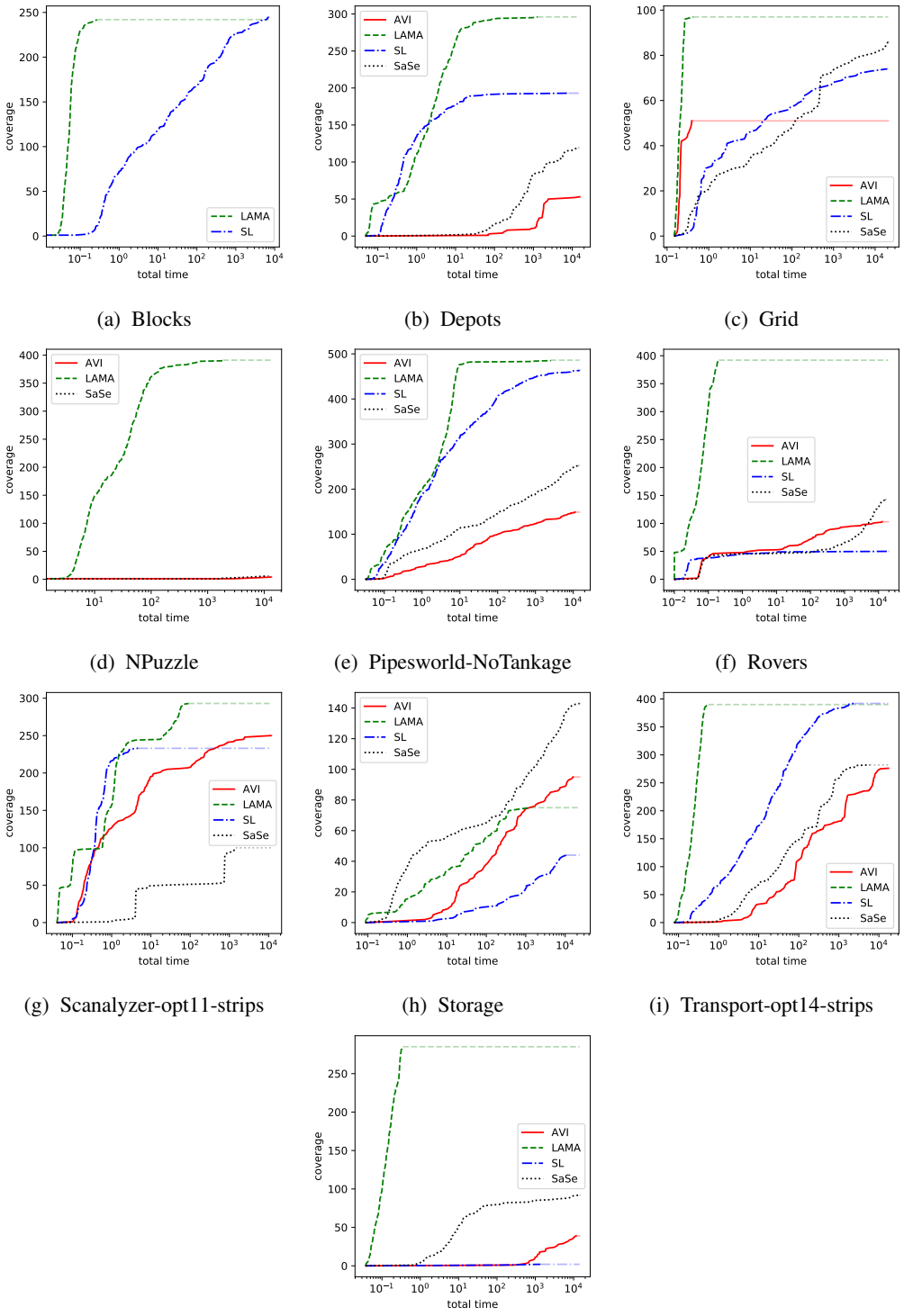
Hard Tasks

Table 2 shows the coverage on the hard tasks. The Blocksworld and Transport domains have no tasks in this category. The hard tasks show a similar picture than the moderate tasks. All techniques solve fewer tasks, but LAMA is still the best technique, and the sampling search configuration is still better than the approximate value iteration technique. Furthermore, in the Storage domain reinforcement learning outperforms LAMA.

Robustness

We observe that for some tasks within a domain our approaches solve either almost none or almost all states (see Table 3, columns *x1*). We speculate that the randomness during training sometimes produces good and sometimes bad models. To verify this, we train for the domains Depots and Scanalyzer four additional models per task. We run each of our five models on a fifth of the test states. If our assumption is correct, then we expect for the same task some models which solve almost all test states, and some models which solve almost no test states. Column *x5* of Table 3 shows for each of the five models how many test states they solves. As expected, most of the trained models solve either all or none of their test states and for the same task it is possible to obtain good and bad models. We also see that for some tasks it is more likely to train a good model than for other tasks.

This raises the question of what would be our performance if we could detect which models are good? For every task we select the model with the highest coverage out of the five models in column *x5* and use those on all test states. Columns *x1'* shows that this greatly increases our coverage.



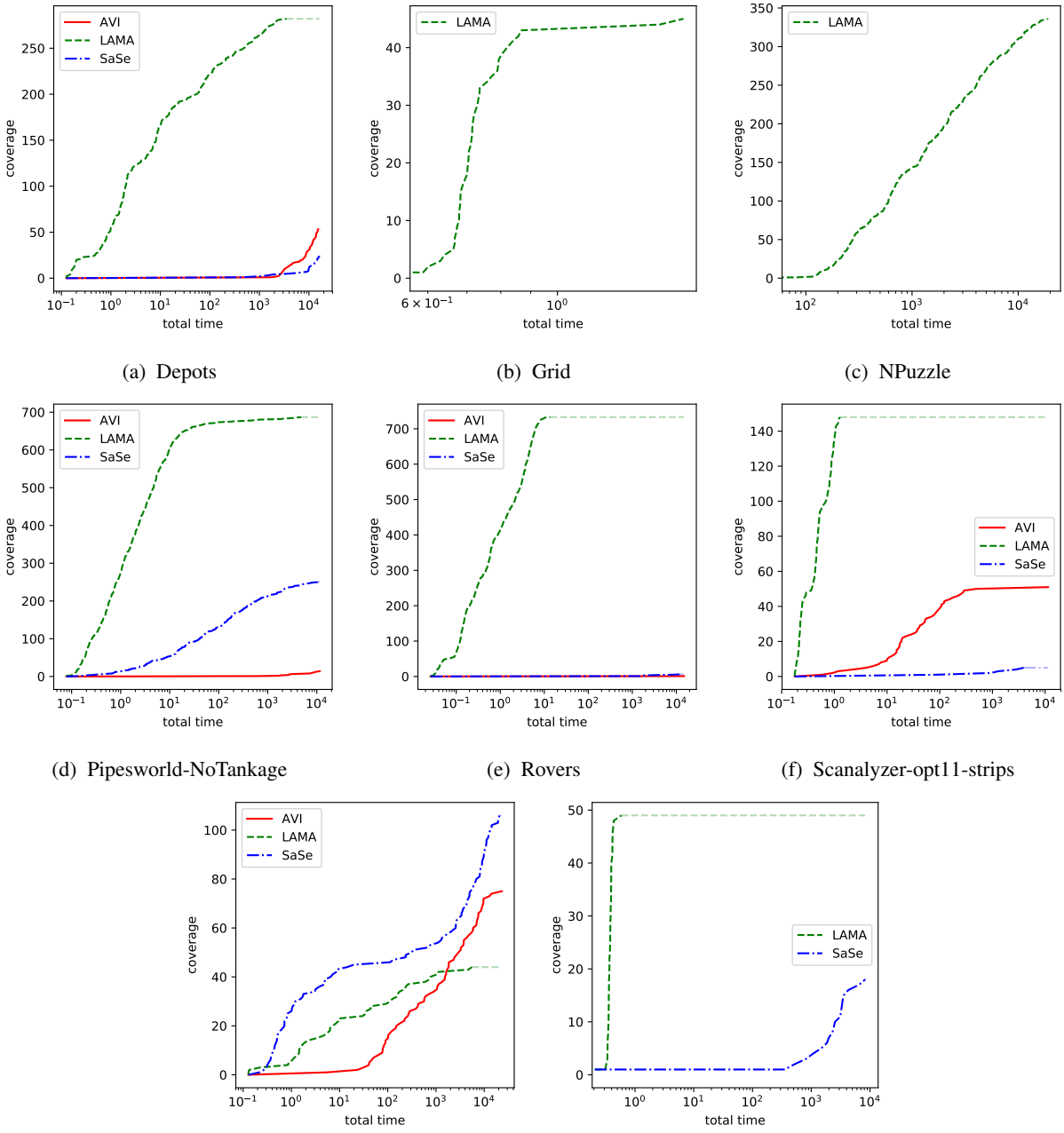
(a) Blocks (b) Depots (c) Grid

(d) NPuzzle (e) Pipesworld-NoTankage (f) Rovers

(g) Scanalyzer-opt11-strips (h) Storage (i) Transport-opt14-strips

(j) Visitall-opt14-strips

Figure 1: Cumulative coverage of LAMA and greedy best-first search with the heuristic trained using approximate value iteration (AVI), sampling search (SaSe), and the supervised learning (SL) baseline on the moderate tasks.



(a) Depots (b) Grid (c) NPuzzle

(d) Pipesworld-NoTankage (e) Rovers (f) Scanalyzer-opt11-strips

(g) Storage (h) Visitall-opt14-strips

Figure 2: Cumulative coverage of LAMA and greedy best-first search with the heuristic trained using approximate value iteration (AVI), sampling search (SaSe), and the supervised learning (SL) baseline on the hard tasks.

	AVI							SaSe						
Depots	x1	x5					x1'	x1	x5					x1'
p05	**50**	10	10	10	9	9	**50**	0	10	10	0	0	0	**41**
p06	0	0	0	0	0	0	0	0	1	1	1	0	0	**10**
p08	1	7	5	2	0	0	**28**	19	10	10	6	6	0	**47**
p09	0	8	0	0	0	0	**32**	**50**	10	1	0	0	0	44
p11	2	10	1	1	0	0	**50**	0	10	7	0	0	0	**49**
p12	0	0	0	0	0	0	0	**5**	0	0	0	0	0	3
p14	11	10	10	9	9	3	**50**	**1**	0	0	0	0	0	0
p15	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p16	0	9	7	0	0	0	**45**	**50**	10	10	10	0	0	**50**
p18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p19	**42**	7	2	2	0	0	**42**	18	6	3	1	0	0	**23**
p20	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p22	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum	106	150					**297**	143	123					**267**

	AVI							SaSe						
Scanalyzer	x1	x5					x1'	x1	x5					x1'
p07	0	2	0	0	0	0	**16**	0	0	0	0	0	0	0
p10	**50**	10	9	3	0	0	**50**	0	0	0	0	0	0	0
p13	**50**	10	9	9	9	9	**50**	**50**	10	10	10	10	10	**50**
p15	**50**	10	10	10	9	8	**50**	**50**	10	10	10	10	10	**50**
p16	**50**	10	10	10	10	10	**50**	0	0	0	0	0	0	0
p17	**50**	10	10	9	8	1	**50**	0	0	0	0	0	0	0
p18	1	8	8	3	0	0	**49**	0	0	0	0	0	0	0
p19	0	0	0	0	0	0	0	5	2	0	0	0	0	**11**
p20	**50**	10	9	9	9	9	**50**	0	10	0	0	0	0	**50**
Sum	301	288					**365**	105	112					**161**

Table 3: Absolute coverage for greedy best-first search using heuristics trained by the approximate value iteration (AVI) configuration and by the sampling search (SaSe) configuration. *x1* uses one model for all test states. *x5* uses five models (10 states per model). *x1'* uses the best model from *x5* on all test states. (Top) Shows the results for Depots. (Bottom) Shows the results for Scanalyzer.

	Moderate Tasks			Hard Tasks	
Domain	AVI	SaSe	SL	AVI	SaSe
depot	68.8	77.0	64.3	26.3	10.3
scanalyzer	88.7	50.0	77.7	66.0	7.3

Table 4: Coverage on Depots and Scanalyzer if the best models from Table 3 column x5 are used to solve all test states.

The performance of a model on a subset of test states approximates well the performance of the model on all test states. Table 4 shows how this increases the coverage fractions. For the moderate tasks of Depots this increases the coverage by 50% (AVI) resp. 40% (SaSe) and our approach would outperform the supervised learning baseline.

A followup question is, how can we detect whether a model will be good on the test states? We saw that a subset of test states approximates well the performance of model on all test states. Thus, a first approach could be to create an additional set of validation states which is independent of the test states. The performance of every trained model is evaluated on the validation states, and we select the model with the best performance on the validation states.

Conclusion

We presented two approaches to learn heuristics for fixed state spaces and goals using reinforcement learning. The first one uses approximate value iteration the other one uses a search in its data generation. We showed that our approaches can outperform the previous state of the art on some domains while requiring only $1/16$ of the time for training. Furthermore, our approach can easily be applied to hard planning tasks.

We observed that most of our trained models are either very good or very bad at solving the test states. We presented a naive test to detect good models and showed that this can drastically improve our coverage. In our next steps, we plan to examine how we can detect during training if a model is on its way to become a good or bad model. We expect to see the performance boost shown for Depots and Scanalyzer also on the other domains.

We also observed that increasing the random walk length to sample states farther away from the goal suffers from diminishing returns. Adding 10 additional steps to the walk does not lead us 10 steps farther away from the goal. We started preliminary experiments which use a known heuristic or the current value function as bias. We believe that biasing the random walk, especially with the trained value function, is an essential step for learning the heuristic estimates of large state spaces.

Acknowledgments

Patrick Ferber was funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science). This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan).

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Man'e, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Fern; a Vi'egas; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence* 1:356–363.

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *Proc. IJCAI 2013*, 2254–2260.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap learning of heuristic functions. In *Proc. SoCS 2010*, 52–60.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Computational Intelligence* 11(4):625–655.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Chollet, F. 2015. Keras. https://keras.io.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *AIJ* 221:73–114.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020a. Code, benchmarks and experiment data for the PRL 2020 workshop paper "Reinforcement Learning for Planning Heuristics". https://doi.org/10.5281/zenodo.4049617.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020b. Neural network heuristics for classical planning: A study of hyperparameter space. In *Proc. ECAI 2020*.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2020. Learning neural search policies for classical planning. In *Proc. ICAPS 2020*, 522–530.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, 1007–1012.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 planner abstracts*, 38–45.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM* 61(3):16:1–63.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *Proc. ICLR 2015*.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online planner selection with graph neural networks and adaptive scheduling. In *Proc. AAAI 2020*, 5077–5084.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, 246–249.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In *Proc. SoCS 2017*, 149–153.

Seipp, J. 2018. Fast Downward Remix. In *IPC-9 planner abstracts*, 74–76.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proc. AAAI 2019*, 7715–7723.

Bridging the gap between Markowitz planning and deep reinforcement learning

Eric Benhamou [1,2], **David Saltiel** [1,3], **Sandrine Ungari** [4], **Abhishek Mukhopadhyay** [5]
[1] AI Square Connect, France, {eric.benhamou,david.saltiel}@aisquareconnect.com
[2]MILES, LAMSADE, Dauphine university, France, eric.benhamou@lamsade.dauphine.fr
[3] LISIC, ULCO, France, david.saltiel@univ-littoral.fr
[4] Societe Generale, Cross Asset Quantitative Research, UK,
[5] Societe Generale, Cross Asset Quantitative Research, France,
{sandrine.ungari,abhishek.mukhopadhyay}@sgcib.com

Abstract

While researchers in the asset management industry have mostly focused on techniques based on financial and risk planning techniques like Markowitz efficient frontier, minimum variance, maximum diversification or equal risk parity, in parallel, another community in machine learning has started working on reinforcement learning and more particularly deep reinforcement learning to solve other decision making problems for challenging task like autonomous driving, robot learning, and on a more conceptual side games solving like Go. This paper aims to bridge the gap between these two approaches by showing Deep Reinforcement Learning (DRL) techniques can shed new lights on portfolio allocation thanks to a more general optimization setting that casts portfolio allocation as an optimal control problem that is not just a one-step optimization, but rather a continuous control optimization with a delayed reward. The advantages are numerous: (i) DRL maps directly market conditions to actions by design and hence should adapt to changing environment, (ii) DRL does not rely on any traditional financial risk assumptions like that risk is represented by variance, (iii) DRL can incorporate additional data and be a multi inputs method as opposed to more traditional optimization methods. We present on an experiment some encouraging results using convolution networks.

Introduction

In asset management, there is a gap between mainstream used methods and new machine learning techniques around reinforcement learning and in particular deep reinforcement learning. The former methods rely on financial risk optimization and solve the planning problem of the optimal portfolio as a single step optimization question. The latter do not make any assumptions about risk, do a more involving multi-steps optimization and solve complex and challenging tasks like autonomous driving (Wang, Jia, and Weng 2018), learning advanced locomotion and manipulation skills from raw sensory inputs (Levine et al. 2015; 2016; Schulman et al. 2015a; 2017; Lillicrap et al. 2015) or on a more conceptual side for reaching supra human level in popular games like Atari (Mnih et al. 2013), Go (Silver et al. 2016; 2017), StarCraft II (Vinyals et al. 2019), etc ... One of the reasons often put forward for this situation is that asset management researchers have mostly been trained with an econometric and financial mathematics background, while the deep reinforcement learning community has been mostly trained in computer science and robotics, leading to two distinctive research communities that do not interact much between each other. In this paper, we aim to present the various approaches to show similarities and differences to bridge the gap between these two approaches. Both methods can help solving the decision making problem of finding the optimal portfolio allocation weights.

Related works

As this paper aims at bridging the gap between traditional asset management portfolio selection methods and deep reinforcement learning, there are too many works to be cited.

On the traditional methods side, the seminal work is (Markowitz 1952) that has led to various extensions like minimum variance (Chopra and Ziemba 1993; Haugen and Baker 1991), (Kritzman 2014), maximum diversification (Choueifaty and Coignard 2008; Choueifaty, Froidure, and Reynier 2012), maximum decorrelation (Christoffersen et al. 2010), risk parity (Maillard, Roncalli, and Teïletche 2010; Roncalli and Weisang 2016). We will review these works in the section entitled *Traditional methods*.

On the reinforcement learning side, the seminal book is (Sutton and Barto 2018). The field of deep reinforcement learning is growing every day at an unprecedented pace, making the citation exercise complicated. But in terms of breakthroughs of deep reinforcement learning, one can cite the work around Atari games from raw pixel inputs (Mnih et al. 2013; 2015), Go (Silver et al. 2016; 2017), StarCraft II (Vinyals et al. 2019), learning advanced locomotion and manipulation skills from raw sensory inputs (Levine et al. 2015; 2016) (Schulman et al. 2015a; 2015b; 2017; Lillicrap et al. 2015), autonomous driving (Wang, Jia, and Weng 2018) and robot learning (Gu et al. 2017).

On the application of deep reinforcement learning methods to portfolio allocations, there is already a growing interest as recent breakthroughs has put growing emphasis on this method. Hence, the field is growing very rapidly and

survey like (Fischer 2018) are already out dated. Driven initially mostly by applications to crypto currencies and Chinese financial markets (Jiang and Liang 2016; Zhengyao et al. 2017; Liang et al. 2018; Yu et al. 2019; Wang and Zhou 2019; Saltiel et al. 2020; Benhamou et al. 2020b; 2020a; 2020c), the field is progressively taking off on other assets (Kolm and Ritter 2019; Liu et al. 2020; Ye et al. 2020; Li et al. 2019; Xiong et al. 2019). More generally, DRL has recently been applied to other problems than portfolio allocation. For instance, (Deng et al. 2016; Zhang, Zohren, and Roberts 2019; Huang 2018; Théate and Ernst 2020; Chakraborty 2019; Nan, Perumal, and Zaiane 2020; Wu et al. 2020) tackle the problem of direct trading strategies (Bao and yang Liu 2019) handles the one of multi agent trading while (Ning, Lin, and Jaimungal 2018) examine optimal execution.

Traditional methods

We are interested in finding an optimal portfolio which makes the planning problem quite different from standard planning problem where the aim is to plan a succession of tasks. Typical planning algorithms are variations around STRIPS (Fikes and Nilsson 1971), that starts by analysis ending goals and means, builds the corresponding graph and finds the optimal graph. Indeed we start from the goals to achieve and try to find means that can lead to them. New work like Graphplan as presented in (Blum and Furst 1995) uses a novel planning graph, to reduce the amount of search needed, while hierarchical task network (HTN) planning leverages the classification to structure networks and hence reduce the number of graph searches. Other algorithms like search algorithm as A^*, B^*, weighted A^* or for full graph search, branch and bound and its extensions, as well as evolutionary algorithms like particle swarm, CMA-ES are also used widely in AI planning etc.. However, when it comes to portfolio allocation, standard methods used by practitioners rely on more traditional financial risk reward optimization problems and follows rather the Markowitz approach as presented below.

Markowitz

The intuition of Markowitz portfolio is to be able to compare various assets and assemble them taking into account both return and risk. Comparing just returns of some financial assets would be too naive. One has to take into account in her/his investment decision returns with associated risk. Risk is not an easy concept. in Modern Portfolio Theory (MPT), risk is represented by the variance of the asset returns. If we take various financial assets and display their returns and risk as in figure 1, we can find an efficient frontier. Indeed there exists an efficient frontier represented by the red dot line.

Mathematically, if we denote by $w = (w_1, ..., w_l)$ the allocation weights with $1 \geq w_i \geq 0$ for $i = 0...l$, summarized by $1 \geq w \geq 0$, with the additional constraints that these weights sum to 1: $\sum_{i=1}^{l} w_i = 1$, we can see this portfolio allocation question as an optimization.
Let $\mu = (\mu_1, ..., \mu_l)^T$ be the expected returns for our l strategies and Σ the matrix of variance covariances of the l strategies' returns. Let r_{min} be the minimum expected return. The Markowitz optimization problem to solve is to minimize the risk given a target of minimum expected return as follows:

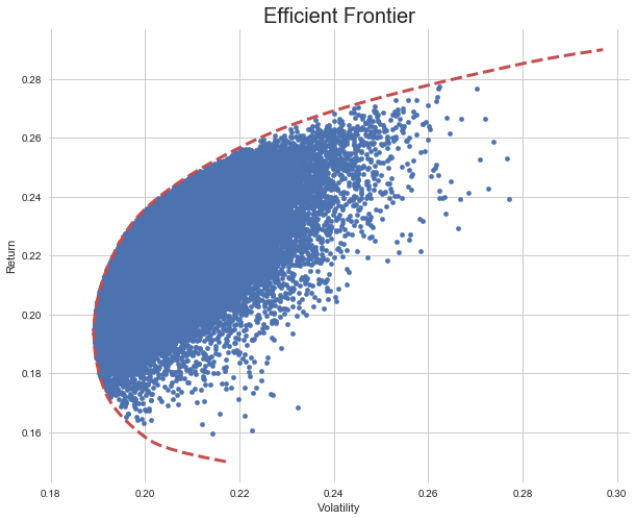


Figure 1: Markowitz efficient frontier for the GAFA: returns taken from 2017 to end of 2019

$$\begin{aligned} \underset{w}{\text{Minimize}} \quad & w^T \Sigma w \\ \text{subject to} \quad & \mu^T w \geq r_{min}, \sum_{i=1...l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{1}$$

It is solved by standard quadratic programming. Thanks to duality, there is an equivalent maximization with a given maximum risk σ_{max} for wich the problem writes as follows:

$$\begin{aligned} \underset{w}{\text{Maximize}} \quad & \mu^T w \\ \text{subject to} \quad & w^T \Sigma w \leq \sigma_{max}, \sum_{i=1...l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{2}$$

Minimum variance portfolio

This seminal model has led to numerous extensions where the overall idea is to use a different optimization objective. As presented in (Chopra and Ziemba 1993; Haugen and Baker 1991), (Kritzman 2014), we can for instance be interested in just minimizing risk (as we are not so much interested in expected returns), which leads to the minimum variance portfolio given by the following optimization program:

$$\begin{aligned} \underset{w}{\text{Minimize}} \quad & w^T \Sigma w \\ \text{subject to} \quad & \sum_{i=1...l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{3}$$

Maximum diversification portfolio Denoting by σ the volatilities of our l strategies, whose values are the diagonal elements of the covariance matrix Σ: $\sigma = (\Sigma_{i,i})_{i=1..l}$, we

can shoot for maximum diversification with the diversification of a portfolio defined as follows: $D = \frac{w^T \sigma}{\sqrt{w^T \sum w}}$. We then solve the following optimization program as presented in (Choueifaty and Coignard 2008; Choueifaty, Froidure, and Reynier 2012)

$$\begin{aligned} \underset{w}{\text{Maximize}} \quad & \frac{w^T \sigma}{\sqrt{w^T \sum w}} \\ \text{subject to} \quad & \sum_{i=1\ldots l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{4}$$

The concept of diversification is simply the ratio of the weighted average of volatilities divided by the portfolio volatility.

Maximum decorrelation portfolio Following (Christoffersen et al. 2010) and denoting by C the correlation matrix of the portfolio strategies, the maximum decorrelation portfolio is obtained by finding the weights that provide the maximum decorrelation or equivalently the minimum correlation as follows:

$$\begin{aligned} \underset{w}{\text{Minimize}} \quad & w^T C w \\ \text{subject to} \quad & \sum_{i=1\ldots l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{5}$$

Risk parity portfolio Another approach following risk parity (Maillard, Roncalli, and Teïletche 2010; Roncalli and Weisang 2016) is to aim for more parity in risk and solve the following optimization program

$$\begin{aligned} \underset{w}{\text{Minimize}} \quad & \frac{1}{2} w^T \Sigma w - \frac{1}{n} \sum_{i=1}^{l} \ln(w_i) \\ \text{subject to} \quad & \sum_{i=1\ldots l} w_i = 1, 1 \geq w \geq 0 \end{aligned} \tag{6}$$

All these optimization techniques are the usual way to solve the planning question of getting the best portfolio allocation. We will see in the following section that there are many alternatives leveraging machine learning that remove cognitive bias of risk and are somehow more able to adapt to changing environment.

Reinforcement learning

Previous financial methods treat the portfolio allocation planning question as a one-step optimization problem, with convex objective functions. There are multiple limitations to this approach:

- they do not relate market conditions to portfolio allocation dynamically.
- they do not take into account that the result of the portfolio allocation may potentially be evaluated much later.
- they make a strong assumptions about risk.

What if we could cast this portfolio allocation planning question as a dynamic control problem where we have some market information and needs to decide at each time step the optimal portfolio allocation problem and evaluate the result with delayed reward? What if we could move from static portfolio allocation to optimal control territory where we can change our portfolio allocation dynamically when market conditions changes. Because the community of portfolio allocation is quite different from the one of reinforcement learning, this approach has been ignored for quite some time even though there is a growing interest for the use of reinforcement learning and deep reinforcement learning over the last few years. We will present here in greater details what deep reinforcement is in order to suggest more discussions and exchanges between these two communities.

Contrary to supervised learning, reinforcement learning do not try to predict future returns. It does not either try to learn the structure of the market implicitly. Reinforcement learning does more: it directly learns the optimal policy for the portfolio allocation in connection with the dynamically changing market conditions.

Deep Reinforcement Learning Intuition

As it name stands for, Deep Reinforcement Learning (DRL) is the combination of Reinforcement Learning (RL) and Deep (D). The usage of deep learning is to represent the policy function in RL. In a nutshell, the setting for applying RL to portfolio management can be summarized as follows:

- current knowledge of the financial markets is formalized via a state variable denoted by s_t.
- Our planning task which is to find an optimal portfolio allocation can be thought as taking an action a_t on this market. This action is precisely the decision of the current portfolio allocation (also called portfolio weights).
- once we have decided the portfolio allocation, we observe the next state s_{t+1}.
- we use a reward to evaluate the performance of our actions. In our particular setting, we can compute this reward only at the the final time of our episode, making it quite special compared to standard reinforcement learning problem. We denote this reward by R_T where T is the final time of our episode. This reward R_T is in a sense similar to our objective function in traditional methods. A typical reward is the final portfolio net performance. It could be obviously other financial performance evaluation criteria like Sharpe, Sortino ratio, etc..

Following standard RL, we model our problem to solve with a Markov Decision Process (MDP) as in (Sutton and Barto 2018). MDP assumes that the agent knows all the states of the environment and has all the information to make the optimal decision in every state. The Markov property implies in addition that knowing the current state is sufficient. MDP assumes a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{P}$ is the state action to next state transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and $\mathcal{R}$ is the immediate reward. The goal of the agent is to learn a policy that maps states to the optimal action

$\pi : \mathcal{S} \rightarrow \mathcal{A}$ and that maximizes the expected discounted reward $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_t]$.

The concept of using deep network is to represent the function that relates dynamically the states to the action called in RL the policy and denoted by $\vec{a}_t = \pi(s_t)$. This function is represented by deep network because of the universal approximation theorem that states that any function can be represented by a deep network provided we have enough layers and nodes. Compared to traditional methods that only solve a one step optimization, we are solving the following dynamic control optimization program:

$$\begin{aligned} \underset{\pi(.)}{\text{Maximize}} \quad & \mathbb{E}[R_T] \\ \text{subject to} \quad & a_t = \pi(s_t) \end{aligned} \tag{7}$$

Note that we maximize the expected value of the cumulated reward $\mathbb{E}[R_T]$ because we are operating in a stochastic environment. To make things simpler, let us assume that the cumulated reward is the final portfolio net performance. Let us write P_t the price at time t of our portfolio, and its return at time t: r_t^P and the portfolio assets return vector at time t: $\vec{r}_t$. The final net performance writes as $P_T/P_0 - 1 = \prod_{t=1}^{T}(1 + r_t^P) - 1$. The returns r_t^P is a function of our planning action a_t as follows: $(1+r_t^P) = 1+\langle \vec{a}_t, \vec{r}_t \rangle$ where $\langle \cdot, \cdot \rangle$ is the standard inner product of two vectors. In addition if we recall that the policy is parametrized by some deep network parameters, θ: $a_t = \pi_\theta(s_t)$, we can make our optimization problem slightly more detailed as follows:

$$\begin{aligned} \underset{\theta}{\text{Maximize}} \quad & \mathbb{E}\left[\prod_{t=1}^{T}(1 + \langle \vec{a}_t, \vec{r}_t \rangle)\right] \\ \text{subject to} \quad & a_t = \pi_\theta(s_t). \end{aligned} \tag{8}$$

It is worth noticing that compared to previous traditional planning methods (optimization 1, 3, 4, 5 or 5), the underlying optimization problem in RL 7 and its rewritting in terms of deep network parameters θ as presented in 8 have many differences:

- First, we are trying to optimize a function π and not simple weights w_i. Although this function at the end is represented by a deep neural network that has admittely also weights, this is conceptually very different as we are optimizing in the space of functions $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that is a much bigger space than simply $\mathbb{R}^l$.
- Second, it is a multi time step optimization at it involves results from time $t = 1$ to $t = T$, making it also more involving.

Partially Observable Markov Decision Process

If there is in addition some noise in our data and we are not able to observe the full state, it is better to use Partially Observable Markov Decision Process (POMDP) as presented initially in (Astrom 1969). In POMDP, only a subset of the information of a given state is available. The partially-informed agent cannot behave optimally. He uses a window of past observations to replace states as in a traditional MDP.

Mathematically, POMDP is a generalization of MDP. POMPD adds two more variables in the tuple, $\mathcal{O}$ and $\mathcal{Z}$ where $\mathcal{O}$ is the set of observations and $\mathcal{Z}$ is the observation transition function $\mathcal{Z} : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \rightarrow [0, 1]$. At each time, the agent is asked to take an action $a_t \in \mathcal{A}$ in a particular environment state $s_t \in \mathcal{S}$, that is followed by the next state s_{t+1} with $\mathcal{P}(s_{t+1}|s_t, a_t)$. The next state s_{t+1} is not observed by the agent. It rather receives an observation $o_{t+1} \in \mathcal{O}$ on the state s_{t+1} with probability $Z(o_{t+1}|s_{t+1}, a_t)$.

From a practical standpoint, the general RL setting is modified by taking a pseudo state formed with a set of past observations $(o_{t-n}, o_{t-n-1}, \ldots, o_{t-1}, o_t)$. In practice to avoid large dimension and the curse of dimension, it is useful to reduce this set and take only a subset of these past observations with $j < n$ past observations, such that $0 < i_1 < \ldots < i_j$ and $i_k \in \mathbb{N}$ is an integer. The set $\delta_1 = (0, i_1, \ldots, i_j)$ is called the observation lags. In our experiment we typically use lag periods like (0, 1, 2, 3, 4, 20, 60) for daily data, where $(0, 1, 2, 3, 4)$ provides last week observation, 20 is for the one-month ago observation (as there is approximately 20 business days in a month) and 60 the three-month ago observation.

Observations

Regular observations There are two types of observations: regular and contextual information. Regular observations are data directly linked to the problem to solve. In the case of an asset management framework, regular observations are past prices observed over a lag period $\delta = (0 < i_1 < \ldots < i_j)$. To normalize data, we rather use past returns computed as $r_t^k = \frac{p_t^k}{p_{t-1}^k} - 1$ where p_t^k is the price at time t of the asset k. To give information about regime changes, our trading agent receives also empirical standard deviation computed over a sliding estimation window denoted by d as follows $\sigma_t^k = \sqrt{\frac{1}{d}\sum_{u=t-d+1}^{t}(r_u - \mu)^2}$, where the empirical mean μ is computed as $\mu = \frac{1}{d}\sum_{u=t-d+1}^{t} r_u$. Hence our regular observations is a three dimensional tensor $A_t = \left[A_t^1, A_t^2\right]$

$$\text{with} \quad A_t^1 = \begin{pmatrix} r_{t-i_j}^1 \ \ldots \ r_t^1 \\ \ldots \ \ldots \ \ldots \\ r_{t-i_j}^m \ \ldots \ r_t^m \end{pmatrix}, \ A_t^2 = \begin{pmatrix} \sigma_{t-i_j}^1 \ \ldots \ \sigma_t^1 \\ \ldots \ \ldots \ \ldots \\ \sigma_{t-i_j}^m \ \ldots \ \sigma_t^m \end{pmatrix}$$

This setting with two layers (past returns and past volatilities) is quite different from the one presented in (Jiang and Liang 2016; Zhengyao et al. 2017; Liang et al. 2018) that uses different layers representing closing, open high low prices. There are various remarks to be made. First, high low information does not make sense for portfolio strategies that are only evaluated daily, which is the case of all the funds. Secondly, open high low prices tend to be highly correlated creating some noise in the inputs. Third, the concept of volatility is crucial to detect regime change and is surprisingly absent from these works as well as from other works like (Yu et al. 2019; Wang and Zhou 2019; Liu et al. 2020; Ye et al. 2020; Li et al. 2019; Xiong et al. 2019).

Context observation Contextual observations are additional information that provide intuition about current con-

text. For our asset manager, they are other financial data not directly linked to its portfolio assumed to have some predictive power for portfolio assets. Contextual observations are stored in a 2D matrix denoted by C_t with stacked past p individual contextual observations. Among these observations, we have the maximum and minimum portfolio strategies return and the maximum portfolio strategies volatility. The latter information is like for regular observations motivated by the stylized fact that standard deviations are useful features to detect crisis. The contextual state writes as $C^t = \begin{pmatrix} c_t^1 \ ... \ c_{t-i_k}^1 \\ ... \ ... \ ... \\ c_t^p \ .... \ c_{t-i_k}^p \end{pmatrix}$. The matrix nature of contextual states C_t implies in particular that we will use 1D convolutions should we use convolutional layers. All in all, observations that are augmented observations, write as $O_t = [A_t, C_t]$, with $A_t = [A_t^1, A_t^2]$ that will feed the two sub-networks of our global network.

Action

In our deep reinforcement learning the augmented asset manager agent needs to decide at each period in which hedging strategy it invests. The augmented asset manager can invest in l strategies that can be simple strategies or strategies that are also done by asset management agent. To cope with reality, the agent will only be able to act after one period. This is because asset managers have a one day turn around to change their position. We will see on experiments that this one day turnaround lag makes a big difference in results. As it has access to l potential hedging strategies, the output is a l dimension vector that provides how much it invest in each hedging strategy. For our deep network, this means that the last layer is a softmax layer to ensure that portfolio weights are between 0 and 100% and sum to 1, denoted by $(p_t^1, ..., p_t^l)$. In addition, to include leverage, our deep network has a second output which is the overall leverage that is between 0 and a maximum leverage value (in our experiment 3), denoted by lvg_t. Hence the final allocation is given by $lvg_t \times (p_t^1, ..., p_t^l)$.

Reward

In terms of reward, we are considering the net performance of our portfolio from t_0 to the last train date t_T computed as follows: $\frac{P_{t_T}}{P_{t_0}} - 1$.

Multi inputs and outputs

We display in figure 2 the architecture of our network. Because we feed our network with both data from the strategies to select but also contextual information, our network is a multiple inputs network.

Additionally, as we want from these inputs to provide not only percentage in the different hedging strategies (with a softmax activation of a dense layer) but also the overall leverage (with a dense layer with one single output neurons), we also have a multi outputs network. Additional hyperparameters that are used in the network as L2 regularization with a coefficient of 1e-8.

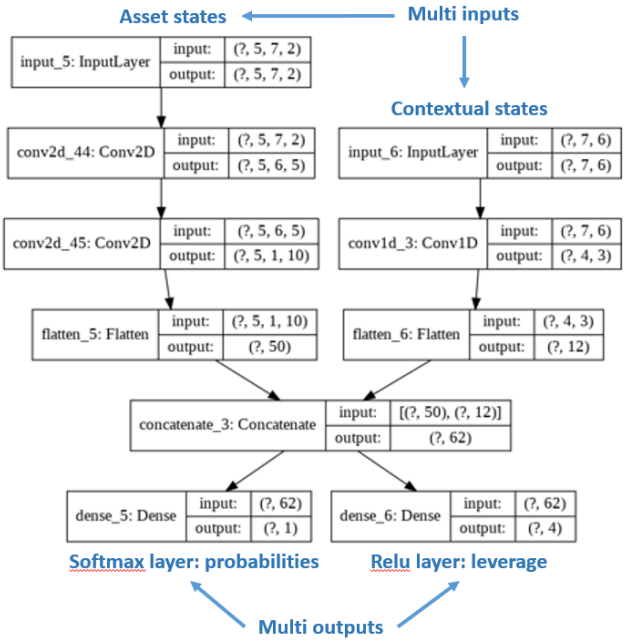


Figure 2: network architecture obtained via tensorflow plot-model function. Our network is very different from standard DRL networks that have single inputs and outputs. Contextual information introduces a second input while the leverage adds a second output

Convolution networks

Because we want to extract some features implicitly with a limited set of parameters, and following (Liang et al. 2018), we use convolution network that perform better than simple full connected layers. For our so called *asset states* named like that because there are the part of the states that relates to the asset, we use two layers of convolutional network with 5 and 10 convolutions. These parameters are found to be efficient on our validation set. In contrast, for the contextual states part, we only use one layer of convolution networks with 3 convolutions. We flatten our two sub network in order to concatenate them into a single network.

Adversarial Policy Gradient

To learn the parameters of our network depicted in 2, we use a modified policy gradient algorithm called adversarial as we introduce noise in the data as suggested in (Liang et al. 2018).. The idea of introducing noise in the data is to have some randomness in each training to make it more robust. This is somehow similar to drop out in deep networks where we randomly perturb the network by randomly removing some neurons to make it more robust and less prone to overfitting. Here, we are perturbing directly the data to create this stochasticity to make the network more robust. A policy is a mapping from the observation space to the action space, $\pi : \mathcal{O} \rightarrow \mathcal{A}$. To achieve this, a policy is specified by a deep network with a set of parameters $\vec{\theta}$. The action is a vector function of the observation given the parameters: $\vec{a}_t =$

$\pi_{\vec{\theta}}(\boldsymbol{o}_t)$. The performance metric of $\pi_{\vec{\theta}}$ for time interval $[0,t]$ is defined as the corresponding total reward function of the interval $J_{[0,t]}(\pi_{\vec{\theta}}) = R\left(\vec{o}_1, \pi_{\vec{\theta}}(o_1), \cdots, \vec{o}_t, \pi_{\vec{\theta}}(o_t), \vec{o}_{t+1}\right)$. After random initialization, the parameters are continuously updated along the gradient direction with a learning rate λ: $\vec{\theta} \longrightarrow \vec{\theta} + \lambda \nabla_{\vec{\theta}} J_{[0,t]}(\pi_{\vec{\theta}})$. The gradient ascent optimization is done with standard Adam (short for Adaptive Moment Estimation) optimizer to have the benefit of adaptive gradient descent with root mean square propagation (Kingma and Ba 2014). The whole process is summarized in algorithm 1.

Algorithm 1 Adversarial Policy Gradient

```
Input: initial policy parameters θ, empty replay buffer D
repeat
    reset replay buffer
    while not terminal do
        Observe observation o and select action a = π_θ(o)
        with probability p and random action with proba-
        bility 1 − p,
        Execute a in the environment
        Observe next observation o', reward r, and done
        signal d to indicate whether o' is terminal
        apply noise to next observation o'
        store (o, a, o') in replay buffer D
        if Terminal then
            for however many updates in D do
                compute final reward R
            end for
            update network parameter with Adam gradient
            ascent θ⃗ ⟶ θ⃗ + λ∇_θ⃗ J_[0,t](π_θ⃗)
        end if
    end while
until convergence
```

In our gradient ascent, we use a learning rate of 0.01, an adversarial Gaussian noise with a standard deviation of 0.002. We do up to 500 maximum iterations with an early stop condition if on the train set, there is no improvement over the last 50 iterations.

Experiments

Goal of the experiment

We are interested in planing a hedging strategy for a risky asset. The experiment is using daily data from 01/05/2000 to 19/06/2020 for the MSCI and 4 SG-CIB proprietary systematic strategies. The risky asset is the MSCI world index whose daily data can be found on Bloomberg. We choose this index because it is a good proxy for a wide range of asset manager portfolios. The hedging strategies are 4 SG-CIB proprietary systematic strategies further described below. Training and testing are done following extending walk forward analysis as presented in (Benhamou et al. 2020b; 2020c; 2020a) with initial training from 2000 to end of 2006 and testing in a rolling 1 year period. Hence, there are 14 training and testing periods, with the different testing period corresponding to all the years from 2007 to 2020 and training done for period starting in 2000 and ending one day before the start of the testing period.

Data-set description

Systematic strategies are similar to asset managers that invest in financial markets according to an adaptive, predefined trading rule. Here, we use 4 SG CIB proprietary 'hedging strategies', that tend to perform when stock markets are down:

- Directional hedges - react to small negative return in equities,
- Gap risk hedges - perform well in sudden market crashes,
- Proxy hedges - tend to perform in some market configurations, like for example when highly indebted stocks under-perform other stocks,
- Duration hedges - invest in bond market, a classical diversifier to equity risk in finance.

The underlying financial instruments vary from put options, listed futures, single stocks, to government bonds. Some of those strategies are akin to an insurance contract and bear a negative cost over the long run. The challenge consists in balancing cost versus benefits.

In practice, asset managers have to decide how much of these hedging strategies are needed on top of an existing portfolio to achieve a better risk reward. The decision making process is often based on contextual information, such as the economic and geopolitical environment, the level of risk aversion among investors and other correlation regimes. The contextual information is modeled by a large range of features :

- the level of risk aversion in financial markets, or market sentiment, measured as an indicator varying between 0 for maximum risk aversion and 1 for maximum risk appetite,
- the bond equity historical correlation, a classical ex-post measure of the diversification benefits of a duration hedge, measured on a 1 month, 3 month and 1 year rolling window,
- The credit spreads of global corporate - investment grade, high yield, in Europe and in the US - known to be an early indicator of potential economic tensions,
- The equity implied volatility, a measure if the 'fear factor' in financial market,
- The spread between the yield of Italian government bonds and the German government bond, a measure of potential tensions in the European Union,
- The US Treasury slope, a classical early indicator for US recession,
- And some more financial variables, often used as a gauge for global trade and activity: the dollar, the level of rates in the US, the estimated earnings per shares (EPS).

A cross validation step selects the most relevant features. In the present case, the first three features are selected. The

rebalancing of strategies in the portfolio comes with transaction costs, that can be quite high since hedges use options. Transactions costs are like frictions in physical systems. They are taken into account dynamically to penalise solutions with a high turnover rate.

Evaluation metrics

Asset managers use a wide range of metrics to evaluate the success of their investment decision. For a thorough review of those metrics, see for example (Cogneau and Hübner 2009). The metrics we are interested in for our hedging problem are listed below:

- annualized return defined as the average annualized compounded return,
- annualized daily based Sharpe ratio defined as the ratio of the annualized return over the annualized daily based volatility μ/σ,
- Sortino ratio computed as the ratio of the annualized return overt the downside standard deviation,
- maximum drawdown (max DD) computed as the maximum of all daily drawdowns. The daily drawdown is computed as the ratio of the difference between the running maximum of the portfolio value defined as $RM_T = \max_{t=0..T}(P_t)$ and the portfolio value over the running maximum of the portfolio value. Hence the drawdon at time T is given by $DD_T = (RM_T - P_T)/RM_T$ while the maximum drawdown $MDD_T = \max_{t=0..T}(DD_t)$. It is the maximum loss in return that an investor will incur if she/he invested at the worst time (at peak).

Results and discussion

Overall, the DRL approach achieves much better results than traditional methods as shown in table 1, except for the maximum drawdown (max DD). Because time horizon is important in the comparison we provide risk measures for the last 2 and 5 years to emphasize that the DRL approach seems more robust than traditional portfolio allocation methods.

When plotting performance results from 2007 to 2020 as shown in figure 3, we see that DRL model is able to deviate upward from the risky asset continuously, indicating a steady performance. In contrast, other financial models are not able to keep their marginal over-performance over time with respect to the risky asset and end slightly below the risky asset.

Allocation chosen by models

The reason of the stronger performance of DRL comes from the way it chooses its allocation. Contrarily to standard financial methods that play the diversification as shown in figure 4, DRL aims at choosing a single hedging strategy most of the time and at changing it dynamically, should the financial market conditions change. In a sense, DRL is doing some cherry picking by selecting what it thinks is the best hedging strategy.

In contrast, traditional models like Markowitz, minimum variance, maximum diversification, maximum decorrelation

Table 1: Models comparison over 2 and 5 years

	2 Years			
	return	Sortino	Sharpe	max DD
Risky asset	8.27%	0.39	0.36	- 0.34
DRL	**20.64%**	**0.94**	**0.96**	- 0.27
Markowitz	-0.25%	- 0.01	- 0.01	- 0.43
MinVariance	-0.22%	- 0.01	- 0.01	- 0.43
MaxDiversification	0.24%	0.01	0.01	- 0.43
MaxDecorrel	14.42%	0.65	0.63	- 0.21
RiskParity	14.17%	0.73	0.72	**-0.19**
	5 Years			
	return	Sortino	Sharpe	max DD
Risky asset	9.16%	0.57	0.54	- 0.34
DRL	**16.95%**	**1.00**	**1.02**	- 0.27
Markowitz	1.48%	0.07	0.06	- 0.43
MinVariance	1.56%	0.08	0.06	- 0.43
MaxDiversification	1.77%	0.08	0.07	- 0.43
MaxDecorrel	7.65%	0.44	0.39	- 0.21
RiskParity	7.46%	0.48	0.43	**-0.19**

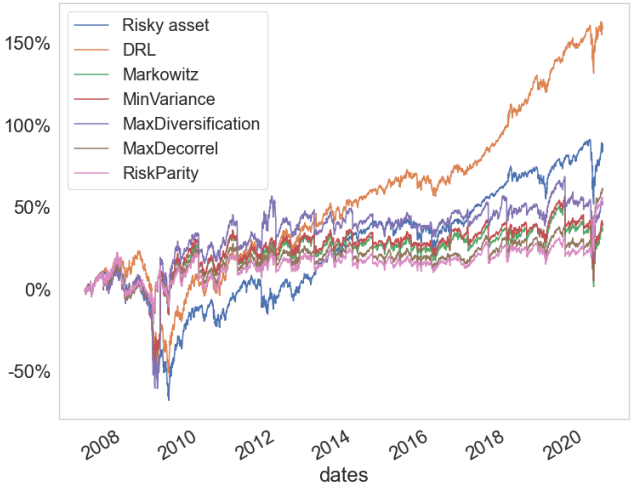


Figure 3: performance of all models

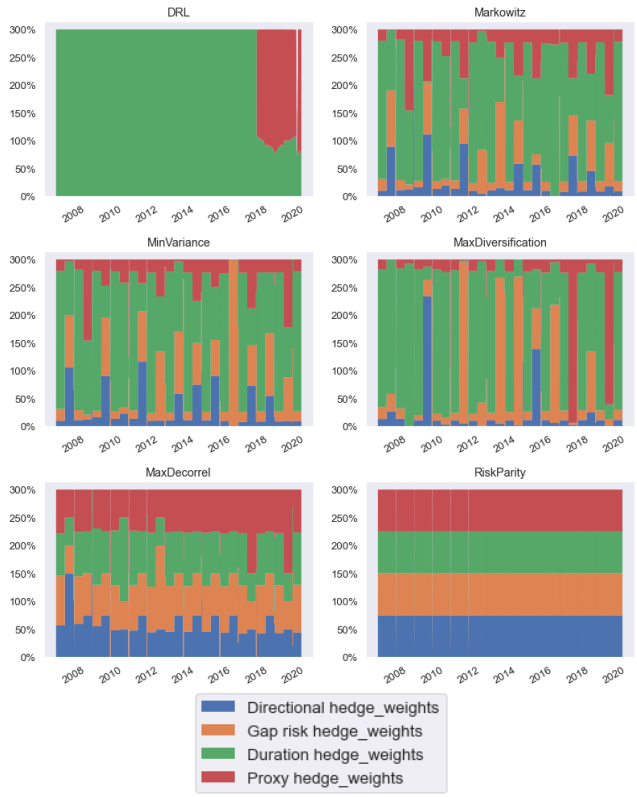


Figure 4: weights for all models

and risk parity provides non null weights for all our hedging strategies and do not do cherry picking at all. They are neither able to change the leverage used in the portfolio as opposed to DRL model.

Adaptation to the Covid Crisis

The DRL model can change its portfolio allocation should the market conditions change. This is the case from 2018 onwards, with a short deleveraging window emphasized by the small blank disruption during the Covid crisis as shown in figure 5. We observe in this figure where we have zoomed over year 2020, that the DRL model is able to reduce leverage from 300 % to 200 % during the Covid crisis (end of February 2020 to start of April 2020). This is a unique feature of our DRL model compared to traditional financial planning models that do not take leverage into account and keeps a leverage of 300 % regardless of market conditions.

Benefits of DRL

As illustrated by the experiment, the advantages of DRL are numerous: (i) DRL maps directly market conditions to actions by design and hence should adapt to changing environment, (ii) DRL does not rely on any traditional financial risk assumptions, (iii) DRL can incorporate additional data and be a multi inputs method as opposed to more traditional optimization methods.

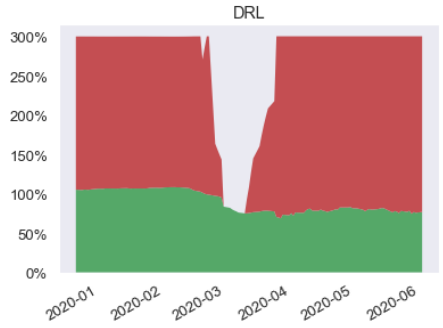


Figure 5: disallocation of DRL model

Future work

As nice as this work is, there is room for improvement as we have only tested a few scenarios and only a limited set of hyper-parameters for our convolutional networks. We should do more intensive testing to confirm that DRL is able to better adapt to changing financial environment. We should also investigate the impact of more layers and other design choice in our network.

Conclusion

In this paper, we discuss how a traditional portfolio allocation problem can be reformulated as a DRL problem, trying to bridge the gaps between the two approaches. We see that the DRL approach enables us to select fewer strategies, improving the overall results as opposed to traditional methods that are built on the concept of diversification. We also stress that DRL can better adapt to changing market conditions and is able to incorporate more information to make decision.

Acknowledgments

We would like to thank Beatrice Guez and Marc Pantic for meaningful remarks. The views contained in this document are those of the authors and do not necessarily reflect the ones of SG CIB.

References

Astrom, K. 1969. Optimal control of markov processes with incomplete state-information ii. the convexity of the loss-function. *Journal of Mathematical Analysis and Applications* 26(2):403–406.

Bao, W., and yang Liu, X. 2019. Multi-agent deep reinforcement learning for liquidation strategy analysis.

Benhamou, E.; Saltiel, D.; Ohana, J.-J.; and Atif, J. 2020a. Detecting and adapting to crisis pattern with context based deep reinforcement learning. *ICPR 2020*.

Benhamou, E.; Saltiel, D.; Ungari, S.; and Mukhopadhyay, A. 2020b. Aamdrl: Augmented asset management with deep reinforcement learning. *arXiv*.

Benhamou, E.; Saltiel, D.; Ungari, S.; and Mukhopadhyay, A. 2020c. Bridging the gap between markowitz planning and deep reinforcement learning. *ICAPS FinPlan workshop*.

Blum, A., and Furst, M. 1995. Fast Planning Through Planning Graph Analysis. In *IJCAI*, 1636–1642.

Chakraborty, S. 2019. Capturing financial markets to apply deep reinforcement learning.

Chopra, V. K., and Ziemba, W. T. 1993. The effect of errors in means, variances, and covariances on optimal portfolio choice. *Journal of Portfolio Management* 19(2):6–11.

Choueifaty, Y., and Coignard, Y. 2008. Toward maximum diversification. *Journal of Portfolio Management* 35(1):40–51.

Choueifaty, Y.; Froidure, T.; and Reynier, J. 2012. Properties of the most diversified portfolio. *Journal of Investment Strategies* 2(2):49–70.

Christoffersen, P.; Errunza, V.; Jacobs, K.; and Jin, X. 2010. Is the potential for international diversification disappearing? Working Paper.

Cogneau, P., and Hübner, G. 2009. The 101 ways to measure portfolio performance. *SSRN Electronic Journal*.

Deng, Y.; Bao, F.; Kong, Y.; Ren, Z.; and Dai, Q. 2016. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems* 28:1–12.

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189.

Fischer, T. G. 2018. Reinforcement learning in financial markets - a survey. *Discussion Papers in Economics 12*.

Gu, S.; Holly, E.; Lillicrap, T.; and Levine, S. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA)*, 3389–3396.

Haugen, R., and Baker, N. 1991. The efficient market inefficiency of capitalization-weighted stock portfolios. *Journal of Portfolio Management* 17:35–40.

Huang, C. Y. 2018. Financial trading as a game: A deep reinforcement learning approach.

Jiang, Z., and Liang, J. 2016. Cryptocurrency Portfolio Management with Deep Reinforcement Learning. *arXiv e-prints*.

Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization.

Kolm, P. N., and Ritter, G. 2019. Modern perspective on reinforcement learning in finance. *SSRN*.

Kritzman, M. 2014. Six practical comments about asset allocation. *Practical Applications* 1(3):6–11.

Levine, S.; Finn, C.; Darrell, T.; and Abbeel, P. 2015. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research* 17.

Levine, S.; Pastor, P.; Krizhevsky, A.; and Quillen, D. 2016. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*.

Li, X.; Li, Y.; Zhan, Y.; and Liu, X.-Y. 2019. Optimistic bull or pessimistic bear: Adaptive deep reinforcement learning for stock portfolio allocation. In *ICML*.

Liang et al. 2018. Adversarial deep reinforcement learning in portfolio management.

Lillicrap, T.; Hunt, J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *CoRR*.

Liu, Y.; Liu, Q.; Zhao, H.; Pan, Z.; and Liu, C. 2020. Adaptive quantitative trading: an imitative deep reinforcement learning approach. In *AAAI*.

Maillard, S.; Roncalli, T.; and Teïletche, J. 2010. The properties of equally weighted risk contribution portfolios. *The Journal of Portfolio Management* 36(4):60–70.

Markowitz, H. 1952. Portfolio selection. *Journal of Finance* 7:77–91.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.; Veness, J.; Bellemare, M.; Graves, A.; Riedmiller, M.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529–33.

Nan, A.; Perumal, A.; and Zaiane, O. R. 2020. Sentiment and knowledge based algorithmic trading with deep reinforcement learning.

Ning, B.; Lin, F. H. T.; and Jaimungal, S. 2018. Double deep q-learning for optimal execution.

Roncalli, T., and Weisang, G. 2016. Risk parity portfolios with risk factors. *Quantitative Finance* 16(3):377–388.

Saltiel, D.; Benhamou, E.; Ohana, J. J.; Laraki, R.; and Atif, J. 2020. Drlps: Deep reinforcement learning for portfolio selection. *ECML PKDD Demo track*.

Schulman, J.; Levine, S.; Moritz, P.; Jordan, M.; and Abbeel, P. 2015a. Trust region policy optimization. In *ICML*.

Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2015b. High-dimensional continuous control using generalized advantage estimation. *ICLR*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *CoRR*.

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–489.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the game of go without human knowledge. *Nature* 550:354–.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Théate, T., and Ernst, D. 2020. Application of deep reinforcement learning in stock trading strategies and stock forecasting.

Vinyals, O.; Babuschkin, I.; Czarnecki, W.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J.; Jaderberg, M.; and Silver, D. 2019. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature* 575.

Wang, H., and Zhou, X. Y. 2019. Continuous-Time Mean-Variance Portfolio Selection: A Reinforcement Learning Framework. *arXiv e-prints*.

Wang, S.; Jia, D.; and Weng, X. 2018. Deep reinforcement learning for autonomous driving. *ArXiv* abs/1811.11329.

Wu, X.; Chen, H.; Wang, J.; Troiano, L.; Loia, V.; and Fujita, H. 2020. Adaptive stock trading strategies with deep reinforcement learning methods. *Information Sciences* 538:142–158.

Xiong, Z.; Liu, X.-Y.; Zhong, S.; Yang, H.; and Walid, A. 2019. Practical deep reinforcement learning approach for stock trading.

Ye, Y.; Pei, H.; Wang, B.; Chen, P.-Y.; Zhu, Y.; Xiao, J.; and Li, B. 2020. Reinforcement-learning based portfolio management with augmented asset movement prediction states. In *AAAI*.

Yu, P.; Lee, J. S.; Kulyatin, I.; Shi, Z.; and Dasgupta, S. 2019. Model-based deep reinforcement learning for financial portfolio optimization. *RWSDM Workshop, ICML 2019*.

Zhang, Z.; Zohren, S.; and Roberts, S. 2019. Deep reinforcement learning for trading.

Zhengyao et al. 2017. Reinforcement learning framework for the financial portfolio management problem. *arXiv*.

Planning From Pixels in Atari With Learned Symbolic Representations

Andrea Dittadi*
Technical University of Denmark
Copenhagen, Denmark
adit@dtu.dk

Frederik K. Drachmann*
Technical University of Denmark
Copenhagen, Denmark
fdrachmann@hotmail.dk

Thomas Bolander
Technical University of Denmark
Copenhagen, Denmark
tobo@dtu.dk

Abstract

Width-based planning methods have been shown to yield state-of-the-art performance in the Atari 2600 video game playing domain using pixel input. One approach consists in an episodic rollout version of the Iterated Width (IW) algorithm called RolloutIW, and uses the B-PROST boolean feature set to represent states. Another approach, π-IW, augments RolloutIW with a learned policy to improve how actions are picked in the rollouts. This policy is implemented as a neural network, and the feature set is derived from an intermediate representation learned by the policy network. Results suggest that learned features can be competitive with hand-crafted ones in the context of width-based search. This paper introduces a new approach, where we leverage variational autoencoders (VAEs) to learn features for the domains in a principled manner, directly from pixels, and without supervision. We use the inference network (or encoder) of the trained VAEs to extract boolean features from screen states, and use them for planning with RolloutIW. The trained model in combination with RolloutIW outperforms the original RolloutIW and human professional play on the Atari 2600 domain and reduces the size of the feature set from 20.5 million to 4,500.

Introduction

Width-based search algorithms have in the last few years become among the state-of-the-art approaches to automated planning, e.g. the original Iterated Width (IW) algorithm (Lipovetzky and Geffner 2012). As in propositional STRIPS planning, states are represented by a set of propositional literals, also called boolean *features*. The state space is searched with breadth-first search (BFS), but the state space explosion problem is handled by pruning states based on their *novelty*. First a parameter k is chosen, called the *width parameter* of the search. Searching with width parameter k essentially means that we only consider k literals/features at a time. A state s generated during the search is called *novel* if there exists a set of k literals/features not made true in any earlier generated state. Unless a state is novel, it is immediately pruned from the search. Clearly, we then reduce the size of the searched state space to be exponential in k. It has been shown that many classical planning problems, e.g. problems from the International Planning Competition (IPC) domains, can be solved efficiently using width-based search with very low values of k.

The essential benefit of using width-based algorithms is the ability to perform semi-structured (based on feature structures) exploration of the state space, and reach deep states that may be important for achieving the planning goals. In classical planning, width-based search has been integrated with heuristic search methods, leading to Best-First Width Search (Lipovetzky and Geffner 2017) that performed well at the 2017 International Planning Competition. Width-based search has also been adapted to reward-driven problems where the algorithm uses a simulator for interacting with the environment (Francès et al. 2017). This has enabled the use of width-based search in reinforcement learning environments such as the the Atari 2600 video game suite, through the Arcade Learning Environment (ALE) (Bellemare et al. 2013). There have been several implementations of width-based search directly using the RAM states of the Atari computer as features (Lipovetzky, Ramirez, and Geffner 2015; Shleyfman, Tuisov, and Domshlak 2016; Jinnai and Fukunaga 2017).

Motivated by the fact that humans do not have access to the internal RAM states of a computer when playing video games, methods for using the algorithms based on the raw screen pixels have been developed. Bandres, Bonet, and Geffner (2018) propose a modified version of IW, called RolloutIW, that uses pixel-based features and achieves results comparable to learning methods in almost real-time on the ALE. The combination of Reinforcement Learning (RL) and RolloutIW in the pixel domain is explored by Junyent, Jonsson, and Gómez (2019), who propose to train a policy to guide the action selection in rollouts.

Regardless of different efforts, width-based algorithms are highly exploratory and, not surprisingly, significantly dependent on the quality of the features defined. The original RolloutIW paper (Bandres, Bonet, and Geffner 2018) uses a set of features called B-PROST (Liang et al. 2015). These are features extracted from the screen pixels by splitting the screen into tiles and keeping track of which colors are present in which tiles. These features have been designed by humans to achieve good performance in ALE. Integrating learning of features efficient for width-based search into the algorithms themselves rather than using hand-crafted fea-

*Equal contribution.

ICAPS 2020 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning.

tures is an interesting challenge—and a challenge that will bring the algorithms more on a par with humans trying to learn to play those video games. In Junyent, Jonsson, and Gómez (2019), the features used were the output features of the last layer of the policy network, improving the performance of IW. With the attempt to learn efficient features for width-based search, and inspired by the results of Junyent, Jonsson, and Gómez (2019), this paper investigates the possibility of a more structured approach to generate features for width-based search using deep generative models.

More specifically, in this paper we use variational autoencoders (VAE)—latent variable models that have been widely used for representation learning (Kingma and Welling 2019) as they allow for approximate inference of the latent variables underlying the data—to learn propositional symbols directly from pixels and without any supervision. We then investigate whether the learned symbolic representation can be successfully used for planning in the Atari 2600 domain.

We compare the planning performance of RolloutIW using our learned representations with RolloutIW using the handcrafted B-PROST features, and report human scores as baseline. Our results show that our learned representations lead to better performance than B-PROST on RolloutIW (and often outperform human players). This is despite the fact that the B-PROST features also contain temporal information (how the screen image dynamically changes), whereas we only extract static features from individual frames. Apart from improving scores in the Atari 2600 domain, we also significantly reduce the number of features (by a factor of more than 10^3). We also investigate in more detail (with ablation studies) which factors seems to lead to good performance on games in the Atari domain.

The main contributions of our paper are hence:

- We train generative models to learn propositional symbols for planning, from pixels and without any supervision.
- We run a large-scale evaluation on Atari 2600 where we compare the performance of RolloutIW with our learned representations to RolloutIW with B-PROST features.
- We investigate with ablation studies the effect of various hyperparameters in both learning and planning.
- We show that our learned representations lead to higher scores than using B-PROST, even though our features don't have any temporal information, and our models are trained from data collected by RolloutIW with B-PROST, limiting the richness of the dataset.

Below, we provide the required background, present the original approach of this paper, discuss our experimental results, and conclude.

Background

In this section, we revise the relevant background on Iterated Width (IW), RolloutIW, planning with pixels, and variational autoencoders (VAEs).

Iterated Width

Iterated Width (IW) (Lipovetzky and Geffner 2012) is a blind-search planning algorithm in which the states are represented by sets of boolean features (sets of propositional atoms). The set of all features/atoms is the *feature set*, denoted F. IW is an algorithm parametrized by a *width parameter* k. We use IW(k) to denote IW with width parameter k. Given an initial state s_0, IW(k) is similar to a standard breadth-first search (BFS) from s_0 except that, when a new state s is generated, it is immediately pruned if it is not novel. A state s generated during search is defined to be *novel* if there is a k-tuple of features (atoms) $t = (f_1, \ldots, f_k) \in F^k$ such that s is the first generated state making all of the f_i true (s making f_i true of course simply means $f_i \in s$, and the intuition is that s then "has" the feature f_i). In particular, in IW(1), the only states that are not pruned are those that contain a feature $f \in F$ for the first time during the search. The maximum number of generated states is exponential in the width parameter (IW(k) generates $O(|F|^k)$ states), whereas in BFS it is exponential in the number of features/atoms.

Rollout IW

RolloutIW(k) (Bandres, Bonet, and Geffner 2018) is a variant of IW(k) that searches via rollouts instead of doing a breadth-first search, hence making it more akin to a depth-first search. A *rollout* is a state-action trajectory $(s_0, a_0, s_1, a_1, \ldots)$ from the initial state s_0 (a path in the state space), where actions a_i are picked at random. It continues until reaching a state that is not novel. A state s is *novel* if it satisfies one of the following: 1) it is distinct from all previously generated states and there exists a k-tuple of features $(f_1, \ldots, f_k)$ that are true in s, but not in any other state of the same depth of the search tree or lower; 2) it is an already earlier generated state and there exists a k-tuple of features $(f_1, \ldots, f_k)$ that are true in s, but not in any other state of lower depth in the search tree. The intuition behind case 1 is that in this case the new state s comes with a combination of k features occurring at a lower depth than earlier encountered, which makes it relevant to explore further. In case 2, s is an existing state already containing the lowest occurrence of one of the combinations of k features, again making it relevant to explore further. When a rollout reaches a state that is not novel, the rollout is terminated, and a new rollout is started from the initial state. The process continues until all possible states are explored, or a given time limit is reached. After the time limit has been reached, an action with maximal expected reward is chosen (the method was developed in the context of the Atari 2600 domain, making use of a simulator to compute action outcomes and rewards). The chosen action is executed, and the algorithm is repeated with the new state as initial state.

RolloutIW is an anytime algorithm that will return an action independently of the time budget. In principle, IW could also be used as an anytime algorithm in the context of the Atari 2600 domain, but it will be severely limited by the breadth-first search strategy that will in practice prevent it from reaching nodes of sufficient depth in the state space, and hence prevent it from discovering rewards that only occur later in the game (Bandres, Bonet, and Geffner 2018).

Planning With Pixels

The Arcade Learning Environment (ALE) (Bellemare et al. 2013) provides an interface to Atari 2600 video games, and has been widely used in recent years as benchmark for reinforcement learning and planning algorithms. In the visual setting, the sensory input consists of a pixel array of size 210×160, where each pixel can have 128 distinct colors. Although in principle the set of $210 \times 160 \times 128$ booleans could be used as features, we will follow Bandres, Bonet, and Geffner (2018) and focus on features that capture meaningful structures from the image.

An example of a visual feature set that has proven successful is *B-PROST* (Liang et al. 2015), which consists of *Basic*, *B-PROS*, and *B-PROT* features. The screen is split into 14×16 disjoint tiles of size 15×10. For each tile (i, j) and color c, the basic feature $f_{i,j,c}$ is 1 iff c is present in (i, j). A B-PROS feature $f_{i,j,c,c'}$ is 1 iff color c is present in tile t, color c' in tile t', and the relative offsets between the tiles are i and j. Similarly, a B-PROT feature $f_{i,j,c,c'}$ is 1 iff color c is present in tile t *in the previous decision point*, color c' is present in tile t' in the current one, and the relative offsets between the tiles are i and j. The number of features in B-PROST is the sum of the number of features in these 3 sets, in total 20,598,848.

Variational Autoencoders

Latent variable models. *Latent variable models* (LVMs) are a type of probabilistic models in which some variables are unobserved. For one datapoint, the marginal distribution over the observed variables $\mathbf{x}$ is:

$$p_\theta(\mathbf{x}) = \int_{\mathbf{z}} p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} \tag{1}$$

where $\mathbf{z}$ are the unobserved *latent variables* and θ denotes the model parameters. This quantity is typically referred to as *marginal likelihood* or *model evidence*. A simple and rather common structure for LVMs is:

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x} \,|\, \mathbf{z}) p_\theta(\mathbf{z}) \tag{2}$$

where both $p_\theta(\mathbf{x} \,|\, \mathbf{z})$ and $p_\theta(\mathbf{z})$ are specified.

If the model's distributions are parameterized by neural networks, the marginal likelihood is typically intractable for lack of an analytical solution or a practical estimator. Since $p_\theta(\mathbf{z} \,|\, \mathbf{x}) = p_\theta(\mathbf{x}, \mathbf{z}) / p_\theta(\mathbf{x})$ and $p_\theta(\mathbf{x}, \mathbf{z})$ is tractable to compute, it follows that the intractability of $p_\theta(\mathbf{x})$ derives in fact from that of the posterior $p_\theta(\mathbf{z} \,|\, \mathbf{x})$.

Amortized variational inference. *Variational inference* (VI) is a common approach to approximating this intractable posterior, where we define a distribution $q_\phi(\mathbf{z} \,|\, \mathbf{x})$ with *variational parameters* ϕ, and optimize it to be "close" to the exact posterior. In the LVM above, for any choice of q_ϕ:

$$\log p_\theta(\mathbf{x}) = \log \mathbb{E}_{q_\phi(\mathbf{z} \,|\, \mathbf{x})} \left[\frac{p_\theta(\mathbf{x} \,|\, \mathbf{z}) p_\theta(\mathbf{z})}{q_\phi(\mathbf{z} \,|\, \mathbf{x})} \right] \tag{3}$$

$$\geq \mathbb{E}_{q_\phi(\mathbf{z} \,|\, \mathbf{x})} \left[\log \frac{p_\theta(\mathbf{x} \,|\, \mathbf{z}) p_\theta(\mathbf{z})}{q_\phi(\mathbf{z} \,|\, \mathbf{x})} \right] \tag{4}$$

$$= \mathcal{L}_{\theta,\phi}(\mathbf{x}) \tag{5}$$

where $\mathcal{L}_{\theta,\phi}(\mathbf{x})$ is a lower bound on the marginal log likelihood also known as *Evidence Lower BOund* (ELBO).

In contrast to traditional VI methods, where per-datapoint variational parameters are optimized separately, *amortized variational inference* utilizes function approximators like neural networks to share variational parameters across datapoints and improve learning efficiency. In this setting, $q_\phi(\mathbf{z} \,|\, \mathbf{x})$ is typically called *inference model* or *encoder*.

Variational Autoencoders (VAEs) (Kingma and Welling 2013; Rezende, Mohamed, and Wierstra 2014) are a framework for amortized VI, in which the ELBO is maximized by jointly optimizing the inference model and the LVM (i.e., ϕ and θ, respectively) with stochastic gradient ascent.

VAE optimization. The ELBO, the objective function to be maximized, can be decomposed as follows:

$$\begin{aligned} \mathcal{L}_{\theta,\phi}(\mathbf{x}) &= \mathbb{E}_{q_\phi} \left[\log p_\theta(\mathbf{x} \,|\, \mathbf{z}) \right] - \mathbb{E}_{q_\phi} \left[\log \frac{q_\phi(\mathbf{z} \,|\, \mathbf{x})}{p_\theta(\mathbf{z})} \right] \\ &= \mathbb{E}_{q_\phi} \left[\log p_\theta(\mathbf{x} \,|\, \mathbf{z}) \right] - D_{\mathrm{KL}}(q_\phi(\mathbf{z} \,|\, \mathbf{x}) \,||\, p_\theta(\mathbf{z})) \end{aligned} \tag{6}$$

where the first term can be interpreted as negative expected *reconstruction error*, and the second term is the KL divergence from the prior $p_\theta(\mathbf{z})$ to the approximate posterior. Minimizing the reconstruction error (i.e., maximizing the first term) pushes 1) the decoder to accurately reconstruct the input (in expectation over the encodings of $\mathbf{x}$), and 2) the inference model to encode $\mathbf{x}$ in such a way that the decoder can do so more effectively.

One of the major challenges in the optimization of VAEs is that the gradients of some terms of the objective function cannot be backpropagated through the sampling step. However, for a rather wide class of probability distributions, a random variable following such distribution can be expressed as a differentiable, deterministic transformation of an auxiliary variable with independent marginal distribution. For example, if z is a sample from a Gaussian random variable with mean $\mu_\phi(x)$ and standard deviation $\sigma_\phi(x)$, then $z = \sigma_\phi(x)\, \epsilon + \mu_\phi(x)$, where $\epsilon \sim \mathcal{N}(0, 1)$. Thanks to this *reparameterization*, z can be differentiated with respect to ϕ by standard backpropagation. This approach, called *pathwise gradient estimator*, typically exhibits lower variance than the alternatives, and is widely used in practice.

From an information theory perspective, optimizing the variational lower bound (6) involves a tradeoff between rate and distortion (Alemi et al. 2017) or, equivalently, between how much the data is compressed (more compression corresponds to a lower KL divergence) and how much information we retain (more information corresponds to a lower reconstruction loss). A straightforward way to control the rate–distortion tradeoff is to use the β-VAE framework (Higgins et al. 2017), in which the training objective (6) is modified by scaling the KL term:

$$\begin{aligned} \mathcal{L}_{\theta,\phi,\beta}(\mathbf{x}) = \mathbb{E}_{q_\phi} \left[\log p_\theta(\mathbf{x} \,|\, \mathbf{z}) \right] \\ - \beta D_{\mathrm{KL}}(q_\phi(\mathbf{z} \,|\, \mathbf{x}) \,||\, p_\theta(\mathbf{z})) \end{aligned} \tag{7}$$

VAEs with discrete variables. Since the categorical distribution is not reparameterizable, training VAEs with categorical latent variables is generally impractical. A solution

to this problem is to replace samples from a categorical distribution with samples from a Gumbel-Softmax distribution (Jang, Gu, and Poole 2016; Maddison, Mnih, and Teh 2016), which can be smoothly annealed into a categorical distribution by making the temperature parameter τ tend to 0. Because Gumbel-Softmax is reparameterizable, the pathwise gradient estimator can be used to get a low-variance—although in this case biased—estimate of the gradient. In this work, we use Bernoulli latent variables (categorical with 2 classes) and their Gumbel-Softmax relaxations.

VAE-IW

Although width-based planning has been shown to be generally very effective for classical planning domains (Lipovetzky and Geffner 2012; 2017; Francès et al. 2017), its performance in practice significantly depends on the quality of the features used. Features that better capture meaningful structure in the environment typically translate to better planning results (Junyent, Jonsson, and Gómez 2019). Here we propose VAE-IW, in which representations extracted by a VAE are used as features for RolloutIW. The main advantages are the following:

- The number of features can be orders of magnitude smaller than B-PROST, leading to faster planning and a smaller memory footprint.
- Autoencoders, in particular VAEs, are a natural approach for learning meaningful, compact representations from data (Bengio, Courville, and Vincent 2013; Tschannen, Bachem, and Lucic 2018; Kingma and Welling 2019).
- No additional preprocessing is needed, such as background masking (Junyent, Jonsson, and Gómez 2019).

On the planning side, we follow Junyent, Jonsson, and Gómez (2019) and use RolloutIW(k) with width $k = 1$ to keep planning efficient even with a small time budget. For example, with a budget of 0.5s per planning phase RolloutIW(2) generates only a few nodes per step, whereas RolloutIW(1) generates hundreds. Width 1 is sufficient to get good results, and this choice makes it easier to compare our experimental results with the aforementioned paper. In the remainder of this section, we detail the two main components of the proposed method: feature learning and planning.

Unsupervised Feature Learning

For feature learning we use a variational autoencoder with a discrete latent space:

$$p_\theta(\mathbf{x}) = \sum_{\mathbf{z}} p_\theta(\mathbf{x}, \mathbf{z}) = \sum_{\mathbf{z}} p_\theta(\mathbf{x} \,|\, \mathbf{z}) p(\mathbf{z}) \tag{8}$$

where the prior is a product of independent Bernoulli distributions with a fixed parameter:

$$p(\mathbf{z}) = \prod_{i=1}^{K} p(z_i) = \prod_{i=1}^{K} \text{Bernoulli}(\mu)\ . \tag{9}$$

Given an image $\mathbf{x}$, we would like to retrieve the binary latent factors $\mathbf{z}$ that generated it, and use them as propositional representation for planning. Since the likelihood function $p_\theta(\mathbf{x} \,|\, \mathbf{z})$ is parameterized by a neural network, and therefore highly nonlinear with respect to the latent variables, inference of the latent variables is intractable (it would require the computation of the sum in (8), which has a number of terms exponential in the number of latent variables).

We define an inference model:

$$q_\phi(\mathbf{z} \,|\, \mathbf{x}) = \prod_{i=1}^{K} q_\phi(z_i \,|\, \mathbf{x}) = \prod_{i=1}^{K} \text{Bernoulli}((\mu_\phi(\mathbf{x}))_i)$$

to approximate the true posterior $p_\theta(\mathbf{z} \,|\, \mathbf{x})$. The *encoder* μ_ϕ is a deep neural network with parameters ϕ that outputs the approximate posterior probabilities of all latent variables given an image $\mathbf{x}$. Using stochastic amortized variational inference, we train the inference and generative models end-to-end by maximizing the ELBO with respect to the parameters of both models. We approximate the discrete Bernoulli distribution of $\mathbf{z}$ with the Gumbel-Softmax relaxation (Jang, Gu, and Poole 2016; Maddison, Mnih, and Teh 2016), which is reparameterizable. This allows us to estimate the gradients of the inference network with the pathwise gradient estimator. Alternatively, the approximate posterior could be optimized directly using a score-function estimator (Williams 1992) with appropriate variance reduction techniques or alternative variational objectives (Bornschein and Bengio 2014; Mnih and Rezende 2016; Le et al. 2018; Masrani, Le, and Wood 2019; Liévin et al. 2020).

Note that we are not necessarily interested in the best generative model—e.g. in terms of marginal log likelihood of the data or visual quality of the generated samples—as much as in representations that are useful for planning. In order to control the tradeoff between the competing terms in (6), we use the β-VAE framework, and following Burgess et al. (2018) we decrease β until good quality reconstructions are obtained.

Planning With Learned Features

After training, the inference model $q_\phi(\mathbf{z} \,|\, \mathbf{x})$ yields approximate posterior distributions of the latent variables. The binary features for downstream planning can be obtained by sampling from the approximate posteriors or by directly thresholding their probabilities:

$$f_i = \begin{cases} 1 & \text{if } \mu_\phi(\mathbf{x})_i \geq \lambda \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where $\lambda \in (0, 1)$ is the threshold. In this work we choose to deterministically threshold the probabilities, as it empirically yields more stable features for planning. Overall, this approach provides a way of efficiently computing a compact, binary representation of an image, which can in turn be interpreted as a set of propositional features to be used in symbolic reasoning systems like IW or RolloutIW.

The planning phase in VAE-IW is based on RolloutIW, but uses the features extracted by the inference network instead of the B-PROST features. Note that while the B-PROST features include temporal information (specifically in the B-PROT subset), the VAE features are computed independently for each frame. This should, everything else being equal, give an advantage to planners that rely on B-PROST features.

Experiments

We evaluate the proposed approach on a suite of 47 Atari 2600 games. We separately trained a VAE for each domain by maximizing (7) using stochastic gradient ascent. As reconstruction loss we used the binary cross-entropy. We used the Adam optimizer (Kingma and Ba 2014) with default parameters and learning rate 10^{-4}. To train the VAEs, we collected 15,000 frames by running RolloutIW(1) with B-PROST features, and split them into a training and validation set of 14,250 and 750 images, respectively. The RGB images of size 210×160 are grayscaled and downsampled to 128×128. VAE-IW experiments were performed with 4GB of RAM on one core of a Xeon Gold 6126 CPU, and a Tesla V100 GPU. The GPU was used both to train the models and to evaluate the encoder in the planning phase.

The encoder consists of 3 convolutional layers interleaved with residual blocks. The single convolutional layers downsample the feature map by a factor of 2, and each residual block includes 2 convolutional layers, resulting in 7 convolutional layers in total. The decoder mirrors such architecture, and performs upsampling with transposed convolutional layers. Further architectural details are provided in the Appendix. The inference network outputs a feature map of shape $H \times W \times C$ which represents the approximate posterior probabilities of the latent variables. Thus, unlike in traditional VAEs where latent variables do not carry any spatial meaning, in our case they are spatially arranged (Vahdat and Kautz 2020). As outlined in the previous section, the Bernoulli probabilities computed by the encoder are thresholded, and the resulting binary features are used as propositional representations for planning in RolloutIW(1), instead of the B-PROST features. For the planning phase of VAE-IW, we use RolloutIW(1) with partial caching and risk aversion (RA) as described by Bandres, Bonet, and Geffner (2018). With partial caching, after each iteration of RolloutIW the branch of the chosen action is kept in memory. Risk aversion is attained by multiplying all negative rewards by a factor $\alpha \gg 1$ when they are propagated up the tree.

Because of their diversity, Atari games vary widely in complexity. We empirically chose a set of hyperparameters that performed reasonably well on a small subset of the games, assuming this would generalize well enough to other games. Following previous work (Lipovetzky and Geffner 2012; Bandres, Bonet, and Geffner 2018), we used a frame skip of 15 and a planning budget of 0.5s per time step. We set an additional limit of 15,000 executed actions for each run, to prevent runs from lasting too long. Note that this constraint is only applied to our method. Table 1 summarizes the main hyperparameters used in our experiments. We expect that better results can be obtained by performing an extensive hyperparameter search on the whole suite of Atari 2600 games.

Parameter	Value
Batch size	64
Learning rate	10^{-4}
Latent space size	$15 \times 15 \times 20 = 4500$
τ	0.5
β	10^{-4}
μ	0.5
α	50000
γ	0.99
λ	0.9
Frame skip	15
Planning budget	0.5s

Table 1: Hyperparameters in VAE-IW experiments unless specified otherwise.

Main Results

In Table 2, we compare the planning performance of our RA VAE-IW(1) to IW(1) with B-PROST features (Lipovetzky and Geffner 2012) and RA RolloutIW(1) (Bandres, Bonet, and Geffner 2018) with B-PROST features. In Figure 1, we normalize the scores of width-based planning methods using scores from random and human play, as in Bandres, Bonet, and Geffner (2018). Note, however, that human play is only meant as a baseline: since humans do not have access to a simulator, a direct comparison cannot be made. Following the literature, we report the average score over 5 runs, with each run ending when the game is over. These results show that learning binary features purely from images, without any supervision, can be beneficial for width-based planning. In particular, using the learned representations in RolloutIW leads to generally higher scores in the Atari 2600 domain.

Note that the performance of VAE-IW depends on the quality and expressiveness of the features extracted by the VAE, which in turn depend on the images the VAE is trained on. Crucially, since we collect data by running RolloutIW(1) with B-PROST features, the performance of VAE-IW is constrained by the effectiveness of the data collection algorithm. The VAE features will not necessarily be meaningful on parts of the game that are significantly different from those in the training set. Surprisingly, however, our method significantly outperforms the baseline that was used for data collection, providing even stronger evidence that the compact set of features learned by a VAE can be successfully utilized for width-based planning algorithms. This form of bootstrapping can be iterated: images collected by VAE-IW could be used for re-training or fine-tuning VAEs, potentially leading to further performance improvements. Although in principle the first iteration of VAEs could be trained from images collected via random play, this is not a viable solution in hard-exploration problems such as many Atari games (Ecoffet et al. 2019).

In addition, there appears to be a significant negative correlation between the average number of true features and the average number of expanded nodes per planning phase (Spearman rank correlation of -0.51, p-value < 0.001). In other words, in domains where the VAE extracts on average more true features, the planning algorithm tends to expand fewer nodes. Thus, it could be potentially fruitful to further investigate the interplay between the average number of true features, the meaningfulness and usefulness of the features, and the efficiency of width-based planning algorithm.

Algorithm Features Planning budget	Human	IW B-PROST 0.5s	RA Rollout IW B-PROST 0.5s	RA VAE-IW VAE $15 \times 15 \times 20$ 0.5s
Alien	6,875.0	1,316.0	7,170.0	**8,144.0**
Amidar	1,676.0	48.0	1,049.2	**1,551.0**
Assault	1,496.0	268.8	336.0	**1,250.0**
Asterix	8,503.0	1,350.0	46,100.0	**999,500.0***
Asteroids	13,157.0	840.0	4,698.0	**14,816.0**
Atlantis	29,028.0	33,160.0	122,220.0	**1,955,280.0***
Battle zone	37,800.0	6,800.0	74,600.0	**191,000.0**
Beam rider	5,775.0	715.2	2,552.8	**3,432.0**
Berzerk		280.0	**1,208.0**	828.0
Bowling	154.0	30.6	44.2	**63.0**
Breakout	31.8	1.6	**86.2**	50.8
Centipede	11,963.0	88,890.0	56,328.0	**113,369.2***
Crazy climber	35,411.0	16,780.0	40,440.0	**941,660.0***
Demon attack	3,401.0	106.0	6,958.0	**276,586.0***
Double dunk	-15.5	-22.0	3.2	**10.4**
Elevator action		1,080.0	0.0	**38,940.0***
Fishing derby	5.5	-83.8	-77.0	**-19.6**
Freeway	29.6	0.6	2.0	**4.8**
Frostbite	4,335.0	106.0	146.0	**254.0**
Gopher	2,321.0	1,036.0	**8,388.0**	7,968.0
Gravitar	2,672.0	380.0	1,660.0	**2,360.0**
Ice hockey	0.9	-13.6	-12.4	**37.6**
James bond 007	406.7	40.0	**10,760.0**	5,280.0
Krull	2,395.0	3,206.8	2,091.8	**3,455.2**
Kung-fu master	22,736.0	440.0	2,620.0	**5,300.0**
Ms. Pac-man	15,693.0	2,578.0	15,115.0	**16,200.6**
Name this game	4,076.0	7,070.0	6,558.0	**18,526.0**
Phoenix		1,266.0	**6,790.0**	6,160.0
Pitfall!		-8.6	-302.8	**-5.8**
Pong	9.3	-20.8	-4.2	**10.4**
Private eye	69,571.0	**2,690.8**	-480.0	60.0
Q*bert	13,455.0	515.0	**15,970.0**	2,070.0
River raid	13,513.0	664.0	6,288.0	**6,818.0**
Road Runner	7,845.0	200.0	**31,140.0**	2,740.0
Robotank	11.9	3.2	**31.2**	30.8
Seaquest	20,182.0	168.0	**2,312.0**	560.0
Skiing		-16,511.0	-16,006.8	**-10,443.8**
Space invaders	1,652.0	280.0	1,149.0	**2,943.0**
Stargunner	10,250.0	840.0	**14,900.0**	1,040.0
Tennis	-8.9	-23.4	-5.4	**-0.6**
Time pilot	5,925.0	2,360.0	3,540.0	**32,440.0**
Tutankham	167.7	71.2	135.6	**180.6**
Up'n down	9,082.0	928.0	34,668.0	**764,264.0***
Video pinball	17,298.0	28,706.4	**216,468.6**	149,284.6
Wizard of wor	4,757.0	5,660.0	43,860.0	**199,900.0**
Yars' revenge		6,352.6	7,848.8	**105,637.0**
Zaxxon	9,173.0	0.0	**15,500.0**	12,120.0
# > Human	n/a	4	18	24
# > 75% Human	n/a	4	21	27
# best	n/a	1	12	34

Table 2: Score comparison of width-based methods in 47 Atari games in the pixel setting. Scores in bold indicate the best overall width-based method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. In the bottom rows we also report the number of domains in which an algorithm's score was greater than the human score, or greater than 75% of the human score. Results for IW and Rollout IW B-PROST are from Bandres, Bonet, and Geffner (2018); human scores are from Liang et al. (2015).

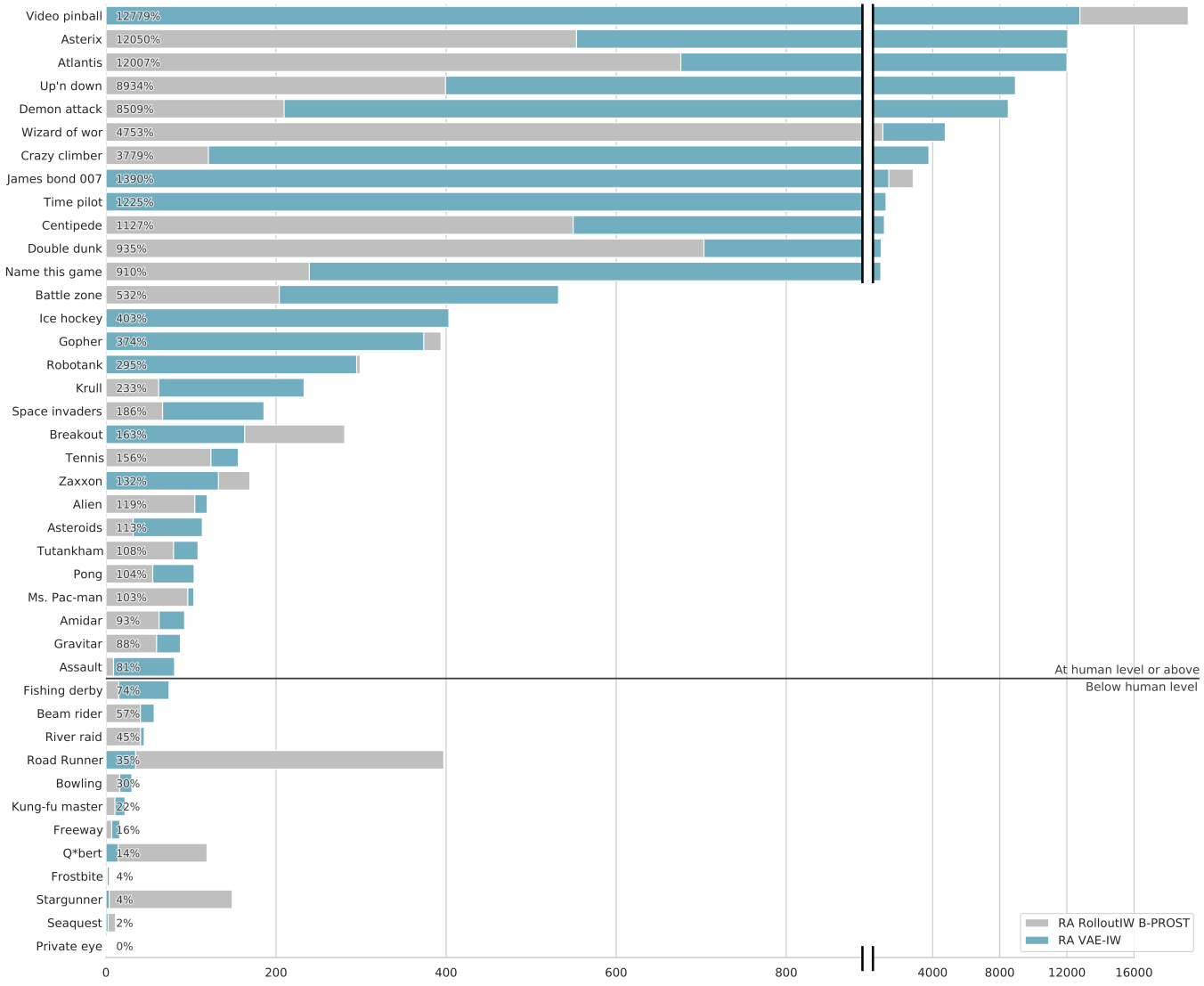


Figure 1: Comparison of the risk-averse variants of VAE-IW (ours, using VAE features) and RolloutIW B-PROST (using B-PROST features). Following Mnih et al. (2015), the performance of both methods is normalized with respect to a professional human game tester (100% level) and random play (0%) as: $100 \times (\text{VAE} - \text{random play})/(\text{human score} - \text{random play})$. RA VAE-IW obtains the highest score among width-based approaches in most games, and it performs at a level that is superior to or comparable with professional human play. The reported percentages are for VAE features.

Ablation Studies

The performance of VAE-IW depends on several hyperparameters related to the planning algorithm and to the probabilistic model. Here we attempt to investigate the effect of some of these parameters.

Modeling choices. One of the major modeling choices is the dimensionality of the latent space, and the spatial structure $H \times W \times C$ of the latent variables. Both of these factors are tightly coupled with the neural architecture underlying the inference and generative networks. As there is no clear heuristic, we explored different neural architectures and latent space sizes. Based on the performance on a few selected domains, we chose two different settings, with latent space size $15 \times 15 \times 20$ and $4 \times 4 \times 200$ (see the Appendix for further details). In Table 12 we compare the performance of RA VAE-IW on these two configurations, keeping the rest of the hyperparameters fixed as the ones used in Table 2. While overall the $15 \times 15 \times 20$ configuration leads to a higher score in most domains, the effect of this modeling choice seems to significantly depend on the domain.

As previously mentioned, we consider the framework of β-VAEs in which β controls the trade-off between reconstruction accuracy and amount of information encoded in

the latent space. For our purposes, β has to be small enough that the model can capture all relevant details in an image. In practice, we decreased β until the model generated sufficiently accurate reconstructions on a selection of Atari games (Burgess et al. 2018). Table 13 reports the performance of VAE-IW when varying $\beta \in \{10^{-4}, 10^{-3}\}$, and shows that the effect of varying β depends on the domain. Intuitively, while a stronger regularization (i.e. higher β) can be detrimental for the reconstructions and thus also for the informativeness of the learned features, it may lead to better representations in less visually challenging domains. In practice, one could for example train VAEs with different regularization strengths, and do unsupervised model selection separately for each domain by using the "elbow method" (Ketchen and Shook 1996) on the reconstruction loss or other relevant metrics.

Planning parameters. Regardless of the features used for planning, the performance of VAE-IW depends on the variant of RolloutIW being used, and on its parameters. In Table 8 we compare the average score of VAE-IW with and without risk aversion, and observe that the risk-averse variant achieves an equal or higher average score in 33 of the 47 domains.

Table 9 shows the results of VAE-IW and RolloutIW with B-PROST features, similarly to Table 2, except that both methods are run *without* risk aversion. With this modification, our method still obtains a higher average score in the majority of domains (32/47).

Another crucial planning parameter is the time budget for planning at each time step. While the main results are based on a 0.5s budget, we also consider a 32s budget, following Bandres, Bonet, and Geffner (2018). In Table 10 we observe that, not surprisingly, the high time budget outperforms the low budget in most domains (34/47). However, in some of them the shorter planning budget yields a significantly higher score (e.g. in Asterix, CrazyClimber, and ElevatorAction). Interestingly, increasing the planning budget seems to leave the average rollout depth unaffected, while the average number of expanded nodes in each planning phase grows significantly. This behaviour is consistently observed in all tested domains (see Figures 2 and 3) and points to the fact that increasing the planning budget improves results mostly by allowing more rollouts.

In Table 11 we compare the average scores obtained by VAE-IW and RolloutIW with B-PROST features, using a 32s planning budget for both methods. Once again, using the compact features learned by a VAE seems to be beneficial, as VAE-IW obtains the highest average score in 29 of the 47 domains.

Related Work

Variational Autoencoders (VAEs) have been extensively used for representation learning (Kingma and Welling 2019) as their amortized inference network lends itself naturally to this task. In the context of automated planning, Asai and Fukunaga (2017) proposed the State Autoencoder (SAE) for propositional symbolic grounding. An SAE is in fact a VAE with Bernoulli latent variables. It is trained by maximizing a modified ELBO that includes an additional entropy regularizer, defined as twice the negative KL divergence. Thus, the objective function being maximized is the ELBO (6) with the sign of the KL flipped. Although unintentional (Asai and Kajino 2019), this proved to be fundamental for the mitigation of the issue of representation instability. A variation of SAE, the zero-suppressed state autoencoder (Asai and Kajino 2019), adds a further regularization term to the propositional representation (features), leading to more stable representations (Asai and Kajino 2019).

Zhang et al. (2018) take a supervised approach to representation learning for planning, and learn a transition graph for planning in the representation space with Dijkstra's algorithm. Konidaris, Kaelbling, and Lozano-Perez (2018) specify a set of skills for a task, and then automatically extract state representations from raw observations. Kurutach et al. (2018) use generative adversarial networks to learn structured representations of images and a deterministic dynamics model, and plan with graph-search methods.

Junyent, Jonsson, and Gómez (2019) proposed π-IW, a variant of RolloutIW(1) where a neural network guides the action selection process in the rollouts, which would otherwise be random. This is reminiscent of AlphaZero (Silver et al. 2018), where a policy network guides the rollouts of Monte Carlo Tree Search (MCTS). Moreover, π-IW plans using features obtained from the last hidden layer of the policy network, instead of B-PROST.

Conclusion

We have introduced a novel combination of width-based planning with learning techniques. The learning employs a VAE to learn relevant features in video games from the Atari 2600 suite, given raw images of screen states as training data. The planning is done with RolloutIW(1) using the features learned by the VAE. Our approach reduces the size of the feature set from the 20.5 million B-PROST features used in previous work in connection with RolloutIW, to only 4,500. Our algorithm, VAE-IW, outperforms the previous methods, proving that VAEs can learn meaningful representations that can be effectively used for width-based planning.

In VAE-IW, the symbolic representations are learned from data collected by RolloutIW using B-PROST features. Increasing the diversity and quality of the training data could potentially lead to better representations, and thus better planning results. One possible way to achieve this could be to iteratively retrain or fine-tune the VAEs on data collected by the current iteration of VAE-IW: The planner would produce new images to retrain the VAE, which could again be used in combination with RolloutIW, resulting in a new generation of VAE-IW. The quality of the representations could also be improved by using more expressive discrete models, for example with a hierarchy of discrete latent variables (Van Den Oord, Vinyals, and Kavukcuoglu 2017; Razavi, van den Oord, and Vinyals 2019). Finally, we can expect further improvements orthogonal to this work, by learning a rollout policy for more effective action selection, as investigated by Junyent, Jonsson, and Gómez (2019).

References

Alemi, A. A.; Poole, B.; Fischer, I.; Dillon, J. V.; Saurous, R. A.; and Murphy, K. 2017. Fixing a broken elbo. *arXiv preprint arXiv:1711.00464*.

Asai, M., and Fukunaga, A. 2017. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *arXiv preprint arXiv:1705.00154*.

Asai, M., and Kajino, H. 2019. Towards stable symbol grounding with zero-suppressed state autoencoder. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 592–600.

Bandres, W.; Bonet, B.; and Geffner, H. 2018. Planning with pixels in (almost) real time. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47:253–279.

Bengio, Y.; Courville, A.; and Vincent, P. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35(8):1798–1828.

Bornschein, J., and Bengio, Y. 2014. Reweighted wake-sleep. *arXiv preprint arXiv:1406.2751*.

Burgess, C. P.; Higgins, I.; Pal, A.; Matthey, L.; Watters, N.; Desjardins, G.; and Lerchner, A. 2018. Understanding disentangling in $beta$-vae. *arXiv preprint arXiv:1804.03599*.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2019. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*.

Francès, G.; Ramírez Jávega, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *IJCAI 2017. Twenty-Sixth International Joint Conference on Artificial Intelligence*.

Higgins, I.; Matthey, L.; Pal, A.; Burgess, C.; Glorot, X.; Botvinick, M.; Mohamed, S.; and Lerchner, A. 2017. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR 2017*.

Jang, E.; Gu, S.; and Poole, B. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Jinnai, Y., and Fukunaga, A. 2017. Learning to prune dominated action sequences in online black-box planning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Junyent, M.; Jonsson, A.; and Gómez, V. 2019. Deep policies for width-based planning in pixel domains. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 646–654.

Ketchen, D. J., and Shook, C. L. 1996. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal* 17(6):441–458.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P., and Welling, M. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Kingma, D. P., and Welling, M. 2019. An introduction to variational autoencoders. *arXiv preprint arXiv:1906.02691*.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research* 61:215–289.

Kurutach, T.; Tamar, A.; Yang, G.; Russell, S. J.; and Abbeel, P. 2018. Learning plannable representations with causal infogan. In *Advances in Neural Information Processing Systems*, 8733–8744.

Le, T. A.; Kosiorek, A. R.; Siddharth, N.; Teh, Y. W.; and Wood, F. 2018. Revisiting reweighted wake-sleep. *arXiv preprint arXiv:1805.10469*.

Liang, Y.; Machado, M. C.; Talvitie, E.; and Bowling, M. 2015. State of the art control of atari games using shallow reinforcement learning. *arXiv preprint arXiv:1512.01563*.

Liévin, V.; Dittadi, A.; Christensen, A.; and Winther, O. 2020. Optimal variance control of the score function gradient estimator for importance weighted bounds. *arXiv preprint arXiv:2008.01998*.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*.

Lipovetzky, N., and Geffner, H. 2017. Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, 3590–3596.

Lipovetzky, N.; Ramirez, M.; and Geffner, H. 2015. Classical planning with simulators: Results on the atari video games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Maddison, C. J.; Mnih, A.; and Teh, Y. W. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

Masrani, V.; Le, T. A.; and Wood, F. 2019. The thermodynamic variational objective. In *Advances in Neural Information Processing Systems*, 11521–11530.

Mnih, A., and Rezende, D. J. 2016. Variational inference for monte carlo objectives. *arXiv preprint arXiv:1602.06725*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Razavi, A.; van den Oord, A.; and Vinyals, O. 2019. Generating diverse high-fidelity images with VQ-VAE-2. In *Advances in Neural Information Processing Systems*, 14866–14876.

Rezende, D. J.; Mohamed, S.; and Wierstra, D. 2014. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*.

Shleyfman, A.; Tuisov, A.; and Domshlak, C. 2016. Blind

search for atari-like online planning revisited. In *IJCAI*, 3251–3257.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Tschannen, M.; Bachem, O.; and Lucic, M. 2018. Recent advances in autoencoder-based representation learning. *arXiv preprint arXiv:1812.05069*.

Vahdat, A., and Kautz, J. 2020. Nvae: A deep hierarchical variational autoencoder. *arXiv preprint arXiv:2007.03898*.

Van Den Oord, A.; Vinyals, O.; and Kavukcuoglu, K. 2017. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, 6306–6315.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3-4):229–256.

Zhang, A.; Sukhbaatar, S.; Lerer, A.; Szlam, A.; and Fergus, R. 2018. Composable planning with attributes. In *International Conference on Machine Learning*, 5842–5851.

Appendix

BatchNorm
LeakyReLU(0.01)
Conv 3×3, padding=1
Dropout(0.2)
BatchNorm
LeakyReLU(0.01)
Conv 3×3, padding=1
Dropout(0.2)
Residual connection
LeakyReLU(0.01)

Table 3: Residual block. The number of channels is always 64. The residual connection consists of summing the activation to the block's input.

Conv 3×3, stride=2
Residual block
Conv 3×3, stride=2
Residual block
Conv 3×3, stride=2
Residual block
Conv 3×3, stride=2
Residual block
Conv 3×3, stride=2, padding=1

Table 4: Encoder with output of spatial size 4×4. The number of channels is always 64 except for the last convolution, in which the number of output channels controls the latent space size.

ConvT 3×3, stride=2
Residual block
ConvT 4×4, stride=2
Residual block
ConvT 4×4, stride=2
Crop 128×128
Sigmoid activation

Table 5: Decoder with input of spatial size 15×15. The number of channels is always 64 except for the last convolution, in which the number of output channels is 1. ConvT denotes transposed convolutional layers.

Conv 4×4, stride=2
Residual block
Conv 4×4, stride=2
Residual block
Conv 3×3, stride=2, padding=1

Table 6: Encoder with output of spatial size 15×15. The number of channels is always 64 except for the last convolution, in which the number of output channels controls the latent space size.

ConvT 3×3, stride=2
Residual block
ConvT 3×3, stride=2
Residual block
ConvT 3×3, stride=2
Residual block
ConvT 3×3, stride=2
Residual block
ConvT 3×3, stride=2
Crop 128×128
Sigmoid activation

Table 7: Decoder with input of spatial size 4×4. The number of channels is always 64 except for the last convolution, in which the number of output channels is 1. ConvT denotes transposed convolutional layers.

Algorithm	VAE-IW	RA VAE-IW
β	10^{-4}	10^{-4}
Planning horizon	0.5s	0.5s
Features	VAE $15 \times 15 \times 20$	VAE $15 \times 15 \times 20$
Alien	5,576.0	**8,144.0**
Amidar	1,093.2	**1,551.0**
Assault	826.4	**1,250.0**
Asterix	**999,500.0***	**999,500.0***
Asteroids	772.0	**14,816.0**
Atlantis	40,620.0	**1,955,280.0***
Battle zone	91,200.0	**191,000.0**
Beam rider	3,179.6	**3,432.0**
Berzerk	566.0	**828.0**
Bowling	**64.6**	63.0
Breakout	13.0	**50.8**
Centipede	22,020.2	**113,369.2**
Crazy climber	661,460.0	**941,660.0***
Demon attack	9,656.0	**276,586.0***
Double dunk	7.2	**10.4**
Elevator action	**76,160.0***	38,940.0*
Fishing derby	-20.2	**-19.6**
Freeway	**6.2**	4.8
Frostbite	248.0	**254.0**
Gopher	6,084.0	**7,968.0**
Gravitar	**2,770.0**	2,360.0
Ice hockey	36.0	**37.6**
James bond 007	640.0	**5,280.0**
Krull	**3,543.2**	3,455.2
Kung-fu master	5,160.0	**5,300.0**
Ms. Pac-man	**20,161.0**	16,200.6
Name this game	13,332.0	**18,526.0**
Phoenix	5,328.0	**6,160.0**
Pitfall!	**0.0**	-5.8
Pong	9.2	**10.4**
Private eye	**139.0**	60.0
Q*bert	1,890.0	**2,070.0**
River raid	4,884.0	**6,818.0**
Road Runner	**4,720.0**	2,740.0
Robotank	21.4	**30.8**
Seaquest	**596.0**	560.0
Skiing	**-9,664.8**	-10,443.8
Space invaders	1,962.0	**2,943.0**
Stargunner	**1,120.0**	1,040.0
Tennis	**5.2**	-0.6
Time pilot	24,840.0	**32,440.0**
Tutankham	167.4	**180.6**
Up'n down	35,964.0	**764,264.0***
Video pinball	**462,619.4**	149,284.6
Wizard of wor	89,380.0	**199,900.0***
Yars' revenge	85,800.6	**105,637.0**
Zaxxon	7,320.0	**12,120.0**
# best	15	33

Table 8: Score comparison of VAE-IW with and without risk aversion. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

Algorithm	Rollout IW	VAE-IW
β		10^{-4}
Planning horizon	0.5s	0.5s
Features	B-PROST	VAE $15 \times 15 \times 20$
Alien	4,238.0	**5,576.0**
Amidar	659.8	**1,093.2**
Assault	285.0	**826.4**
Asterix	45,780.0	**999,500.0**
Asteroids	**4,344.0**	772.0
Atlantis	**64,200.0**	40,620.0
Battle zone	39,600.0	**91,200.0**
Beam rider	2,188.0	**3,179.6**
Berzerk	**644.0**	566.0
Bowling	47.6	**64.6**
Breakout	**82.4**	13.0
Centipede	**36,980.2**	22,020.2
Crazy climber	39,220.0	**661,460.0***
Demon attack	2,780.0	**9,656.0**
Double dunk	3.6	**7.2**
Elevator action	0.0	**76,160.0***
Fishing derby	-68.0	**-20.2**
Freeway	2.8	**6.2**
Frostbite	220.0	**248.0**
Gopher	**7,216.0**	6,084.0
Gravitar	1,630.0	**2,770.0**
Ice hockey	-6.0	**36.0**
James bond 007	450.0	**640.0**
Krull	1,892.8	**3,543.2**
Kung-fu master	2,080.0	**5,160.0**
Ms. Pac-man	9,178.4	**20,161.0**
Name this game	6,226.0	**13,332.0**
Phoenix	**5,750.0**	5,328.0
Pitfall!	-81.4	**0.0**
Pong	-7.4	**9.2**
Private eye	-322.0	**139.0**
Q*bert	**3,375.0**	1,890.0
River raid	**6,088.0**	4,884.0
Road Runner	2,360.0	**4,720.0**
Robotank	**31.0**	21.4
Seaquest	**980.0**	596.0
Skiing	-15,738.8	**-9,664.8**
Space invaders	**2,628.0**	1,962.0
Stargunner	**13,360.0**	1,120.0
Tennis	-18.6	**5.2**
Time pilot	7,640.0	**24,840.0**
Tutankham	128.4	**167.4**
Up'n down	**36,236.0**	35,964.0
Video pinball	203,765.4	**462,619.4**
Wizard of wor	37,220.0	**89,380.0**
Yars' revenge	5,225.4	**85,800.6**
Zaxxon	**9,280.0**	7,320.0
# best	15	32

Table 9: Score comparison between RolloutIW B-PROST and VAE-IW. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

Algorithm	VAE-IW	VAE-IW
β	10^{-4}	10^{-4}
Planning horizon	0.5s	32s
Features	VAE $15 \times 15 \times 20$	VAE $15 \times 15 \times 20$
Alien	5,576.0	**8,536.0**
Amidar	1,093.2	**1,955.8**
Assault	826.4	**1,338.4**
Asterix	**999,500.0**	417,100.0
Asteroids	772.0	**1,158.0**
Atlantis	40,620.0	**49,500.0**
BattleZone	91,200.0	**234,200.0**
BeamRider	3,179.6	**5,580.0**
Berzerk	**566.0**	554.0
Bowling	**64.6**	46.8
Breakout	13.0	**72.4**
Centipede	22,020.2	**166,244.8**
CrazyClimber	**661,460.0**	129,840.0
DemonAttack	**9,656.0**	5,397.0
DoubleDunk	**7.2**	5.2
ElevatorAction	**76,160.0**	0.0
FishingDerby	-20.2	**20.6**
Freeway	6.2	**29.4**
Frostbite	248.0	**280.0**
Gopher	6,084.0	**17,604.0**
Gravitar	**2,770.0**	2,640.0
IceHockey	36.0	**44.2**
Jamesbond	640.0	**650.0**
Krull	3,543.2	**6,664.0**
KungFuMaster	5,160.0	**20,960.0**
MsPacman	20,161.0	**25,759.0**
NameThisGame	13,332.0	**15,276.0**
Phoenix	5,328.0	**5,960.0**
Pitfall	**0.0**	-0.4
Pong	9.2	**12.0**
PrivateEye	139.0	**157.8**
Qbert	1,890.0	**4,760.0**
Riverraid	4,884.0	**5,372.0**
RoadRunner	4,720.0	**8,540.0**
Robotank	21.4	**24.2**
Seaquest	**596.0**	324.0
Skiing	**-9,664.8**	-9,705.0
SpaceInvaders	1,962.0	**2,972.0**
StarGunner	1,120.0	**1,180.0**
Tennis	5.2	**12.6**
TimePilot	**24,840.0**	24,220.0
Tutankham	167.4	**197.2**
UpNDown	35,964.0	**91,592.0**
VideoPinball	462,619.4	**833,518.4**
WizardOfWor	**89,380.0**	76,460.0
YarsRevenge	85,800.6	**188,551.2**
Zaxxon	7,320.0	**30,200.0**
# best	13	34

Table 10: Score comparison of VAE-IW with planning budget of 0.5 or 32 seconds. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

Algorithm	RolloutIW	VAE-IW
β		10^{-4}
Planning horizon	32s	32s
Features	B-PROST	VAE $15 \times 15 \times 20$
Alien	6,896.0	**8,536.0**
Amidar	1,698.6	**1,955.8**
Assault	319.2	**1,338.4**
Asterix	66,100.0	**417,100.0**
Asteroids	**7,258.0**	1,158.0
Atlantis	**151,120.0**	49,500.0
BattleZone	**414,000.0**	234,200.0
BeamRider	2,464.8	**5,580.0**
Berzerk	**862.0**	554.0
Bowling	45.8	**46.8**
Breakout	36.0	**72.4**
Centipede	65,162.6	**166,244.8**
CrazyClimber	43,960.0	**129,840.0**
DemonAttack	**9,996.0**	5,397.0
DoubleDunk	**20.0**	5.2
ElevatorAction	**0.0**	**0.0**
FishingDerby	-16.2	**20.6**
Freeway	12.6	**29.4**
Frostbite	**5,484.0**	280.0
Gopher	13,176.0	**17,604.0**
Gravitar	**3,700.0**	2,640.0
IceHockey	6.6	**44.2**
Jamesbond	**22,250.0**	650.0
Krull	1,151.2	**6,664.0**
KungFuMaster	14,920.0	**20,960.0**
MsPacman	19,667.0	**25,759.0**
NameThisGame	5,980.0	**15,276.0**
Phoenix	**7,636.0**	5,960.0
Pitfall	-130.8	**-0.4**
Pong	**17.6**	12.0
PrivateEye	**3,157.2**	157.8
Qbert	**8,390.0**	4,760.0
Riverraid	**8,156.0**	5,372.0
RoadRunner	**37,080.0**	8,540.0
Robotank	**52.6**	24.2
Seaquest	**10,932.0**	324.0
Skiing	-16,477.0	**-9,705.0**
SpaceInvaders	1,980.0	**2,972.0**
StarGunner	**15,640.0**	1,180.0
Tennis	-2.2	**12.6**
TimePilot	8,140.0	**24,220.0**
Tutankham	184.0	**197.2**
UpNDown	44,306.0	**91,592.0**
VideoPinball	382,294.8	**833,518.4**
WizardOfWor	73,820.0	**76,460.0**
YarsRevenge	9,866.4	**188,551.2**
Zaxxon	22,880.0	**30,200.0**
# best	19	29

Table 11: Score comparison between RolloutIW B-PROST and VAE-IW with planning budget of 32 seconds. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

Algorithm	RA VAE-IW	RA VAE-IW
β	10^{-4}	10^{-4}
Planning horizon	0.5s	0.5s
Features	VAE $15 \times 15 \times 20$	VAE $4 \times 4 \times 200$
Alien	**8,144.0**	7,592.0
Amidar	**1,551.0**	1,526.6
Assault	1,250.0	**1,308.6**
Asterix	**999,500.0***	**999,500.0***
Asteroids	**14,816.0**	8,852.0
Atlantis	**1,955,280.0***	1,912,700.0*
Battle zone	191,000.0	**228,000.0**
Beam rider	3,432.0	**3,450.0**
Berzerk	**828.0**	752.0
Bowling	**63.0**	47.6
Breakout	**50.8**	28.4
Centipede	113,369.2	**253,823.6**
Crazy climber	**941,660.0***	930,420.0*
Demon attack	276,586.0*	**292,099.0***
Double dunk	**10.4**	6.0
Elevator action	38,940.0*	**109,680.0***
Fishing derby	**-19.6**	-31.2
Freeway	**4.8**	2.0
Frostbite	**254.0**	244.0
Gopher	7,968.0	**8,504.0**
Gravitar	**2,360.0**	1,560.0
Ice hockey	**37.6**	37.2
James bond 007	5,280.0	**11,000.0**
Krull	**3,455.2**	3,219.0
Kung-fu master	**5,300.0**	3,600.0
Ms. Pac-man	**16,200.6**	15,066.8
Name this game	**18,526.0**	13,670.0
Phoenix	6,160.0	**9,210.0**
Pitfall!	**-5.8**	-9.6
Pong	**10.4**	3.6
Private eye	60.0	**115.4**
Q*bert	2,070.0	**4,935.0**
River raid	**6,818.0**	6,790.0
Road Runner	**2,740.0**	2,320.0
Robotank	30.8	**45.2**
Seaquest	560.0	**1,084.0**
Skiing	**-10,443.8**	-11,906.4
Space invaders	**2,943.0**	2,753.0
Stargunner	1,040.0	**1,200.0**
Tennis	**-0.6**	-6.6
Time pilot	**32,440.0**	23,460.0
Tutankham	**180.6**	158.4
Up'n down	**764,264.0***	627,706.0*
Video pinball	149,284.6	**248,101.2**
Wizard of wor	**199,900.0***	111,580.0
Yars' revenge	**105,637.0**	97,004.6
Zaxxon	12,120.0	**17,360.0**
# best	31	17

Table 12: Score comparison of VAE-IW with latent space size $15 \times 15 \times 20$ and $4 \times 4 \times 200$. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

Algorithm	RA VAE-IW	RA VAE-IW
β	10^{-3}	10^{-4}
Planning horizon	0.5s	0.5s
Features	VAE $15 \times 15 \times 20$	VAE $15 \times 15 \times 20$
Alien	4,902.0	**8,144.0**
Amidar	1,186.2	**1,551.0**
Assault	**1,468.2**	1,250.0
Asterix	**999,500.0***	**999,500.0***
Asteroids	7,204.0	**14,816.0**
Atlantis	**1,978,540.0***	1,955,280.0*
Battle zone	**406,600.0**	191,000.0
Beam rider	**4,377.6**	3,432.0
Berzerk	774.0	**828.0**
Bowling	35.0	**63.0**
Breakout	**53.4**	50.8
Centipede	**296,791.4**	113,369.2
Crazy climber	**976,580.0***	941,660.0*
Demon attack	**301,886.0**	276,586.0
Double dunk	7.4	**10.4**
Elevator action	**68,920.0**	38,940.0
Fishing derby	**-15.6**	-19.6
Freeway	4.0	**4.8**
Frostbite	**262.0**	254.0
Gopher	5,420.0	**7,968.0**
Gravitar	2,300.0	**2,360.0**
Ice hockey	34.0	**37.6**
James bond 007	630.0	**5,280.0**
Krull	**3,486.4**	3,455.2
Kung-fu master	4,960.0	**5,300.0**
Ms. Pac-man	**17,483.0**	16,200.6
Name this game	15,120.0	**18,526.0**
Phoenix	5,524.0	**6,160.0**
Pitfall!	-6.8	**-5.8**
Pong	-2.2	**10.4**
Private eye	40.0	**60.0**
Q*bert	**8,040.0**	2,070.0
River raid	6,078.0	**6,818.0**
Road Runner	2,080.0	**2,740.0**
Robotank	**44.6**	30.8
Seaquest	316.0	**560.0**
Skiing	-11,027.6	**-10,443.8**
Space invaders	2,721.0	**2,943.0**
Stargunner	**1,100.0**	1,040.0
Tennis	-16.6	**-0.6**
Time pilot	30,920.0	**32,440.0**
Tutankham	165.8	**180.6**
Up'n down	682,080.0*	**764,264.0***
Video pinball	**445,085.8**	149,284.6
Wizard of wor	184,260.0	**199,900.0***
Yars' revenge	77,950.8	**105,637.0**
Zaxxon	10,520.0	**12,120.0**
# best	18	30

Table 13: Score comparison of RA VAE-IW with different values of β. Scores in bold indicate the best method, and scores with a star indicate that the algorithm reached the limit on executed actions at least once. Ties are counted as won for both methods.

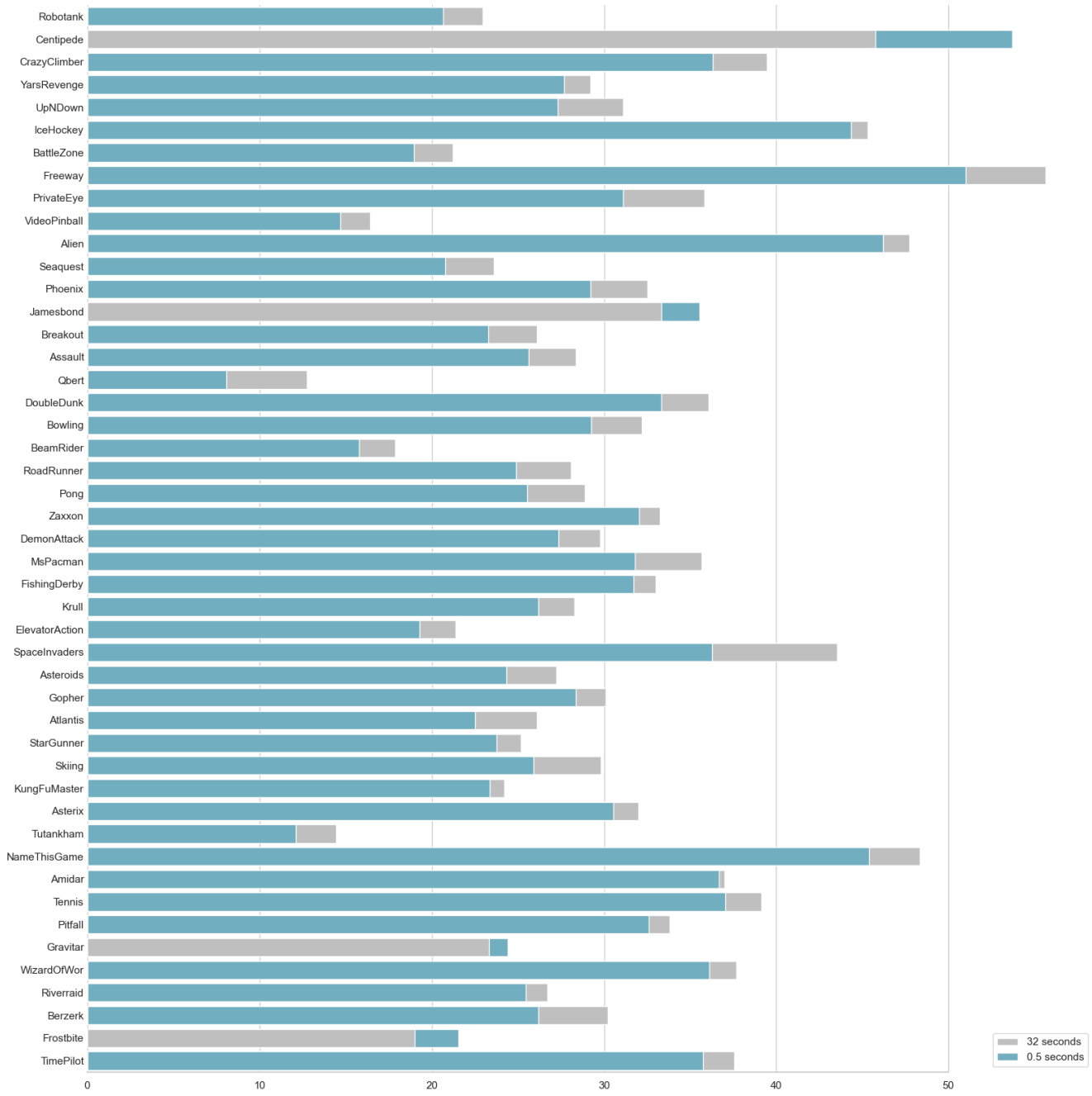


Figure 2: We calculate the mean number of expanded nodes of after each planning phase for each domain. The data is collected with one run for each domain. The comparison is between the 0.5 second RA VAE-IW and 32 second RA VAE-IW.

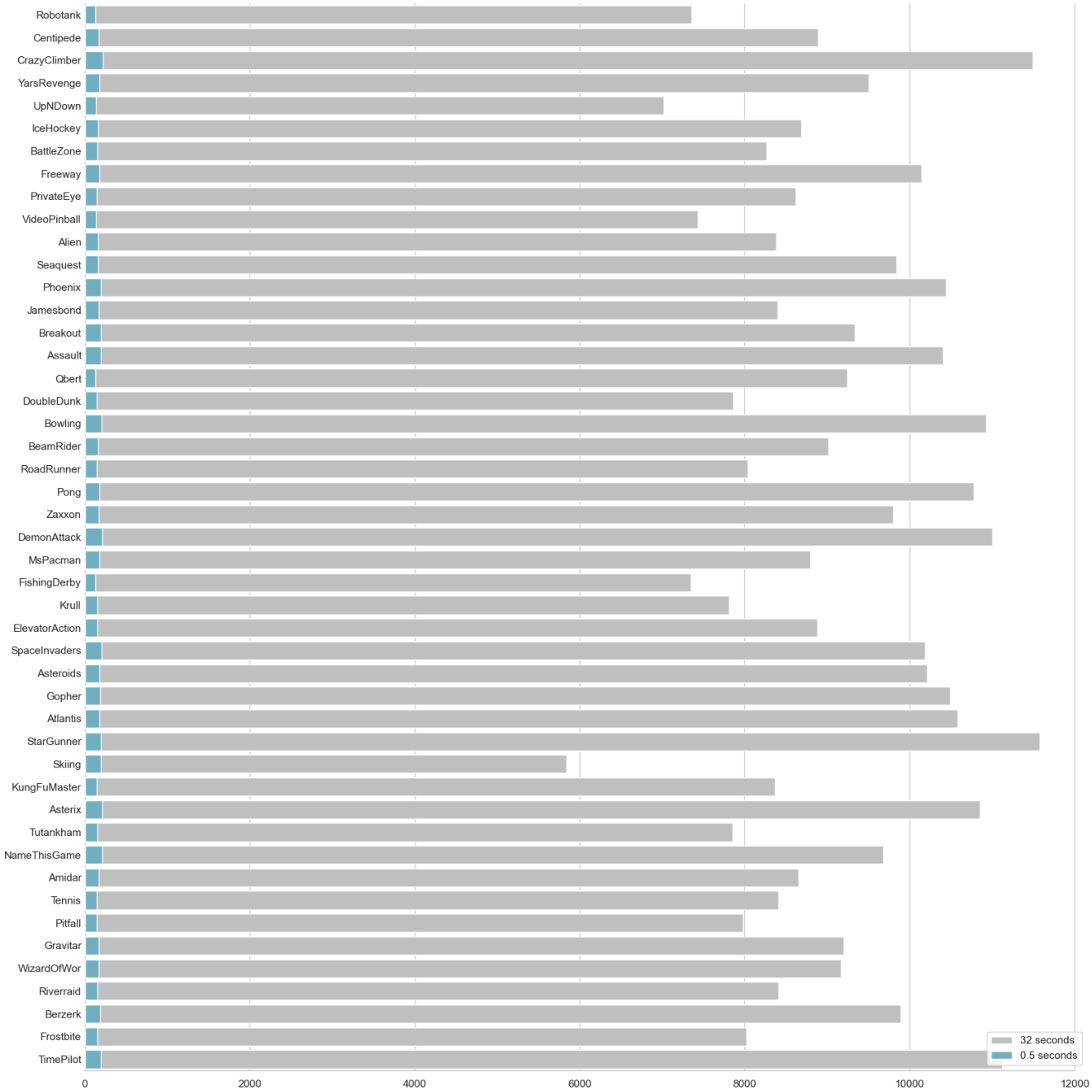


Figure 3: We calculate the mean of the max depth nodes in the planning tree after each planning phase for each domain. The data is collected with one run for each domain. The comparison is between the 0.5 second RA VAE-IW and 32 second RA VAE-IW.

Offline Learning for Planning: A Summary

Giorgio Angelotti,[1,2] **Nicolas Drougard,**[2] **Caroline P. C. Chanel**[2]
[1]ANITI - Artificial and Natural Intelligence Toulouse Institute, Université de Toulouse,
41 Allées Jules Guesde, 31013 Toulouse - CEDEX 6, France
[2]ISAE-SUPAERO, Université de Toulouse,
10 Avenue Edouard Belin, 31055 Toulouse - CEDEX 4, France
{name.surname}@isae-supaero.fr

Abstract

The training of autonomous agents often requires expensive and unsafe trial-and-error interactions with the environment. Nowadays several data sets containing recorded experiences of intelligent agents performing various tasks, spanning from the control of unmanned vehicles to human-robot interaction and medical applications are accessible on the internet. With the intention of limiting the costs of the learning procedure it is convenient to exploit the information that is already available rather than collecting new data. Nevertheless, the incapability to augment the batch can lead the autonomous agents to develop far from optimal behaviours when the sampled experiences do not allow for a good estimate of the true distribution of the environment. Offline learning is the area of machine learning concerned with efficiently obtaining an optimal policy with a batch of previously collected experiences without further interaction with the environment. In this paper we adumbrate the ideas motivating the development of the state-of-the-art offline learning baselines. The listed methods consist in the introduction of epistemic uncertainty dependent constraints during the classical resolution of a Markov Decision Process, with and without function approximators, that aims to alleviate the bad effects of the distributional mismatch between the available samples and real world. We provide comments on the practical utility of the theoretical bounds that justify the application of these algorithms and suggest the utilization of Generative Adversarial Networks to estimate the distributional shift that affects all of the proposed model-free and model-based approaches.

Learning using a single batch of collected experiences is a statistical challenge of crucial importance for the development of intelligent agents, specially in scenarios where the interaction with the environment can be expensive, risky or unpractical. There are countless examples that fall in these categories: the training of unmanned aerial vehicles (Baek et al. 2013), self-driving cars (Mirchevska et al. 2018), medical applications (Jonsson 2018), Human-Robot interaction (Chanel et al. 2020). Several environments are so complex that a direct formulation of a model based on mere intuition is inappropriate and unsafe because, depending on the task, any mistake made by the agent can lead to catastrophic aftermaths. It is therefore necessary to infer the world dynamics from a batch of previously collected experiences. The said data set should be *large* and *diverse* enough for allowing useful information extraction.

The process of learning an optimal policy can be mathematically formalized as the resolution of a Markov Decision Process (MDP) if the state of the system can be considered as fully observable and the action effects are non necessarily deterministic. This paper addresses the problems linked to the resolution of MDPs starting from a single batch of collected experiences by writing up a summary of the state-of-the-art methods on offline learning and planning, and outlining their pros and cons. When the data set is fixed the distributional shift between the true, unknown, underlying MDP and its best data-driven estimate can be non negligible and lead, on resolution, to bad performing policies. This discrepancy can be seen tightly linked to the uncertainty we possess about the model. Several offline learning baselines try to handle this issue or by constraining the policy or by reshaping the reward taking into account a local quantification of the said epistemic uncertainty and hence adapting the classical resolution paradigms (Levine et al. 2020).
The document will be structured as follows:

1. Firstly, a recap of MDP resolution with MDP planning algorithms, but also with reinforcement learning (RL) algorithms is proposed.
2. Then, an intuitive description of offline model learning and batch RL is presented.
3. And finally, a discussion is provided with comments on the theoretical guarantees for the performance of the listed baselines and suggestions for further improvements in this field, like resorting to Generative Adversarial Networks (GANs) to better estimate the underlying distributions.

1 A Review of MDPs

An MDP is formally defined as a tuple $M \stackrel{\text{def}}{=} (S, A, T, r, \gamma, \mu_0)$ where S is the set of states, A the set of actions, $T : A \times S \times S \to [0, 1]$ is the state transition function defining the probability that dictates the evolution from $s \in S$ to $s' \in S$ after taking the action $a \in A$, $R : A \times S \to [R_{min}, R_{max}]$ with $R_{max}, R_{min} \in \mathrm{R}$ and

$R_{max} > R_{min}$ is the reward function that indicates what the agent gains when it selects action $a \in A$ and the system state is $s \in S$, $\gamma \in [0, 1)$ is called the discount factor and $\mu_0 : S \to [0, 1]$ is the initial probability distribution over states $s \in S$ at time $t = 0$. A policy is defined as a function that maps states to actions, such as $\pi : A \times S \to [0, 1]$; $\pi(a|s)$ can be interpreted as the probability of taking action $a \in A$ when being in the state $s \in S$. Time evolution is discrete and at every time step the agent observes the system, acts on the environment and earns a reward. The following definitions refer to an MDP with discrete A and S but they can be straightforwardly rearranged to address MDP with continuous states and actions spaces.

Solving an MDP amounts to finding a policy π^* which, $\forall s \in S$, maximizes the value function:

$$V_M^\pi(s) \stackrel{\text{def}}{=} \mathrm{E}_{\substack{a_t \sim \pi(\cdot|s_t) \\ s_{t+1} \sim T(\cdot|s_t, a_t)}} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s \right].$$

The value function can also be expressed recursively as the fixed point of the Bellman operator:

$$V_M^\pi(s) = \sum_{a \in A} \pi(a|s) \left(R(s, a) + \gamma \sum_{s' \in S} T(s'|s, a) V_M^\pi(s') \right).$$

We define also the Q-value function:

$$Q^\pi(s, a) \stackrel{\text{def}}{=} R(s, a) + \gamma \sum_{s' \in S} T\left(s' \mid s, a\right) V_M^\pi(s')$$

and we notice that $V_M^\pi(s) = \mathrm{E}_{a \sim \pi(\cdot|s)} \left[Q^\pi(s, a) \right]$.

Resolution Schemes

With basic planning algorithms like Value or Policy Iteration where the contraction property of the Bellman operator is exploited, one can compute a value V and a policy π that iteratively converge to V^* and π^*, respectively (Sutton and Barto 1998; Mausam and Kolobov 2012). These algorithms require to store in memory the whole state space. However, the application of the Bellman operator demands that all the functions that compose the MDP are known. What can we do, for instance, if the transition function is unknown?

Model-free Approaches In such scenarios, temporal difference (TD) schemes like *Q-learning* (Watkins and Dayan 1992) can be applied. In Q-learning the Q-function is computed iteratively by minimizing the TD error using sampled transitions (s, a, r, s'). Q-learning is a model free RL algorithm because 1) it does not require an a-priori knowledge of the model, 2) it exploits a growing batch of sampled experiences. Another popular model free approach is based on Policy Gradients (Williams 1992) which maximizes an estimate of the value function with respect to the policy where the expected value over the transition distribution is replaced by sampled transitions.

Policy Gradients methods serve as the base for the Actor-Critic architecture by which the variance of the gradient with respect to the policy is reduced either by replacing the cumulative reward with an estimate of the Q-value or by subtracting from it an estimate of the value function (Sutton et al. 1999). The module that compute Q-value and value function estimates is called the Critic and the one that computes π thanks to the Policy Gradient method is named the Actor.

Model-based Approaches Another route to follow is that of first using a batch of previously sampled experiences to obtain both $\hat{T}$ and $\hat{R}$ which are respectively estimates of the transition function T and of the reward function R of the unknown MDP. And then, to directly execute a planning algorithm or to use $\hat{T}$ as a generative model of new fictitious experiences (s, a, r, s') and subsequently apply a model-free technique using the new data set augmented with the artificial transitions. Such a scheme was first mentioned in the Dyna-Q algorithm (Sutton and Barto 1998), even though it was prescribed to be used in combination with periodical further explorations of the environment. Regarding the MDP planning literature, techniques which exploit heuristic guided trial-based solving have been created to address large finite state spaces. Amongst all *Upper Confidence bounds applied to Trees* (UCT) and more recently, PROST (Kocsis and Szepesvári 2006; Keller and Eyerich 2012). Such algorithms could be applied to the estimated MDP generative model, using $\hat{T}$ and $\hat{R}$.

Notice that when new data is generated the optimal policy of the MDP defined by $\hat{T}$ can be different from the one of the original MDP since they really define two different decision processes. The authors of the work (van Hasselt, Hessel, and Aslanides 2019) questioned the advantage of using a model to generate fictitious data over working directly on the batch with model free algorithms. Model-based techniques are normally more data efficient than model free competitors since probably the model learning stage can capture more easily the characteristics needed to estimate the Q-value and value function. However, in that paper it is empirically displayed that an appropriately fine-tuned model-free algorithm can achieve a superior data efficiency performance.

Function Approximators

When the state or the action space has the cardinality of the continuum a tabular representation of policies and value functions is unfeasible. If the states are characterized by continuous feature vectors, planning algorithms are not applicable without a preliminary discretization. (Munos and Moore 2002) proposes a variable resolution discretization of S assuming that a perfect generative model is available. The latter enables to split recursively the feature space where more control is required preventing an unforgivable loss of resolution in the transition function of the aggregate MDP. Even though the variable resolution scheme provides a more efficient splitting criterion than an uniform grid, it does not manage to escape from the curse of dimensionality. Conversely, some promising attempts have been fulfilled in the case of a continuous action space and finite state space (Mansley, Weinstein, and Littman 2011). Resorting to the Universal Approximation Theorem (Csáji 2001) it has been found practical to use function approximators in order to estimate the policy and the value functions. The increase in computational power of the last decade gave birth to a rich

community of scientists and engineers who use function approximators with thousands of parameters such as neural networks. Model-free algorithms using neural networks are Deep Q-learning Networks (Mnih et al. 2015), Policy Gradients (Williams 1992; Sutton et al. 1999) and their subsequent improvements (Hessel et al. 2018) (Schulman et al. 2015; Mnih et al. 2016; Barth-Maron et al. 2018; Haarnoja et al. 2018) that led to development of agents which achieved better than human performances in games like Go (Silver et al. 2016), Chess (Silver et al. 2017), and also some video games of the ATARI suite.

In model-based settings, approximators usually need the specification of a prior distribution for T which is often chosen Gaussian since these algorithms are usually applied to problems driven by a deterministic dynamics or to problems whose intrinsic stochasticity can be thought being induced by a Gaussian distribution in some latent space (Deisenroth and Rasmussen 2011; Chua et al. 2018; Hafner et al. 2019; Kaiser et al. 2020; Hafner et al. 2020). The latter is a strong limitation of these approaches since, more often than not, taking decisions under uncertainty amounts to deal with multi-modal transition distributions that would be poorly described by a normal distribution.

2 Single Batch Learning

As we have stated in the introduction, learning from a single batch of collected experiences is a necessity of compelling importance for a safe, cost limited and data efficient development of intelligent agents. We will see that several algorithms which constrain the optimal policy obtained with RL or planning tools to one that does not drive the agent to regions of $S \times A$ that have been poorly sampled in the data set lead to more effective policies than the one used to collect the batch. Usually the results are also better than the one obtained with a policy that has been calculated by straightforwardly applying the schemes listed in Section 1.

The utilization of function approximators to estimate the value functions using a single batch requires theoretical delicacy since many convergence guarantees do not stand. In the paper (Chen and Jiang 2019) the authors realized that usually two fundamental assumptions are implicitly required in order for the following algorithms to work:

1. mild shifts between the distributions of the real world and the one inferred from the data in the batch,
2. conditions on the class of candidate value-functions stronger than just the membership of the optimal Q-value to this function class.

Related to those points, (Chen and Jiang 2019) explores the notion of *concentratability* coefficient (Munos 2003), which is hereafter recalled.

$\forall (s,a) \in S \times A$ and $\forall h \geq 0, \forall \pi$:

$$\frac{P\left(s_h = s, a_h = a | s_0 \sim \mu_0, \pi\right)}{\mu_{\mathcal{B}}(s,a)} \leq C,$$

where $\mu_{\mathcal{B}}$ is the probability distribution that generated the batch assuming that the transitions are independent and identically distributed. The existence of C ensures that any attainable distribution of state-action pairs is not too far away from $\mu_{\mathcal{B}}$. The main result reported in (Chen and Jiang 2019) is that not constraining C precludes sampled efficient learning even with "the most favourable" data distribution $\mu_{\mathcal{B}}$.

Rather than focusing on the practical implementation of the different methodologies, which as we will see is often approximate due to the intractability of the terms present in the derived theoretical bounds, we aim to perform a simple yet comprehensible adumbration of the ideas that support their development. With this in mind we are going to neglect implementation related technicalities and sketch the theoretical foundations of single batch learning algorithms.

Constraints for Model-free Algorithms

The first successful applications of offline learning for planning and control with function approximators are very recent (Fujimoto, Meger, and Precup 2019; Fujimoto et al. 2019). In these works the authors showed that performing Q-learning to solve a finite state MDP using a fixed batch $\mathcal{B}$ leads to the optimal policy $\pi^*_{\mathcal{B}}$ for the MDP $M_{\mathcal{B}}$ whose transition function is the most likely one with respect to the transitions $(s,a,r,s') \in \mathcal{B}$. More often than not, the optimal policy for $M_{\mathcal{B}}$ performs poorly in the true environment. The discrepancy between the transition function of the original process and the one learnt from the batch will be the key element of the following discussion.

Indeed, the *extrapolation error* for a given policy π:

$$\epsilon(s,a) \stackrel{\text{def}}{=} Q^{\pi}(s,a) - Q^{\pi}_{\mathcal{B}}(s,a)$$

defined as the difference between the Q-value function of the real MDP and the Q-value function of the most likely MDP learnt from the batch could be computed with an operator similar to the Bellman's one:

$$\begin{aligned} \epsilon(s,a) \quad = \quad & \gamma \sum_{s' \in S} \Big[\big(T(s'|s,a) - T_{\mathcal{B}}(s'|s,a) \big) V^{\pi}_{\mathcal{B}}(s') \\ & + T(s'|s,a) \sum_{a' \in A} \pi(a'|s') \epsilon(s',a') \Big] \end{aligned}$$

The authors noticed that the extrapolation error is a function of divergence between the true transition distribution and the one estimated from the batch along with the error at succeeding states. Their idea is then to minimize the error by constraining the policy to visit regions of $S \times A$ where the transition distributions are similar. Henceforth, they modified Q-Learning and Deep Q-Learning algorithms to force the new "optimal" policy to be *not so distant* from the one that was used during the collection of the batch. They train a generator network that gets as an input a state s to estimate the batch generating policy and then allow for a small perturbation around it. The magnitude of the perturbation is an hyperparameter. In this way, they obtain a policy that always achieves better performance than the one used during the batch collection. This algorithm is called Batch Constrained Q-Learning (BCQ).

In a subsequent work, it has been shown that the error in the estimation of the Q-value with neural networks is generated by the back-up of poor estimates of the Q-value that

comes from regions $S \times A$ that were badly sampled in $\mathcal{B}$ (Kumar et al. 2019). To contrast the accumulation of the error, the authors developed the Bootstrapping Error Accumulation Reduction (BEAR) algorithm which, exploiting the notion of distribution concentratability, manages to constrain the improved policy to the support of the one that generated the batch. Strictly speaking, they blame the back-up of Q-value estimates of states with Out Of Distribution (OOD) actions for increasing the extrapolation error. They should blame for OOD state-action transitions, but in offline Q-learning the Q function is computed only at states that are in the replay buffer. This constraint is softer than the one imposed by BCQ and it has been showed to provide better results.

When BEAR and BCQ are applied on batches generated with a random policy, they can eventually perform worse than Deep Q-learning naively applied using the batch as a fixed replay buffer. In these cases, if the data set is big enough, there are not many OOD actions (Kumar et al. 2019). Probably, enforcing a constraint as done in BEAR and BCQ will provide a too little window for policy improvement.

In the same year, yet another inspiring paper about offline reinforcement learning was published (Wu, Tucker, and Nachum 2019). The authors of the latter showed that any policy constraining approach like BCQ, BEAR and KL-Control (Jaques et al. 2019) can be obtained as a special case of their Behaviour Regularized Actor Critic (BRAC) algorithm.

The general idea is to either 1) penalize the value function estimated by the actor or 2) regularize the policy generated by the critic by a distance in probability space between the batch collector policy $\pi_{\mathcal{B}}$ and the currently evaluated one, as:

$$V_D^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t \mathrm{E}_{\substack{a_t \sim \pi(\cdot|s_t) \\ s_{t+1} \sim T(\cdot|s_t, a_t)}} \Big[r(s_t, a_t) - \alpha D\big(\pi(\cdot|s_t), \pi_{\mathcal{B}}(\cdot|s_t)\big) | s = s_0 \Big]$$

where D is a distance function in probability space (e.g. Kernel MMD, Kullback-Leibler, Total Variation, Wasserstein, etc) and α is an hyperparameter. While the policy regularized learning objective of the actor maximizes the following criterion:

$$\mathop{\mathrm{E}}_{(s,a,r,s') \sim \mathcal{B}} \left[\mathop{\mathrm{E}}_{a \sim \pi(\cdot|s)} [Q(s,a)] - \alpha D\big(\pi(\cdot|s), \pi_{\mathcal{B}}(\cdot|s)\big) \right]$$

Their results showed that overall value penalization works better than policy regularization and the distance D that provides the best performing policy is the Kullback-Leibler divergence.

In their practical implementation both BCQ and BEAR use an average Q-value over an ensemble of Q-value networks to reduce the prediction error. In BRAC, the minimum Q-value over an ensemble of Q-value networks is used.

Extrapolation Error Reduction with Random Ensembles of Q-value Networks

In parallel to the previous studies, the authors of (Agarwal, Schuurmans, and Norouzi 2019) empirically demonstrated that the stabilization of Deep Q-learning networks using a single data set can be achieved by training at the same time a multitude of different Deep Q-value Networks with their weights differently initialized. During training the final estimate of the Q-value will be a normalized *random* linear combination of the output of the intermediate Q-functions, while in the end they will just consider as the final Q-value estimate their average. The linear combination step is equivalent to a Dropout layer in a neural network. By doing so the final output will be stabler and if a network will be more affected than another by an OOD action back-up, the final average over the random ensemble will likely mitigate this error. The authors called this neural network architecture Random Ensemble Mixture (REM).

Generative model learning for Model-based approaches

Steps forward in the development of model based RL using a single batch have been done respectively in MOPO and MOReL (Yu et al. 2020; Kidambi et al. 2020). In both cases, a generative model is first learnt from the batch and then used to create new transitions. On the augmented data set then a model-free algorithm is applied. In this fashion, since we can use the generative model to "explore" the $S \times A$ space, the error in the Bellman back-up will not be induced directly by ill sampled regions but by the *epistemic error* of the model.

Intuitively, the more chaotic is the underlying system, the greater will a trajectory generated by the learnt model diverge from a real one given the very same starting state distribution and an identical sequence of actions to apply. Broadly speaking, as described below, the two methods build a penalized (MOPO) and pessimistic (MOReL) MDP whose optimal policies are encouraged to visit regions of $S \times A$ where the epistemic error is expected to be little (MOPO), or areas that would be likely to be sampled by the same distribution dynamics that generated the batch (MOReL).

Model Error Penalized MDP In MOPO, defining as $\eta_M[\pi] \stackrel{\text{def}}{=} \mathrm{E}_{s \sim \mu_0} [V_M^{\pi}(s)]$ as the performance of a policy for the MDP M, a theoretical bound for $\eta_M[\pi] - \eta_{\hat{M}}[\pi]$ is recovered. In particular, they show that:

$$\eta_M[\pi] \geq \mathrm{E}_{(s,a) \sim \rho_{\hat{M}}^{\pi}} \Big[r(s,a) - \frac{\gamma}{1-\gamma} \max_{s'} \big(V_M^{\pi}(s')\big) \cdot D_{TV}\Big(T(\cdot|s,a), \hat{T}(\cdot|s,a)\Big) \Big] = \eta_{\tilde{M}}[\pi] \quad (1)$$

where $\rho_{\hat{M}}^{\pi}$ is the discounted state-action distribution of transitions along the Markov Chain induced by $\hat{T}$ and π starting from the initial state distribution μ_0.

The right hand side is the performance of the MDP $\tilde{M}$ whose dynamics is driven by $\hat{T}$, but with reward function penalized by a term which is directly proportional to the

total variation distance between the true and the inferred transition functions. Since both D_{TV} and $\max_{s'} V^{\pi}_{M}(s')$ are unknown, in the practical implementation the penalty is replaced by $\lambda u(s,a)$ where λ is an hyperparameter and $u : S \times A \rightarrow [0, +\infty)$ such that $u(s,a) \geq D_{TV}(T(\cdot|s,a), \hat{T}(\cdot|s,a)) \; \forall (s,a) \in S \times A$. Therefore finding the optimal policy for the penalized MDP amounts to obtaining the policy that maximizes the lower bound on η_M.

Notwithstanding, we believe that their bound is greatly dependent on the choice of a proper hyperparameter λ and function u, which is not trivial for stochastic MDPs while it can be appropriately approximated by the covariance of a Gaussian Process for deterministic environments like the one used as test-cases in their paper. Moreover, if the penalization is too big the lower threshold will be likely of little use. Imagine the extreme situation where $\forall (s,a) \in S \times A$, $r > 0$ and $r - \lambda u < 0$. In this case $\eta_M[\pi] > 0 \; \forall \pi$ trivially, while, calling $\tilde{M}$ the reward penalized MDP with dynamics driven by $\hat{T}$, $\eta_{\tilde{M}}[\pi] < 0$. Therefore, maximizing the bound will not necessarily lead to a policy that works better than chance on the real MDP.

In (Yu et al. 2020) the performance of a policy is defined as the expected value of V^{π}_{M} over the starting state distribution μ_0. This starting state distribution is then interpreted as the distribution of states in the batch. However, the latter could be much different from the true starting state distribution if the batch is of modest size.

Therefore a more robust definition could be $\eta_M[\mu^{\pi}_{M}, \pi] \overset{\text{def}}{=} \mathrm{E}_{s \sim \mu^{\pi}_{M}}[V^{\pi}_{M}(s)]$, where μ^{π}_{M} is the stationary distribution of states (if it exists) for the MDP M with dynamics dictated by the policy π. Using this new definition, the lower bound on $\eta_M[\mu_M, \pi]$ acquires an extra term dependent on the difference $\Delta^{\pi}_{\hat{M},M}(s) = \mu^{\pi}_{\hat{M}}(s) - \mu^{\pi}_{M}(s)$. Indeed, the performance of a policy in the true MDP can be expressed as:

$$\begin{aligned} \eta_M\left[\mu^{\pi}_{M}, \pi\right] &= \mathrm{E}_{s \sim \mu^{\pi}_{M}}\left[V^{\pi}_{M}(s)\right] \\ &= \mathrm{E}_{\mu^{\pi}_{M}}\left[V^{\pi}_{M}(s)\right] - \int_S d\mu(s) \Delta^{\pi}_{\hat{M},M}(s) V^{\pi}_{M}(s) \end{aligned}$$

where, $d\mu(s)$ is a measure over the state space. It is then possible to obtain the same bound of Equation (1) but with the extra term dependent on the integral over the state space of the discrepancy between the stationary distributions:

$$\begin{aligned} &\eta_{\hat{M}}\left[\mu^{\pi}_{\hat{M}}, \pi\right] - \eta_M\left[\mu^{\pi}_{M}, \pi\right] = \\ &= \mathrm{E}_{s \sim \mu^{\pi}_{\hat{M}}}\left[V^{\pi}_{\hat{M}}(s)\right] - \mathrm{E}_{s \sim \mu^{\pi}_{M}}\left[V^{\pi}_{M}(s)\right] = \\ &= \mathrm{E}_{s \sim \mu^{\pi}_{\hat{M}}}\left[V^{\pi}_{\hat{M}}(s) - V^{\pi}_{M}(s)\right] + \int_S d\mu(s) \Delta^{\pi}_{\hat{M},M}(s) V^{\pi}_{M}(s) \end{aligned}$$

where the expected value over the distribution $\mu^{\pi}_{\hat{M}}$ is similar to the one computed in (Yu et al. 2020) but with $\mu_0 = \mu^{\pi}_{\hat{M}}$. Therefore the right hand side can be bounded from below:

$$\begin{aligned} \eta_{\hat{M}}\left[\mu^{\pi}_{\hat{M}}, \pi\right] - \eta_M\left[\mu^{\pi}_{M}, \pi\right] \leq \frac{\gamma}{1-\gamma} \max_{s'}\left(V^{\pi}_{M}(s')\right) \\ \cdot \mathrm{E}_{(s,a) \sim \rho^{\pi}_{M}}\left[D_{TV}\left(T(\cdot|s,a), \hat{T}(\cdot|s,a)\right)\right] \\ + \int_S d\mu(s) \Delta^{\pi}_{\hat{M},M}(s) V^{\pi}_{M}(s) \end{aligned}$$

where $\rho^{\pi}_{\hat{M}}$ now is the discounted state-action distribution of transitions along the Markov Chain induced by $\hat{T}$ and π starting from the stationary distribution $\mu^{\pi}_{\hat{M}}$.
Exploiting the definition of penalized MDP $\tilde{M}$:

$$\eta_M\left[\mu^{\pi}_{M}, \pi\right] \geq \eta_{\tilde{M}}\left[\mu^{\pi}_{\hat{M}}, \pi\right] - \int_S d\mu(s) \Delta^{\pi}_{\hat{M},M}(s) V^{\pi}_{M}(s)$$

The latter can again be bounded from above by plugging in the absolute value of Δ and the max of V over the state space:

$$\begin{aligned} \eta_M[\mu^{\pi}_{M}, \pi] \quad \geq \quad & \eta_{\tilde{M}}[\mu^{\pi}_{\hat{M}}, \pi] \\ & - \max_{s'} V^{\pi}_{M}(s') \int_S d\mu(s) |\Delta^{\pi}_{\hat{M},M}(s)| \end{aligned}$$

It is remarkable that now the optimal policy for $\tilde{M}$ does not necessarily maximizes the bound. Assuming that our algorithm is monotonically improving the policy, it could be then convenient to stop it earlier and settle for a sub-optimal policy which in turn maximizes the bound. It's all about balancing the trade-off between the optimality condition for $\tilde{M}$ and the discrepancy within the stationary distributions. The newly added term is unfortunately intractable due to the lack of knowledge about the MDP.

In the implementation of MOPO new trajectories are generated starting from states already present in the batch up to h following time steps. Ablation experiments have shown that the roll-out horizon h is indeed required to obtain good results. We suspect that the state distributional shift that was neglected is to be blamed for the occurrence of the behaviour. Generating data that are not so far away from ones in the batch prevents the accumulation of model error, but this theoretical aspect, even if already mentioned in (Janner et al. 2019), should not affect the bound that aims to be valid on any uncertainty penalized MDP independently of other factors.

Pessimistic MDP The authors of MOReL define an MDP with an extra absorbing state y. The state space of the pessimistic MDP is $\tilde{S} = S \cup \{y\}$, while the transition function

$$\tilde{T}(s'|s,a) = \begin{cases} \delta_{s',y} & \text{if } D_{TV}\left(\hat{T}(\cdot|s,a), T(\cdot|s,a)\right) > \theta, \\ \delta_{s',y} & \text{else if } s = y, \\ \hat{T}(s'|s,a) & \text{otherwise.} \end{cases}$$

The reward function R is identical to the original one except for y: $R(y,a) = -\kappa \; \forall a \in A$.
θ is a freely chosen threshold and $\kappa >> 0$ is a penalty. Essentially if the model error is greater than θ the agent will end up for sure in the strongly penalized absorbing state. Therefore any optimal policy for the pessimistic MDP will try to avoid transitions for which the model error is high. The optimal policy $\hat{\pi}$ when applied on the real MDP bounds from above the performance of the optimal policy of the true MDP.

$$\begin{aligned} \frac{4R_{max}}{1-\gamma}\left(\zeta(\mu_0, \hat{\mu}_0, T, \hat{T}) + \mathrm{E}\left[\gamma^{T^{\pi^*}_{U}}\right]\right) \\ \geq \eta_M[\mu_0, \pi^*] - \eta_M[\mu_0, \hat{\pi}] \end{aligned}$$

with,

$$\zeta(\mu_0, \hat{\mu}_0, T, \hat{T}) = D_{TV}\left(\mu_0, \hat{\mu}_0\right) + \frac{\gamma}{1-\gamma} \max_{s,a} D_{TV}(\hat{T}, T)$$

The two performances are similar if the D_{TV} between the real starting state distribution and the one inferred from the batch is negligible, if the maximum model error is little, and also, if the expected value of the first hitting time of the absorbing state while applying the policy π^* in the pessimistic model $\gamma^{T_{\mathcal{U}}^{\pi^*}}$ is small.

Again, in a practical implementation the choice of good estimators of the epistemic error, of the distributional distance between starting states, and also, of the first hitting time is of crucial importance. In the large batch regime the authors neglect the first two terms and focus only on the expectation of the first hitting time which can be bounded from the above by the *discounted* distribution of visits to $(s, a) \in \mathcal{U}$, where $\mathcal{U}$ represents the *unknown* state-action pairs that lead to the absorbing state, when applying $a \sim \pi^*$.The late distribution can be in turn bounded by a term proportional to the support mismatch of the distribution of states that were never sampled in the original data set.

3 Discussion

Theoretical bounds and function approximators

The theoretical bounds which justify the creation of the previously listed algorithms rely either on a penalty or on a regularization term proportional to a sort of uncertainty that obnubilates our knowledge about the underlying system. Sometimes the penalty is expressed as an estimate of the epistemic model error, other times as a difference between starting or stationary state distributions, finally it can be quantified as Out Of Distributions (OOD) state-action pairs with respect to the policy used during the collection of the data set.

The penalty or regularization term is often proportional to an upper bound of the value function or to a free hyperparameter. As we have seen the latter statement implies that when this constant is too big the intractable performance of a policy on the real MDP is bounded by a tractable term which unfortunately will be of little use.

Only REM stabilizes the accumulation of the error in Q-learning thanks to a constraintless weighted random ensemble average. Despite its nicety, the stabilization is not properly a goal-oriented correction to the deviation of the optimal Q-value estimated using a single batch from the real one.

Model-based approaches also learn a function $\hat{R}$, however there is no term linked to the uncertainty in the evaluation of the reward in the batch penalized resolution scheme for offline learning algorithms. We believe that such a term proportional to the reward error is not truly necessary since we expect it to be stemmed from the same regions of $S \times A$ that are badly sampled in the data set $\mathcal{B}$ and considering that the penalization is already applied on the reward or value functions.

Finding a proper estimator of the *errors* is not trivial. The algorithms were often tested on deterministic environments where a reasonable estimator of the model error can be achieved by the maximal variance in between an ensemble of different Gaussian models. Since it's reasonable to expect that the model error will be high in regions that were ill-sampled in the batch, another way to measure it could be getting an estimate of the probability that a given (s, a) could have been generated by the same process that gave birth to the batch. Therefore estimating the probability distribution of (s, a, s') in the true MDP with policy $\pi_{\mathcal{B}}$ is a priority.

The most practical way of learning a probability distribution function without a prior could be to use a GAN (Goodfellow et al. 2014). A GAN is comprised of a Generator and a Discriminator. The first is a neural network which receives random noise as an input and generates an output with the same shape of the data in a training set. The second gets an input with the correct shape and provides as output a real number. While training the Discriminator tries to identify which data was present in the training batch between samples that really populate it and the output of the generator. The higher the output of the Discriminator on a sample will be, the most likely that sample will be in the batch if the Discriminator is well trained. At the same time the goal of the Generator will be that of fooling the Discriminator. The loss functions minimized during the training are peculiar of a min-max game. Sophisticated GANs architecture can use a well trained Generator to build fake samples that could fool even a human. A striking example is StyleGAN2 by NVIDIA (Karras et al. 2019) which can generate high quality dimensional pictures of people that do not really exist.

We believe that the use of a GAN's Discriminator trained on $\mathcal{B}$ to obtain an estimate of the log-likelihood of a transition (s, a, r, s') with respect to the unknown transition distribution should be a promising venue for penalizing the reward and/or the value functions with a more pertinent estimator of the distance between the true distribution of data and the one we can infer from a single batch. In this way we may be able to recover an informative quantity about the distributional shifts in a non parametric way that is independent of any possible prior and might, in principle, also work for systems driven by a stochastic time evolution. Doing so we would drop off the Gaussian assumption that has been so far used in almost all of the model-based techniques.

However, GANs have some weak spots: the training is unstable because the loss function is not convex, the procedure takes time, and they suffer from mode collapse. The latter is maybe the most problematic issue since when the unknown latent distributions is multi-modal the Generator may focus on building up samples that benefits from characteristics that are typical only of a little slice of the whole set. Since the Discriminator is trained alongside the Generator, it will learn to recognize samples that are typical of that specific mode. Several approaches to mitigate mode collapse (Ghosh et al. 2018) and training instability (Arjovsky, Chintala, and Bottou 2017) have been attempted so far, but the issues can be still considered unsolved.

Off-policy Evaluation

It is necessary to find statistically robust methods which are able of estimating how well an algorithm will run in the real

world without interacting with it. Off-policy evaluation is an active field which would require a summary of its own. Recent approaches utilize optimized versions of Importance Sampling to estimate the unknown ratio between stationary distributions of states under dynamics driven by different policies. Recent works propose to create a sort of Discriminator and optimize a min-max loss function to serve this purpose (Liu et al. 2018; Zhang et al. 2020).

The application of a min-max optimization loss function in the field strengthens our intuition that the implementation of a GAN anyhow in the estimation of the distributional shift might be useful.

Batch Quality and Size Scalability

It is of compelling necessity to develop and test the baselines in common environments using the same batches to shine a light onto the change in performance of the different paradigms when the quality (and the variety) and the size of the available data increase. D4RL (Datasets for Deep Data-Driven Reinforcement Learning) is a collection of data sets recorded using policies of different qualities (random, medium, expert) on the typical benchmarking environments used by the RL community (OpenGym, MuJoCo, Atari etc.) (Fu et al. 2020). However, the offline learning community has yet to settle to the use of a common pipeline for benchmarks. The results achieved by MOReL using D4RL are reported in Table 1, for comparison with different baselines examine the reference (Kidambi et al. 2020). Independently of the quality of the batch we notice an improvement in the performance, expressed as the average cumulative reward over a sequence of trajectories, of the optimal policy for the pessimistic MDP when evaluated in the true environment.

The results achieved with MOPO are reported in Table 2. MOPO performs better than all previous baselines on randomly generated batches and on data sets which consist of the full replay buffer of a Soft-Actor Critic (SAC) trained partially up to an environmental specific performance threshold. Surprisingly, on batches generated with the sub-optimal trained SAC the best baselines are BRAC with value function penalty and BEAR. The main difference between the last two types of data sets is that while the latter is generated with a fixed policy, the previous one is a collection of transitions gathered with a mixture of differently performing policies. When the sub-optimal policies are not so bad, it seems reasonable to just slightly modify them to obtain better results, hence BEAR and BRAC looks like viable methods. However, when the overall batch policy is not so good, constraining the reward with respect to the model error (and the transitions close to the ones present in the batch up to a roll-out horizon) can be more fulfilling.

As mentioned also by the authors of BRAC, their algorithm when applied to small data sets becomes more susceptible to the choice of the hyperparameters. This is probably because on small data set the distributional shift / model error can become significant. It is crystal clear that the field needs better theoretical foundations and better algorithms in order to learn more safe and performing policies from small batches collected with strategies of any quality, even uniform random ones.

Environment	Pure-Random	Pure-Partial
Hopper-v2	2354 ± 443 (20)	3642 ± 54 (1376)
HalfCheetah-v2	2698 ± 230 (-638)	6028 ± 192 (4198)
Walker2d-v2	1290 ± 325 (-7)	3709 ± 159 (1463)
Ant-v2	1001 ± 3 (-263)	3663 ± 247 (1154)

Table 1: Average cumulative return of the policy obtained with MOReL as reported in (Kidambi et al. 2020). A Pure-Partial policy is a partially trained suboptimal policy. The number between parentheses is the average cumulative reward with the batch collecting policy. All results are averaged over 5 random seeds.

4 Conclusions

In this paper we have examined the state-of-the-art RL and planning algorithms motivated by the necessity to exploit their application to improve offline learning using a single batch of collected experiences. This is challenging problem of crucial importance for the development of intelligent agents. In particular, when the interaction of such agents with the environment is expensive, risky or unpractical. Our goal was that of providing to the reader a self-contained summary of the general ideas that flow behind the main topic. For simplicity, we focused on MDPs but once the listed difficulties will be addressed we aim to extend the discussion to Partially Observable MDPs which are a more appropriate object to describe the interaction of agents in partial observable environments. We started with a recap of MDPs and resolution schemes. Then we presented the single batch learning and planning problem. Our main contribution is an outline of model-free and model-based batch RL algorithms while providing comments on size scalability, efficiency and on the usefulness of theoretical bounds. In particular, we proposed an improvement of the definition of performance of the value function following a specific policy that led us to believing that a sub-optimal policy for a reward uncertainty penalized MDP can be better than the optimal one when applied in the true environment. Secondly, we analyzed the penalization introduced in all sorts of offline-learning algorithms. We showed that if the coefficients multiplying the distributional shift estimator are too big then the theoretical threshold which bounds the performance of the policy applied in the real world is always respected, and therefore of little practical utility. We also advised the future implementation of GANs for a better estimate of distributional shifts and model errors. Indeed, estimators that optimizes a min-max loss function give hint that this might be a viable solution.

References

Agarwal, R.; Schuurmans, D.; and Norouzi, M. 2019. An optimistic perspective on offline reinforcement learning. *arXiv: Learning*.

Arjovsky, M.; Chintala, S.; and Bottou, L. 2017. Wasserstein gan. *ArXiv* abs/1701.07875.

Baek, S.; Kwon, H.; Yoder, J.; and Pack, D. 2013. Optimal path planning of a target-following fixed-wing uav using sequential decision processes. 2955–2962.

Data set type	Environment	Batch Mean	Batch Max	SAC	BEAR	BRAC-vp	MBPO	MOPO
random	halfcheetah	-303.2	-0.1	3502.0	2885.6	3207.3	3533.0	**3679.8**
random	hopper	299.26	365.9	347.7	289.5	370.5	126.6	**412.8**
random	walker2d	0.9	57.3	192.0	307.6	23.9	395.9	**596.3**
medium	halfcheetah	3953.0	4410.7	-808.6	4508.7	**5365.3**	3230.0	4706.9
medium	hopper	1021.7	3254.3	5.7	**1527.9**	1030.0	137.8	840.9
medium	walker2d	498.4	3752.7	44.2	1526.7	**3734.3**	582.5	645.5
mixed	halfcheetah	2300.6	4834.2	-581.3	4211.3	5413.8	5598.4	**6418.3**
mixed	hopper	470.5	1377.9	93.3	802.7	5.3	1599.2	**2988.7**
mixed	walker2d	358.4	1956.5	87.8	495.3	44.5	1021.8	**1540.7**
med-expert	halfcheetah	8074.9	12940.2	-55.7	6132.5	5342.4	926.6	**6913.5**
med-expert	hopper	1850.5	3760.5	32.9	109.8	5.1	**1803.6**	1663.5
med-expert	walker2d	1062.3	5408.6	-5.1	1193.6	**3058.0**	351.7	2527.1

Table 2: Results for D4RL datasets as reported in (Yu et al. 2020). Each number is the average undiscounted return of the policy at the last iteration of training, averaged over 3 random seeds. Data set types depend on the policy used to collect the batch: random (random policy), medium (suboptimally trained agent with a Soft Actor-Critic), mixed (adoperate as batch the replay buffer use to train a Soft-Actor Critic until an envirnomental specific threshold is reached), medium-expert (mix of an optimal policy and a random or a partially trained one). SAC column stands for a Soft Actor Critic (model-free) agent. BRAC-vp is the version of BRAC with the value penalty. MBPO is the vanilla model-based algorithm described in (Janner et al. 2019). Check the reference (Yu et al. 2020) for details about the hyperparameters.

Barth-Maron, G.; Hoffman, M. W.; Budden, D.; Dabney, W.; Horgan, D.; Dhruva, T.; Muldal, A.; Heess, N. M. O.; and Lillicrap, T. P. 2018. Distributed distributional deterministic policy gradients. *ArXiv* abs/1804.08617.

Chanel, C. P. C.; Roy, R. N.; Dehais, F.; and Drougard, N. 2020. Towards mixed-initiative human–robot interaction: Assessment of discriminative physiological and behavioral features for performance prediction. *Sensors* 20:296.

Chen, J., and Jiang, N. 2019. Information-theoretic considerations in batch reinforcement learning. In Chaudhuri, K., and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, 1042–1051. PMLR.

Chua, K.; Calandra, R.; McAllister, R.; and Levine, S. 2018. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *ArXiv* abs/1805.12114.

Csáji, B. C. 2001. Approximation with artificial neural networks.

Deisenroth, M. P., and Rasmussen, C. E. 2011. Pilco: A model-based and data-efficient approach to policy search. In *In Proceedings of the International Conference on Machine Learning*.

Fu, J.; Kumar, A.; Nachum, O.; Tucker, G.; and Levine, S. 2020. D4rl: Datasets for deep data-driven reinforcement learning. *ArXiv* abs/2004.07219.

Fujimoto, S.; Conti, E.; Ghavamzadeh, M.; and Pineau, J. 2019. Benchmarking batch deep reinforcement learning algorithms. *ArXiv* abs/1910.01708.

Fujimoto, S.; Meger, D.; and Precup, D. 2019. Off-policy deep reinforcement learning without exploration. volume 97 of *Proceedings of Machine Learning Research*, 2052–2062. Long Beach, California, USA: PMLR.

Ghosh, A.; Kulharia, V.; Namboodiri, V. P.; Torr, P. H. S.; and Dokania, P. K. 2018. Multi-agent diverse generative adversarial networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* 8513–8521.

Goodfellow, I. J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A. C.; and Bengio, Y. 2014. Generative adversarial nets. In *NIPS*.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*.

Hafner, D.; Lillicrap, T. P.; Fischer, I. S.; Villegas, R.; Ha, D. R.; Lee, H.; and Davidson, J. 2019. Learning latent dynamics for planning from pixels. In *ICML*.

Hafner, D.; Lillicrap, T. P.; Ba, J.; and Norouzi, M. 2020. Dream to control: Learning behaviors by latent imagination. *ArXiv* abs/1912.01603.

Hessel, M.; Modayil, J.; van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M. G.; and Silver, D. 2018. Rainbow: Combining improvements in deep reinforcement learning. *ArXiv* abs/1710.02298.

Janner, M.; Fu, J.; Zhang, M.; and Levine, S. 2019. When to trust your model: Model-based policy optimization. In *NeurIPS*.

Jaques, N.; Ghandeharioun, A.; Shen, J. H.; Ferguson, C.; Lapedriza, À.; Jones, N. J.; Gu, S.; and Picard, R. W. 2019. Way off-policy batch deep reinforcement learning of implicit human preferences in dialog. *ArXiv* abs/1907.00456.

Jonsson, A. 2018. Deep reinforcement learning in medicine. *Kidney Diseases* 5:1–5.

Kaiser, L.; Babaeizadeh, M.; Milos, P.; Osinski, B.; Campbell, R. H.; Czechowski, K.; Erhan, D.; Finn, C.; Kozakowski, P.; Levine, S.; Sepassi, R.; Tucker, G.; and Michalewski, H. 2020. Model-based reinforcement learning for atari. *ArXiv* abs/1903.00374.

Karras, T.; Laine, S.; Aittala, M.; Hellsten, J.; Lehtinen, J.; and Aila, T. 2019. Analyzing and improving the image quality of stylegan. *ArXiv* abs/1912.04958.

Keller, T., and Eyerich, P. 2012. Prost: Probabilistic planning based on uct. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *ICAPS*. AAAI.

Kidambi, R.; Rajeswaran, A.; Netrapalli, P.; and Joachims, T. 2020. Morel. *ArXiv* abs/2005.05951.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. volume 2006, 282–293.

Kumar, A.; Fu, J.; Soh, M.; Tucker, G.; and Levine, S. 2019. Stabilizing off-policy q-learning via bootstrapping error reduction. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; dAlché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc. 11784–11794.

Levine, S.; Kumar, A.; Tucker, G.; and Fu, J. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *ArXiv* abs/2005.01643.

Liu, Q.; Li, L.; Tang, Z.; and Zhou, D. 2018. Breaking the curse of horizon: Infinite-horizon off-policy estimation. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc. 5356–5366.

Mansley, C.; Weinstein, A.; and Littman, M. 2011. Sample-based planning for continuous action markov decision processes. In *Twenty-First International Conference on Automated Planning and Scheduling*.

Mausam, and Kolobov, A. 2012. *Planning with Markov Decision Processes: An AI Perspective*.

Mirchevska, B.; Pek, C.; Werling, M.; Althoff, M.; and Boedecker, J. 2018. High-level decision making for safe and reasonable autonomous lane changing using reinforcement learning. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 2156–2162.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *ICML*.

Munos, R., and Moore, A. 2002. Variable resolution discretization in optimal control. *Machine Learning* 49, Numbersâ:291–.

Munos, R. 2003. Error bounds for approximate policy iteration. In *ICML*.

Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M. I.; and Moritz, P. 2015. Trust region policy optimization. In *ICML*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–503.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv* abs/1712.01815.

Sutton, R. S., and Barto, A. G. 1998. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks* 16:285–286.

Sutton, R. S.; McAllester, D. A.; Singh, S. P.; and Mansour, Y. 1999. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*.

van Hasselt, H. P.; Hessel, M.; and Aslanides, J. 2019. When to use parametric models in reinforcement learning? In Wallach, H.; Larochelle, H.; Beygelzimer, A.; dAlché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc. 14322–14333.

Watkins, C., and Dayan, P. 1992. Q-learning. *Machine Learning* 8:279–292.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* 8(3–4):229–256.

Wu, Y.; Tucker, G.; and Nachum, O. 2019. Behavior regularized offline reinforcement learning. *ArXiv* abs/1911.11361.

Yu, T.; Thomas, G.; Yu, L.; Ermon, S.; Zou, J.; Levine, S.; Finn, C.; and Ma, T. 2020. Mopo: Model-based offline policy optimization. *arXiv preprint arXiv:2005.13239*.

Zhang, R.; Dai, B.; Li, L.; and Schuurmans, D. 2020. Gendice: Generalized offline estimation of stationary values. In *International Conference on Learning Representations*.

Real-time Planning as Data-driven Decision-making

Maximilian Fickert[*1], Tianyi Gu[*2], Leonhard Staut[*1], Sai Lekyang[2], Wheeler Ruml[2], Jörg Hoffmann[1], Marek Petrik[2]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]Department of Computer Science, University of New Hampshire, USA
{fickert,hoffmann}@cs.uni-saarland.de, {gu, ruml, mpetrik}@cs.unh.edu,
s9ldstau@stud.uni-saarland.de, sai.lekyang@gmail.com

Abstract

If reinforcement learning (RL) is the use of incrementally gathered data to drive decision-making, then any heuristic search strategy is fundamentally an RL process. This is perhaps clearest in real-time planning, where an agent must select the next action to take within a fixed time bound. Even in deterministic domains, real-time action selection inherently suffers from uncertainty about those portions of the state space that have not yet been computed by the lookahead search. In this paper, we present new results in a line of research that explores how an agent can benefit from metareasoning about this uncertainty. Taking inspiration from prior work in distributional methods from RL, the Nancy search framework represents its uncertainty explicitly as beliefs over cost-to-go. Nancy then expands nodes so as to minimize the expected regret in case a non-optimal action is chosen. We present detailed results showing how beliefs can be informed by prior experience and we experimentally compare Nancy against both conventional real-time search algorithms like LSS-LRTA* and approaches from RL that exploit uncertainty, such as Monte Carlo tree search and Kaelbling's interval estimation. We find that Nancy generally outperforms previous methods, particularly on more difficult problems. This work illustrates how the distributional perspective from Bayesian RL can be adapted to deterministic planning settings, and how deterministic planning can provide useful testbeds for methods that metareason about uncertainty during planning.

Introduction

Some AI applications are subject to real-time constraints, where the agent must select its next action within a fixed time bound. Typical examples include user interfaces or the control of cyber-physical systems, where unbounded pauses between system actions are undesirable or potentially dangerous. Real-time planning methods tackle this problem setting. Given a forward model of the domain dynamics, the planning agent incrementally plans toward a goal, trying to minimize the total cost of the resulting trajectory. (Unlike some reinforcement learning settings, we assume here that the state transition function can be applied on arbitrary states.) Many real-time heuristic search methods follow the basic three-phase paradigm set down in the seminal work of Korf (1990):

1) starting at the agent's current state, expand a fixed number of nodes according to the given time bound to form a lookahead search space (LSS);
2) use the heuristic values of the frontier nodes in combination with the path costs incurred to reach them to estimate the cost-to-goal for each currently-applicable action, and commit to the action with the lowest estimate;
3) to prevent the agent from cycling if it returns to the same state in the future, update the heuristic values of one or more states in the LSS.

For example, in the popular and typical algorithm LSS-LRTA* (Koenig and Sun 2008), the lookahead in step 1 is performed using A* (Hart, Nilsson, and Raphael 1968), the value estimates in step 2 are implicitly calculated for each node as the minimum f value among its successors (the 'minimin' backup), and the learning in step 3 is performed by updating h values for all nodes in the LSS using a variant of Dijkstra's algorithm. Similar methods are used in reinforcement learning (RL), such as RTDP (Barto, Bradtke, and Singh 1995). While elegant and often successful, this paradigm does not explicitly address the uncertainty inherent in real-time planning. As the search computes only a miniscule fraction of the state space (up to the LSS frontier), it must commit to action decisions subject to uncertainty about the part of the state space beyond that frontier. In this paper, we advance a line of research inspired by work in RL, viewing real-time planning as a form of decision-making under uncertainty.

For example, as pointed out by Mutchler (1986), A*'s policy of expanding the frontier node with the lowest f value is not, in general, the optimal way to make use of a limited number of node expansions. If we view the agent as facing a decision under uncertainty, it is sometimes beneficial to gain knowledge about inferior-looking options. For example, consider the situation depicted in Figure 1. The figure shows the agent's current beliefs about the expected total plan cost that would be incurred by committing to the ap-

[*]These authors contributed equally to this work.

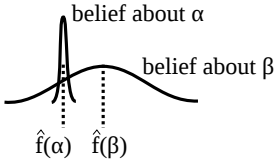


Figure 1: Should an agent expand nodes under α or β?

plicable actions α and β respectively. Each such belief is a probability distribution over possible costs, with the expected value denoted by $\hat{f}(\cdot)$. In the displayed situation, the agent is quite certain about the value of α but quite uncertain about the value of β. It is likely that α is better, but there is a significant possibility that β may be better instead. Given this, expanding frontier nodes under α can be less useful than expanding under β, even though β is believed to have a higher expected cost. It can be more important to explore the possibility that a poorly-understood option might in fact be great than to nail down the exact value of a good-looking option that is already well understood. Note that this is distinct from the well-known exploration-versus-exploration dilemma, as we are given the exact amount of time that we should spend exploring. We also note that this problem does not arise in off-line optimal search, where every node whose f value is less than the optimal solution cost must be expanded.

Although insights such as this derived from a reasoning under uncertainty perspective seem powerful, they are not widely applied in deterministic planning. Recently, a real-time heuristic search framework called Nancy has been proposed, that uses explicit reasoning about uncertainty to guide its search (Mitchell et al. 2019; Fickert et al. 2020). Nancy treats the traditional heuristic estimate of cost-to-go from a given state as uncertain evidence, inducing beliefs – probability distributions – over the actual remaining cost. Nancy uses a node-expansion strategy that minimizes the expected regret in case a non-optimal action is chosen.

Two alternative methods have been proposed to define the beliefs at frontier nodes. The first one is assumption-based (Mitchell et al. 2019), assuming Gaussian distributions as in prior work (O'Ceallaigh and Ruml 2015) and tuning them on-line. As these assumptions are not necessarily justified, the second method relies on data from previous search experience (Fickert et al. 2020), approximating the beliefs from data gathered in the same problem family.

In this paper, we provide further experimental results for Nancy. First, we examine the belief distributions that are learned from training data, showing in detail for the first time how heuristic error varies across planning problems. Second, we provide a comparison to methods previously proposed in RL for exploiting uncertainty in search. Monte-Carlo tree search (MCTS) methods (Kocsis and Szepesvári 2006; Keller and Helmert 2013; Feldman and Domshlak 2014; Silver et al. 2018), which originated in probabilistic planning, maintain node-value averages along with node-visited counts and use these to give a boost to actions with uncertain values. The previous work perhaps closest to ours, Interval Estimation (Kaelbling 1993), explicitly represents uncertainty and uses it to guide search effort. We adapt these methods to our setting. Overall, we find that Nancy typically outperforms previous methods despite its metareasoning overhead, suggesting that it makes better use of limited node expansions. Nancy's success illustrates the strength of adopting the reasoning under uncertainty perspective from RL for resource-bounded decision-making, even in completely deterministic problem domains. It also provides a clearly defined setting might interest RL researchers that isolates the uncertainty due to computational resource bounds from that present in MDPs due to stochastic actions effects.

Previous Work

There have been many proposals for real-time heuristic search methods. Some, like LSS-LRTA*, are very general and apply to any state space search problem. Others assume undirected state spaces in which it is always possible to immediately return to a node's parent state. Several are specialized to grid-based pathfinding. In this paper, we choose LSS-LRTA* as our point of comparison due to its simplicity and generality.

Mutchler (1986) raises the question of how best to allocate a limited number of expansions. His analysis considers complete binary trees of uniform depth where each edge is randomly assigned cost 1 with probability p and cost 0 otherwise. He proves that a minimum f expansion policy is not optimal for such trees in general, but that it is optimal for certain values of p and certain numbers of expansions. It is not clear how to apply these results to more realistic state spaces.

Pemberton and Korf (1994) point out that it can be useful to use different criteria for action selection versus node expansion. They use binary trees with random edge costs uniformly distributed between 0 and 1 and use a computer algebra package to generate code to compute exact $\hat{f}$ values under the assumption that only one or two tree levels remain until a goal (the 'last incremental decision problem'). As we will discuss in more detail below, this requires representing and reasoning about the distribution of possible values under child nodes in order to compute the distribution at each parent node. They conclude that a strategy based on expected values is barely better than the classic minimin strategy and impractical to compute for state spaces beyond tiny trees. They also investigate a method in which the nodes with minimum f are expanded and the action with minimum $\hat{f}$ is selected and find that it performs better than using f for both.

Given the pessimism surrounding exact estimates, Pemberton (1995) proposes an approximate method called k-best. Only the k best frontier nodes below a top-level action are used to compute its value, allowing a fixed inventory of equations derived in advance to be used to compute expected values during search. Although this approach did surpass minimin in experiments on random binary trees, Pemberton concludes that its complexity makes it impractical. It is also not clear how to apply these results beyond random binary trees.

Our problem setting bears a superficial similarity to the

exploration/exploitation trade-off examined in reinforcement learning. However, note that our central challenge is how to make use of a given number of expansions — we do not have to decide between exploring for more information (by expanding additional nodes) or exploiting our current estimates (by committing to the currently-best-looking action). DTA* (Russell and Wefald 1991) and Mo'RTS (O'Ceallaigh and Ruml 2015) are examples of real-time search algorithms that directly address that trade-off. Both are based on estimating the value of the information potentially gained by additional lookahead search and comparing this to a time penalty for the delay incurred. DTA* expands the frontier node with minimum f and Mo'RTS expands the frontier node with minimum $\hat{f}$.

MCTS algorithms such as UCT (Kocsis and Szepesvári 2006) share our motivation of recognizing the uncertainty in the agent's beliefs and trying to generate relevant parts of the state space. Tolpin and Shimony (2012) emphasize the purpose of lookahead as aiding in the choice of the agent's next action and, as we will below, they take an approach motivated by the value of information. Lieck and Toussaint (2017) investigate selective sampling for MCTS. However, unlike most work in MCTS, we focus on deterministic problems and we have no need to sample action transitions or perform roll-outs. Furthermore, real-time planning can arise in applications where perhaps only a dozen nodes can be generated per decision, a regime where MCTS algorithms can perform poorly, as a single roll-out may generate hundreds of nodes.

Work on active learning also emphasizes careful selection of computations to refine beliefs. For example, Frazier, Powell, and Dayanik (2008) present an approach they term 'the knowledge gradient' for allocating measurements subject to noise in order to maximize decision quality. More broadly, the notion of representing beliefs over values during learning and decision-making has been pursued in Bayesian reinforcement learning (Dearden, Friedman, and Russell 1998; McMahan, Likhachev, and Gordon 2005; Sanner et al. 2009; Bellemare, Dabney, and Munos 2017).

The Nancy Framework

A real-time heuristic search algorithm is made up of several parts that work together to choose the action that is most reasonable to execute next. The lookahead component determines in which direction to search, i.e. which node to expand next, in order to make the best use of the limited time. Successive lookahead steps create the local search space. The backup component runs after each lookahead step and updates the goal distance estimates of each search node based on its successors, by propagating the information from the leafs of the search tree up towards the root. In this section we give a complete description of the Nancy real-time search algorithm. Its lookahead is based on a the expected regret of making a non-optimal action choice, that is, α, the action with the smallest expected cost, turned out to have a higher actual cost than one of the alternatives β. This follows the classical characterization of risk as the sum of expected losses in a bad event (α had a higher cost than β) weighted by the probability of that event. Nancy's eponymous backup component propagates the belief distributions of the best-looking child to its parents.

Algorithm 1: Nancy

```
1  s := s_start
2  π_curr := ⟨⟩
3  while s⟦π_curr⟧ is not a goal state do
4      t := risk_lookahead(s)
5      π_curr := update_path(s, t, π_curr, π)
6      apply_next(s, π_curr)
7      backup(lss)
8  while s is not a goal state do
9      apply_next(s, π_curr)
10 fn apply_next(s, π_curr)
11     let a_0, ..., a_n be the action sequence of π_curr
12     s := s⟦a_0⟧
13     π_curr := ⟨a_1, ..., a_n⟩
14 fn update_path(s, t, π_curr, π)
15     if (π_curr = ⟨⟩ or
16         t is a goal state or
17         f̂(t) < f̂(s⟦π_curr⟧) or
18         f̂(t) = f̂(s⟦π_curr⟧) and ĥ(t) < ĥ(s⟦π_curr⟧))
       then
19         return π
20     return π_curr
```

The Components of Nancy

Algorithm 1 shows the pseudo-code for Nancy. The lookahead uses risk to guide the direction of the lookahead, building up the local search space (lss) and returning the overall best frontier node according to $\hat{f}$ (line 4), breaking ties by $\hat{h}$. The backup function is used to update the beliefs by backing up the beliefs from the frontier of the local search space towards the root. Fickert et al. (2020) prove that Nancy is complete and will always reach a goal under certain mild conditions.

Risk-based Lookahead We now explain Nancy's lookahead strategy in more detail. Lookahead is performed to minimize risk. The idea is to find the optimal action to execute next while also becoming more certain about possible alternative actions. Algorithm 2 shows the pseudo-code for the risk-based lookahead. In the lookahead phase, Nancy uses her first expansion to generate the top level actions (line 1). From that point forward, Nancy expands nodes such that an approximation of risk is minimized, until the expansion or time limit of the lookahead runs out. Each top-level action (TLA) has an associated open list (denoted by TLA.open in the pseudo-code) that is ordered by $\hat{f}$. Before each expansion, Nancy has to pick the open list where the expansion will take place. For this purpose, Nancy estimates $\mathcal{B}_{post}$ which denotes the updated belief Nancy expects after performing one expansion. The open list where Nancy expects

Algorithm 2: Risk-Based Lookahead

Input: s : state
Output: t : target state with minimal $\hat{f}$

1 Generate *TLAs*
2 **while** *lookahead limit is not reached* **do**
3 **for** *tla in TLAs* **do**
4 Swap in $\mathcal{B}_{post}(s[\![tla]\!])$ for $\mathcal{B}(s[\![tla]\!])$
5 $risk[s[\![tla]\!]] \coloneqq \texttt{risk}(TLAs)$
6 Restore original $\mathcal{B}(s[\![tla]\!])$
7 $chosen \coloneqq \arg\min_{tla \in TLAs}(risk[tla])$
8 $t \coloneqq chosen.open.pop\_min()$
9 **if** *t is goal* **then**
10 **return** t
11 **for** $a \in A(t)$ **do**
12 Estimate and cache $\mathcal{B}(t[\![a]\!])$ and $\mathcal{B}_{post}(t[\![a]\!])$
13 $\texttt{push}(chosen.open, t[\![a]\!], \hat{f}(t[\![a]\!]))$
14 $u \coloneqq chosen.open.min()$
15 $\mathcal{B}(s[\![chosen]\!]) \coloneqq \mathcal{B}(u) + g(u)$
16 $\mathcal{B}_{post}(s[\![chosen]\!]) \coloneqq \mathcal{B}_{post}(u) + g(u)$
17 $best \coloneqq \arg\min_{tla \in TLAs}(\hat{f}(s[\![tla]\!]))$
18 **return** $best.open.min()$

to arrive at a belief with minimal risk is then selected and the actual expansion follows, (Alg. 2, line 11). After each expansion, new information is obtained about the frontier of the local search space. To make use of this information, the belief at the top level needs to be updated, such that it agrees with the new best frontier node.

This lookahead process is repeated until the expansion or time limit is reached, or a goal state is selected for expansion. Once the lookahead phase ends, the search performs Nancy backups and executes the TLA with the lowest expected cost (Algorithm 1, line 12). In the learning phase, the beliefs $\mathcal{B}$ and post-expansion beliefs $\mathcal{B}_{post}$ of all nodes within the local search space are updated (Algorithm 1, line 7). This learning process performs a dynamic programming-like learning step to update the $\hat{h}$-values of the expanded states (like LSS-LRTA*).

Assumption-Based Nancy

Nancy's risk-based lookahead strategy relies on belief distributions over the remaining cost to a goal. To model such distributions, following O'Ceallaigh and Ruml (2015), we first build Gaussian distributions centered on $\hat{f}$ with a variance proportional to the difference between a node's $\hat{f}$ and f values:

$$B(n) \sim \mathcal{N}\Big(\hat{f}(n), \Big(\frac{\hat{f}(n) - f(n)}{2}\Big)^2\Big)$$

The mean value $\hat{f}(n)$ is estimated using one-step heuristic error estimate (Thayer, Dionne, and Ruml 2011). The variance model reflects the common assumption that heuristics are more accurate as one approaches a goal, because the difference between $f(n)$ and $\hat{f}(n)$ is proportional to $d(n)$, the estimated remaining search distance (number of steps-to-go). In this way, search experience is used to continually adjust the algorithm's skepticism of its heuristic and inform its beliefs as the search evolves. Secondly, the Gaussian beliefs were truncated from below at the admissible f value and above at three standard deviations.

Algorithm 3: Data Collection

1 Pick a set of training problem instances $\mathcal{T}$
2 Pick a search algorithm $\mathcal{S}$
3 **for** t *in* $\mathcal{T}$ **do**
4 Solve t with $\mathcal{S}$, while recording all expanded states
5 **for** *state* s *expanded by* $\mathcal{S}$ **do**
6 Solve s optimally
7 Store pair $(h(s), h^*(s))$

Data-Driven Nancy

The assumption-based instantiation of Nancy makes use of several assumptions about heuristic behavior. We next describe an alternative approach to obtain these beliefs, which is based on data. The idea of this method is to use statistics about heuristic behavior gathered in an offline phase prior to the search to construct the beliefs. Hence, some or all of the assumptions to construct beliefs at runtime are replaced with data. Here we cover the details of the data-driven variant of Nancy, which we call DDNancy.

Data Generation The purpose of Nancy's assumptions is to obtain a better estimate of the possible true goal distances $h^*(s)$ when only $h(s)$ is available. The approach to replace these assumptions is to run an offline training phase that learns the distribution of h^*-values.

Algorithm 3 shows a high-level overview of the data collection process. The h^* distributions are generated offline by collecting (h, h^*) pairs from a number of training instances. Each training instance is first solved by an initial search. Each state that was expanded by that search is then solved optimally, and its h and h^* values are stored. By collecting all these pairs, we obtain a set of h^* values for each h value, making up the distribution.

In its search, DDNancy uses the heuristic values to look up the corresponding distribution of h^* values. It may happen that DDNancy encounters a state with a heuristic value h that was not observed in the data gathering process. In that case, we perform an online extrapolation step on the data. We pick the largest $h' \leq h$ for which we have a distribution in the data set. The distribution for h is extrapolated by shifting the distribution of h' by the difference between h and h', i.e., adding $(h - h')$ to each data point in the distribution of h'. The newly created distribution is cached to have it available for the remaining search.

In the learning step, the data-driven instantiation of Nancy does not change the heuristic values. Instead, the change in heuristic value is transferred to the distributions, and the data

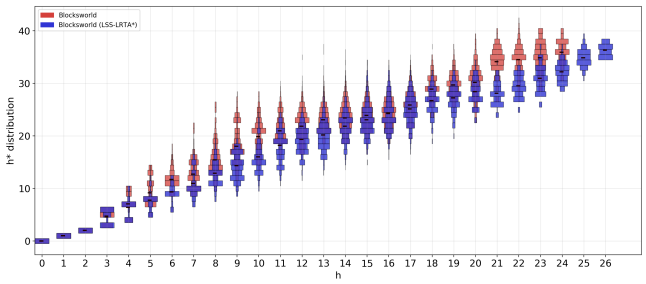


Figure 2: Beliefs in Blocksworld generated with weighted A* and LSS-LRTA*.

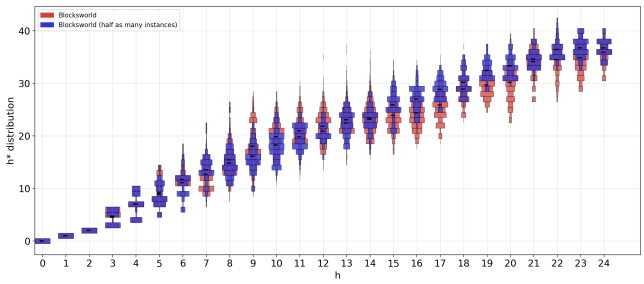


Figure 3: Beliefs in Blocksworld generated on 35 examples instances and on 17.

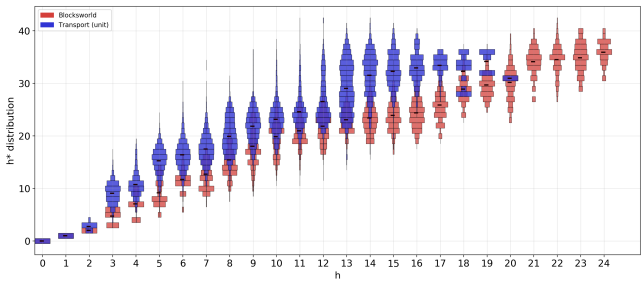


Figure 4: Beliefs generated for Blocksworld and Transport (unit-cost).

points are shifted accordingly (similar to the extrapolation procedure). In the implementation, we simply store the current shift value and a pointer to the corresponding distribution for each expanded state.

In the training process, the initial search is used to sample the states for the data collection (as the expanded states are then solved optimally). There are two key motivations to use this strategy instead of training on the entire state space. The first reason is practical feasibility. For instances above a certain size (e.g. Blocksworld with more than 10 blocks), solving the entire state space would require an unreasonable amount of time and memory. The second reason is that considering the entire state space may not make the data more accurate, as most of these states are not seen in the actual search. Instead, we want the process to focus on states that are representative for states encountered by Nancy, and inform the algorithm how the heuristic typically behaves on such a state. The initial search algorithm should therefore have similar behavior to Nancy to generate a good set of sample states, and improve the accuracy of the resulting distributions. We generate the data for each domain separately. While this requires additional work for each new domain DDNancy is intended to run on, the heuristic behavior can vary a lot between different domains, and domain-specific data can capture the behavior more accurately.

Since DDNancy is a suboptimal search algorithm, we also use a suboptimal search algorithm to sample the states for the data generation. Figure 2 shows the belief distributions that result from using weighted A* with a weight of 2 and LSS-LRTA* for the sampling. The generated beliefs are very similar, with only minor differences for large heuristic values where fewer samples have been observed. This is somewhat expected; while the two algorithm expand different sets of states due to their different expansion strategies, the underlying instances are the same, and the algorithms will find similar solutions. We conjecture that it is unlikely for them to observe a very different heuristic behavior over their respective sets of states. For the experiments in the later section, we use weighted A* with a weight of 2 to sample the training states.

A further key concern for any data-driven algorithm is how to determine the number of examples necessary to ensure that the data observed in those examples are representative for the general case. This is of special importance with our setup, since we only consider a subset of all states for any given training instance. Figure 3 shows a comparison of the resulting beliefs on Blocksworld when only using half of the available training instances as an empirical indication whether our data is sufficient. As expected, the spread of h^* values is smaller when reducing the number of sample states. Overall however, the distributions have similar shapes.

Figure 4 shows an example of the generated data for Blocksworld and the unit-cost version of Transport to demonstrate the differences in heuristic behavior on these domains. The expected value increases roughly monotonically in both domains, but slightly faster in Transport, where the expected value makes a jump when going from $h = 2$ to $h = 3$. Furthermore, the variance of observed h^* values is greater in Transport. In Blocksworld, the training process encountered states with slightly larger heuristic values than those in Transport.

A similar comparison of the generated data for the classic search domains is shown in Figure 5. While the distributions are very smooth for the pancake puzzle and the racetrack domain, the expected value of the distributions on the 15-puzzle makes a large jump at $h = 4$. This shows the potential inaccuracy of the Manhattan distance heuristic in the 15-puzzle: there are states where the heuristic value is small, but an optimal solution still requires a significant number of moves.

Empirical Comparison

While Nancy is, to our knowledge, the first method for real-time heuristic graph search that bases its search strategy on

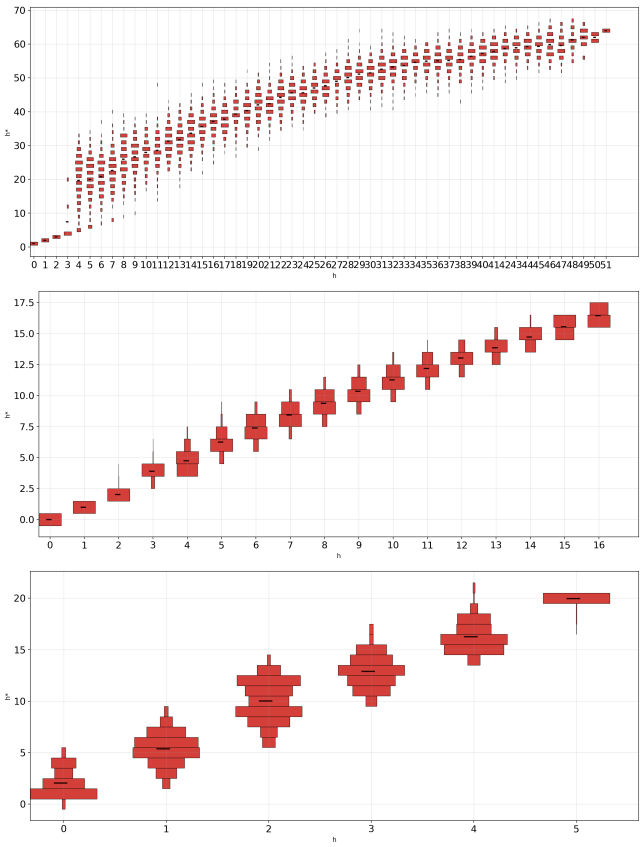


Figure 5: Beliefs gathered for the uniform-cost 15-puzzle (top), 16-pancake (middle), and Barto Racetrack (bottom).

belief distributions, there has been much previous work in the RL community on search methods that attempt to estimate and exploit value uncertainty. However, most RL domains feature stochastic actions, while in our setting the uncertainty stems entirely from the bounded computation of the agent. We consider two prominent approaches: interval estimation and Monte-Carlo tree search. In each case, we adapt the previous work to our setting and empirically compare it to Nancy.

Domains

We show experiments on the three classic search domains. First is the classic 100 15-puzzle instances published by Korf (1985). We test two variants: uniform-cost, in which every actions costs one, and heavy, in which the action cost is equal to the label of the moved tile. We use the Manhattan distance heuristic for all the real-time search algorithms.

Second is the pancake problem (Kleitman et al. 1975; Gates and Papadimitriou 1979; Heydari and Sudborough 1997) where the objective is to sort a sequence of pancakes through a minimal number of prefix reversals. We use the GAP heuristic (Helmert 2010) for for all the real-time search algorithms. We test three size of pancakes: 16, 32, and 40. One hundred instances of each size were tested per experiment.

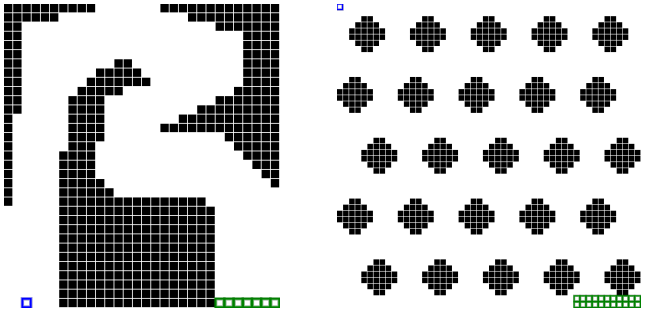


Figure 6: Barto (left) and uniform (right) Racetrack variants.

Third is the Racetrack domain which is very similar to the grid pathfinding problem, but features additional actions and inertia. It is reminiscent of autonomous driving and is a variant of the popular Racetrack problem (Barto, Bradtke, and Singh 1995). Figure 6 shows the two maps used in our experiments. The track shown on the left was created by Hansen and Zilberstein (2001), and the cluttered track on the right by Cserna et al. (2018). The agent moves in a grid attempting to reach one of a set of goal locations while avoiding static obstacles. Each action modifies the acceleration of the agent by -1, 0, or 1 in both the horizontal and vertical directions, making for a total of 9 distinct actions. There is no limit on the agent's speed. The system state includes the agent's location and velocity. The objective is to minimize the number of time steps until a goal cell is reached. The heuristic function is the maximum, either horizontally or vertically, of the distance to the goal divided by an estimate of the maximum achievable velocity in that dimension. This is admissible. For each of the two maps, we created 25 instances with starting positions chosen randomly among those cells that were at least 90% of the maximum distance from a goal.

Interval Estimation

Interval Estimation (IE) (Kaelbling 1993; Strehl and Littman 2004) applies the philosophy of 'optimism in the face of uncertainty' to the problem of action selection in a two-armed bandit problem. While the method has previously been investigated primarily for use in MDPs and RL, we adapt it here for use in real-time deterministic planning. When deciding under which TLA the next node should be expanded, given the belief distributions of all the TLAs, IE chooses the TLA with the lowest lower bound on the 95% confidence interval of the backed up cost-to-goal estimate instead of performing a computationally complex risk analysis. To adapt IE to the real-time search setting, we need to augment it with a mechanism for heuristic value updates. For each node in the LSS, we back up the belief from the child with the lowest lower confidence bound. Thus, the best frontier distributions under each TLA are eventually backed up to the TLA. The interval estimation approach naturally practices the spirit of uncertainty-based exploration in a very computationally efficient way.

Figure 7 shows an experimental comparison of Nancy and LSS-LRTA* to IE (and also to a Monte-Carlo tree search

approach, which we discuss in the next subsection). IE performs well in our experiments, and closely matches the performance of Nancy. While Nancy has a slight advantage on the sliding tile and pancake puzzles, IE works perhaps marginally better on Racetrack. Overall, IE is very competitive, and yet simple to implement and computationally efficient.

Monte Carlo Tree Search

Monte-Carlo tree search (Browne et al. 2012) approaches such as UCT (Kocsis and Szepesvári 2006) are popular for solving stochastic problems such as with MDPs (Keller and Eyerich 2012) and POMDPs (Silver and Veness 2010). Recently, Schulte and Keller (2014) adopted this to deterministic planning problems as trial-based heuristic tree search (THTS). Like Nancy, THTS also takes the uncertainty about the heuristic into consideration (though more implicitly). However, THTS was only described as an offline search framework. We adapted it to the real-time setting based on LSS-LRTA*. We replace the A* lookahead in the expansion phase with the THTS algorithm. More precisely, we use the THTS-WA* instantiation of the THTS framework since this variation had the best results for the base setting in the original paper (without the preferred operators enhancement that is specific to domain-independent planning). In the learning phase, we also use a reversed Dijkstra's algorithm to update the heuristic values, working from the uninitialized frontier tree nodes inwards. In the decision making phase, we use the identical strategy as LSS-LRTA* and move towards the node with minimal f-value.

Consider again Figure 7. The real-time variant of THTS performs poorly on the 15-puzzle (in particular the heavy-tile version). On the pancake puzzle, it again beaten by most of the other considered approaches, but it does beat LSS-LRTA* on the larger instances with small lookahead. On the Racetrack domain on the other hand, it outperforms all other algorithms, for all considered lookahead values. Overall, the uncertainty-aware algorithms (Nancy, IE, and THTS) surpass the conventional LSS-LRTA* baseline, with Nancy and IE being the most robust.

Discussion

Viewed broadly, reinforcement learning considers how action selection should be informed by data that is gathered during execution. This is exactly what heuristic search strategies do. The states and costs computed during lookahead are data that inform action selection. As Nancy shows, a heuristic search can use this data in two ways. Clearly, the computed lookahead states in a real-time search setting inform the selection of the action for the agent to execute. But more broadly, the problem of designing any heuristic search strategy, even an off-line one, is an RL problem at the computational level, in that the search space computed so far can inform the choice of which nodes should be expanded next. For an optimal off-line search like A*, all nodes whose f values are less than the optimal solution cost C^* must be expanded, so there is little flexibility and less need for a sophisticated expansion strategy. But, in contrast, the tight resource limitations of real-time search strongly highlight the need for care in selecting even computational expansion actions.

This metareasoning problem of heuristic search can be conceptualized as a POMDP in which each state represents an entire state space graph, complete with costs on every arc and h values at every vertex. (To avoid confusion in this discussion, we will use the term 'vertex' for a node in the state space graph and the term 'state' for a state in the POMDP.) The search does not know which exact state space graph it is dealing with, thus its situation is captured by a belief distribution over states. Every node expansion gathers data that rule out those state space graphs that are inconsistent with the computed successor nodes, action costs, and h values. A goal state is a belief that has positive support only on state spaces that all share the same path from the initial vertex to a goal vertex, providing a solution to the original problem but potentially harboring remaining uncertainty about the unseen portions of the graph. Solving this POMDP for a policy that, for example, minimizes expected solution length would give a heuristic search strategy that finds a solution as quickly as possible by minimizing the expected number of expansions.

Approaching such a problem in practice depends crucially on exploiting structure in the h values, the arc costs, and the distance to the nearest goal. The data-driven version of Nancy highlights this. However, while Nancy does try to predict how its beliefs will change with additional search under frontier nodes, note that it is myopic and does not really plan at the metalevel. For example, if nodes with $h = 3$ were to be predicted to have higher expected distance to goal than nodes with $h = 4$, due perhaps to a misleading heuristic, data-driven Nancy will not realize that it must nonetheless ignore the tempting $h = 4$ nodes from time to time and try expanding some $h = 3$ nodes in order to eventually reach some $h = 2$ nodes and eventually the goal.

In related work, Lin et al. (2015) formalize the problem of metareasoning for an MDP and find that its computational complexity is polynomial in the time required to solve the MDP itself. This indicates the impracticality of optimal metareasoning, motivating approximations such as those used by Nancy.

Conclusion

Inspired by distributional methods from RL, the Nancy framework reconsiders real-time search as a decision making process where limited information creates uncertainty. Nancy models this uncertainty using belief distributions and reasons about it to guide the search. In this paper, we presented further experimental results regarding this approach. First, we presented detailed results regarding heuristic error data, which Nancy can use as the basis for its beliefs. Second, we reported an experimental comparison with approaches from RL that exploit value uncertainty, such as Monte Carlo tree search and Kaelbling's interval estimation. We find that our approach, Nancy, generally outperforms previous methods, particularly on more difficult problems. This work illustrates how distributional methods from RL can be adapted to deterministic planning settings, and how

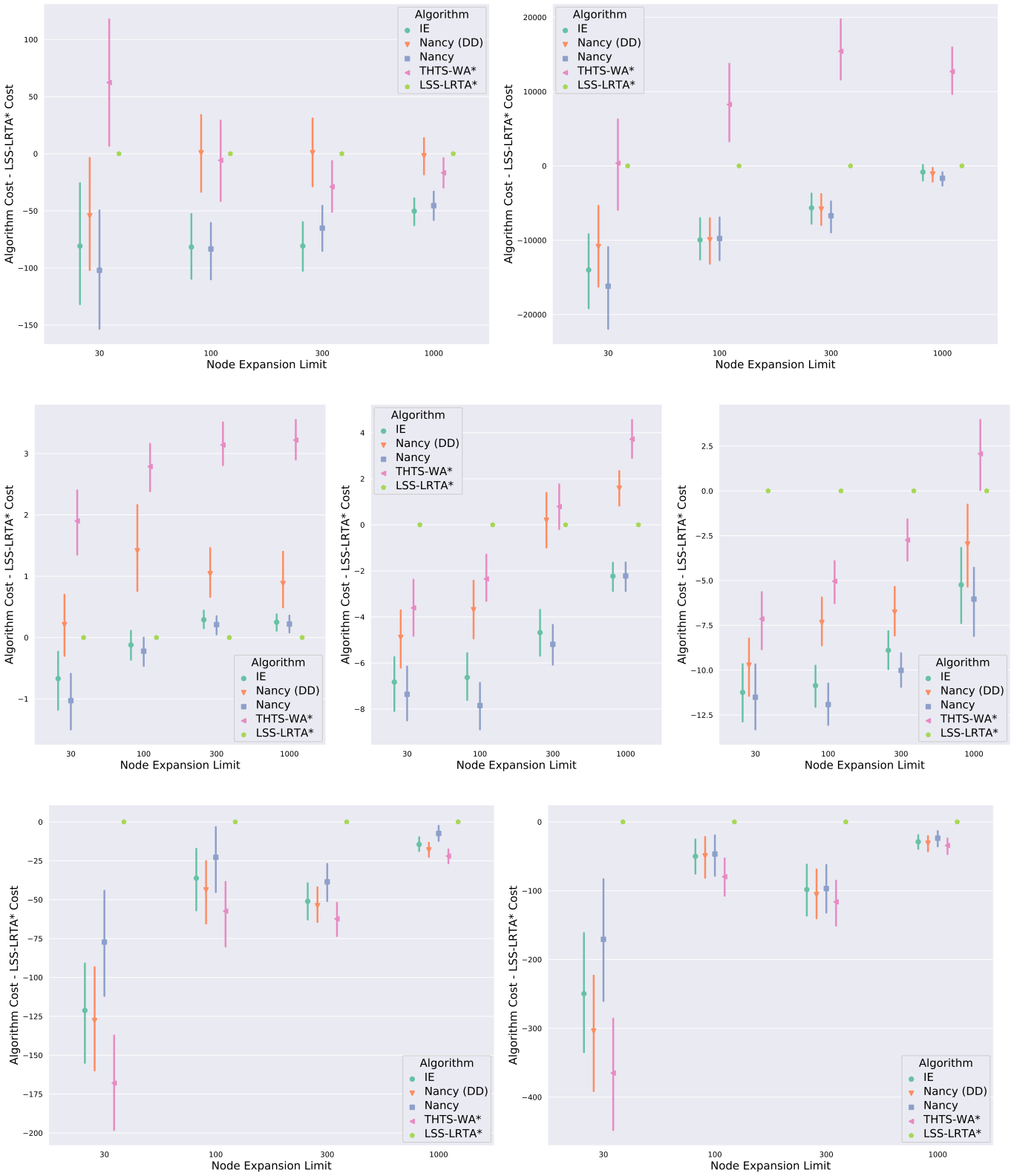


Figure 7: Comparison to IE and THTS. Top: 15-puzzle (left: unit, right: heavy); Middle: pancake (16, 32, and 40); Bottom: Racetrack (left: Barto, right: uniform).

deterministic planning can provide useful testbeds for exploring methods that metareason about uncertainty during planning.

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72(1):81–138.

Bellemare, M. G.; Dabney, W.; and Munos, R. 2017. A distributional perspective on reinforcement learning. In *Proceedings of ICML-17*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Cserna, B.; Doyle, W. J.; Ramsdell, J. S.; and Ruml, W. 2018. Avoiding dead ends in real-time heuristic search. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Dearden, R.; Friedman, N.; and Russell, S. 1998. Bayesian Q-learning. In *Proceedings of AAAI-98*, 761–768.

Feldman, Z., and Domshlak, C. 2014. Simple regret optimization in online planning for markov decision processes. *Journal of Artificial Intelligence Research* 51:165–205.

Fickert, M.; Gu, T.; Staut, L.; Ruml, W.; Hoffmann, J.; and Patrik, M. 2020. Beliefs we can believe in: Replacing assumptions with data in real-time search. In *AAAI-20*.

Frazier, P. I.; Powell, W. B.; and Dayanik, S. 2008. A knowledge-gradient policy for sequential information collection. *SIAM J. on Control and Opt.* 47(5):2410–2439.

Gates, W. H., and Papadimitriou, C. H. 1979. Bounds for sorting by prefix reversal. *Discrete mathematics* 27(1):47–57.

Hansen, E. A., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1-2):35–62.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Sys. Sci. and Cybernetics* SSC-4(2):100–107.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Proceedings of SoCS-10*.

Heydari, M. H., and Sudborough, I. H. 1997. On the diameter of the pancake network. *J. Algorithms* 25(1):67–94.

Kaelbling, L. P. 1993. *Learning in Embedded Systems*. MIT Press.

Keller, T., and Eyerich, P. 2012. Prost: Probabilistic planning based on uct. In *Proceedings of ICAPS-12*.

Keller, T., and Helmert, M. 2013. Trial-based heuristic tree search for finite horizon MDPs. In *Proceedings ICAPS-13*.

Kleitman, D.; Kramer, E.; Conway, J.; Bell, S.; and Dweighter, H. 1975. Elementary problems: E2564-e2569. *The American Mathematical Monthly* 82(10):1009–1010.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of ECML-06*, 282–293.

Koenig, S., and Sun, X. 2008. Comparing real-time and incremental heuristic search for real-time situated agents. *J. Auton. Agents and Multi-Agent Sys.* 18(3):313—341.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *IJCAI-85*, 1034–1036.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Lieck, R., and Toussaint, M. 2017. Active tree search. In *ICAPS Workshop on Planning, Search, and Optimization*.

Lin, C. H.; Kolobov, A.; Kamar, E.; and Horvitz, E. 2015. Metareasoning for planning under uncertainty. In *Proceedings of IJCAI-15*.

McMahan, H. B.; Likhachev, M.; and Gordon, G. J. 2005. Bounded real-time dynamic programming: Rtdp with monotone upper bounds and performance guarantees. In *Proceedings of ICML-05*, 569–576.

Mitchell, A.; Ruml, W.; Spaniol, F.; Hoffmann, J.; and Petrik, M. 2019. Real-time planning as decision-making under uncertainty. In *Proceedings of AAAI-19*.

Mutchler, D. 1986. Optimal allocation of very limited search resources. In *Proceedings of AAAI-86*.

O'Ceallaigh, D., and Ruml, W. 2015. Metareasoning in real-time heuristic search. In *SoCS-15*.

Pemberton, J. C., and Korf, R. E. 1994. Incremental search algorithms for real-time decision making. In *AIPS-94*.

Pemberton, J. C. 1995. k-best: A new method for real-time decision making. In *IJCAI-95*.

Russell, S., and Wefald, E. 1991. *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Sanner, S.; Goetschalckx, R.; Driessens, K.; and Shani, G. 2009. Bayesian real-time dynamic programming. In *Proceedings of IJCAI-09*.

Schulte, T., and Keller, T. 2014. Balancing exploration and exploitation in classical planning. In *SoCS-14*.

Silver, D., and Veness, J. 2010. Monte-carlo planning in large POMDPs. In *NIPS*, 2164–2172.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Strehl, A. L., and Littman, M. L. 2004. An empirical evaluation of interval estimation for markov decision processes. In *IEEE ICTAI-04*.

Thayer, J. T.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proceedings of ICAPS-11*.

Tolpin, D., and Shimony, S. E. 2012. MCTS based on simple regret. In *Proceedings of AAAI-12*.