30th International Conference on
Automated Planning and Scheduling

October 19–30, 2020, online

HSDIP 2020

Proceedings of the 12th Workshop on

Heuristics and Search for Domain-independent Planning (HSDIP)

Edited by:

Alberto Camacho, Salomé Eriksson, Daniel Fiešer, Guillem Francès, Florian Geißer, Patrik Haslum, Jendrik Seipp, Silvan Sievers, David Speck, Álvaro Torralba

Organization

Alberto Camacho
Google and University of Toronto, Canada

Salomé Eriksson
University of Basel, Switzerland

Daniel Fiešer
Czech Technical University, Czech Republic

Guillem Francès
Universitat Pompeu Fabra, Spain

Florian Geißer
The Australian National University, Australia

Patrik Haslum
The Australian National University, Australia

Jendrik Seipp
University of Basel, Switzerland

Silvan Sievers
University of Basel, Switzerland

David Speck
University of Freiburg, Germany

Álvaro Torralba
Aalborg University, Denmark

Foreword

Planning as heuristic search remains among the dominating approaches to many variations of domain-independent planning, including classical planning, temporal planning, planning under uncertainty and adversarial planning, for nearly two decades. The research on both heuristics and search techniqes is thriving, now more than ever, as evidenced by both the quality and the quantity of submissions on the topic to major AI conferences and workshops.

This workshop seeks to understand the underlying principles of current heuristics and search methods, their limitations, ways for overcoming those limitations, as well as the synergy between heuristics and search. To this end, this workshop intends to offer a discussion forum and a unique opportunity to showcase new and emerging ideas to leading researchers in the area. Past workshops have featured novel methods that have grown and formed indispensable lines of research.

This year is the 12th edition of the workshop series on Heuristics for Domain-independent Planning (HDIP), which was first held in 2007. HDIP was subsequently held in 2009 and 2011. With the fourth workshop in 2012, the organizers sought to recognize the role of search algorithms by acknowledging search in the name of the workshop, renaming it to the workshop on Heuristics and Search for Domain-independent Planning (HSDIP). The workshop continued flourishing under the new name and has become an annual event at ICAPS.

Alberto Camacho, Salomé Eriksson, Daniel Fiešer, Guillem Francès, Florian Geißer,
Patrik Haslum, Jendrik Seipp, Silvan Sievers, David Speck, Álvaro Torralba
October 2020

Contents

Learning Search-Space Specific Heuristics Using Neural Network

Liu Yu, Ryo Kuroiwa,[1] Alex Fukunaga,[2]
[1]Department of Mechanical and Industrial Engineering, University of Toronto
[2]Graduate School of Arts and Sciences, The University of Tokyo
liuyu.ai@outlook.com, mhgeoe@gmail.com, fukunaga@idea.c.u-tokyo.ac.jp

Abstract

We propose and evaluate a system which learns a neural-network heuristic function for forward search-based, satisficing classical planning. Our system learns distance-to-goal estimators from scratch, given a single PDDL training instance. Training data is generated by backward regression search or by backward search from given or guessed goal states. In domains such as the 24-puzzle where all instances share the same search space, such heuristics can also be reused across all instances in the domain. We show that this relatively simple system can perform surprisingly well, sometimes competitive with well-known domain-independent heuristics.

1 Introduction

State-space search using heuristic search algorithms such as GBFS is a state-of-the-art technique for satisficing, domain-independent planning. Search performance is largely determined by the heuristic evaluation function used to decide which state to expand next. Heuristic function effectiveness for domain-independent planning depends on the domain, as different heuristics represent different approaches to exploiting available information. Designing heuristics which work well across many domains is nontrivial, so learning-based approaches are an active area of research.

In one setting for learning search control knowledge for planning (exemplified by the Learning Track of the IPC), a set of training problem instances (and/or a problem instance generator) is given, and the task is to learn a *domain-specific* heuristic for that domain. Previous work on learning heuristics and other search control policies (e.g., selection among several search strategies) in this setting include (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009; de la Rosa et al. 2011; Garrett, Kaelbling, and Lozano-Pérez 2016; Sievers et al. 2019; Gomoluch, Alrajeh, and Russo 2019). Another type of setting seeks to learn domain-independent planning heuristics, which generalize not only to domains used during training, but also to unseen domains (Shen, Trevizan, and Thiébaux 2020; Gomoluch et al. 2017).

Inter-instance speedup learning, or "on-line learning", is a setting where only one problem instance (no training instances or problem generator) is given, and the task is to solve that instance as quickly as possible. Speedup learning within a single problem solving episode is worthwhile if the *total time spent by the solver (including learning)* is faster than the time required to solve the problem using other methods. Previous work on on-line learning for search-based planning includes learning decision rules for combining heuristics (Domshlak, Karpas, and Markovitch 2010) and macro operator learning (Coles and Smith 2007).

On-line learning can be used to learn an *instance-specific heuristic*. Previous work on instance-specific learning includes bootstrap heuristic learning (Arfaee, Zilles, and Holte 2011), as well as LHFCP, a single-instance neural network heuristic learning system (Geissman 2015). Instance-specific learning can be generalized to *single search space learning*, where many problem instances share a single search space. For example, all instances of the 15-puzzle domain share the same search space – different instances have different initial states, all on the same connected state space graph. Thus, a learned heuristic function which performs well for one instance of the 15-puzzle can be directly applied to other instances of the domain.

We propose and evaluate SING, a neural network-based instance-specific and single search space heuristic learning system for domain-independent, classical planning. SING is closely related to LHFCP, an approach to supervised learning of heuristics which generates training data using backward search (2015). Given a PDDL problem instance I, LHFCP learns a heuristic h_{nn} for I. To generate training data for h_{nn}, LHFCP performs a series of backward searches from a goal state of I to collect a set of states and their approximate distances from the goal. After training, h_{nn} is used as the heuristic function by GBFS to solve I. This does not require any additional training instances as input, nor pre-existing heuristics to bootstrap its performance. However, LHFCP performed comparably to blind search (Geissman 2015), so achieving competitive performance with this approach remained an open problem.

SING expands upon this basic approach in several ways: (1) improved backward search space using either (a) explicit search with inferred inverse operators or (b) regression, (2) depth-first search (vs. random walk), (3) boolean state representation, and (4) relative error loss function. We experimentally evaluate SING for learning domain-specific heuristics for domains where instances share a single state space, and show performance competitive with the Fast Forward heuristic (h_{ff}) (Hoffmann and Nebel 2001), and the land-

mark count heuristic (h_{lm}) (Hoffmann, Porteous, and Sebastia 2004) on several domains. We also evaluate SING as an instance-specific heuristic learner, and show that the learned heuristics consistently outperforms blind search on a broad range of standard IPC benchmark domains, and performs competitively on some domains, even when the learning times are accounted for within the time limit.

2 Preliminaries and Background

We consider domain-independent classical planning problems which can be defined as follows. A SAS+ planning task (Bäckström and Nebel 1995) is a 4-tuple, $\Pi = \langle V, O, I, G, \rangle$, where $V = x_1, ..., x_n$ is a set of state variables, each with an associated finite domain $Dom(x_i)$; A state s is a complete assignment of values to variables. $\mathcal{S}$ is the set of all states. O is a set of actions, where each action $a \in O$ is a tuple $(pre(a), \mathit{eff}(a))$, where $pre(a)$ and $\mathit{eff}(a)$ are sets of *partial* state variable assignments $x_i = v, v \in Dom(x_i)$; $I \in S$ is the initial state, and G is a partial assignment of state variables defining a goal condition ($s \in S$ is a goal state if $G \subset s$). A plan for Π is a sequence of applicable actions which when applied to I results in a state which satisfies all goal conditions. Search-based planners seek a path from the start state to a goal state using a search algorithm such as best-first search guided by a heuristic state evaluation function.

A natural approach to learn heuristic functions for search-based planning is a supervised learning framework consisting of the following stages: (1) **Training Sample Generation**: generate many state/distance pairs which will be used as training data. (2) **Training**: Train a heuristic function h which predicts distances from a given state to a goal. (3) **Search**: Use h as the heuristic evaluation function in a standard heuristic search algorithm such as GBFS. This paper focuses mostly on stage (1), training data generation.

Ferber et al. (2020) investigated an approach where the training data was generated using forward search from the start states of training instances. They perform random walks (200 steps) from the initial state, and from each step visited in the random walk, they perform a forward search (a "teacher search" using a heuristic such as h_{ff}) to a goal in order to find the distance to the goal. If the teacher search finds a path to the goal, the states on the path as well as the distance-to-goal for the states on the path are added to the training data. This approach can be practical for shared search space heuristic learning, where the costs of the teacher searches can be amortized among many instances on the same search space. However, this is not practical for satisficing, single instance heuristic learning where there is only one problem solving episode, as requiring forward search to the goal in order to gather training data obviates the need to learn a heuristic for that particular instance.

An alternative approach to generating training data uses backward search from the goal. A backward search starting at a goal state (provided in or guessed/derived from the problem specification) is performed, storing encountered states and their (estimated) distances from thoe goal as the training data. Arfaee, Zilles, and Holte (2011) used this approach in a bootstrap system for heuristic learning, which starts with a weak neural net heuristic h_0 and generates increasingly more powerful heuristics by using the current heuristic to solve problem instances, using states generated during the search as training data for the next heuristic improvement step. If h_0 is too weak to solve training set problems, they generate training data by random walks from the goal state to generate easy problem instances that can be solved by h_0. Lelis et al. (2016) proposed BiSS-h, an improvement which uses a solution cost estimator instead of search for training data generation. Arfaee et al. and Lelis et al. evaluated their work on domain-specific solvers for the 24-puzzle, pancake puzzle, and the Rubik's cube.

Geissmann (2015) investigated a backward-search approach to training data generation for domain-independent classical planning. His system, LHFCP, uses backward search to generate training data for learning a neural network heuristic function which estimates the distance from a state to a goal. To generate training data, LHFCP performs backward search (random walk) in an explicit search space. It generates the start state for backward search by generating a state which satisfies the goal conditions, with values unspecified by the goal condition filled in randomly. LHFCP relies on the operators in the original (forward) problem to perform backward search. Search using the heuristics learned by LHFCP across a wide range of IPC domains performed comparably to blind search (Geissman 2015). Geissmann also investigated a variation of LHFCP which applied BiSS-h to classical planning but reported poor results, attributed to difficulties in efficiently implementing BiSS for classical planning. Thus, a successful backward-search based approach to training data generation for domain-independent classical planning remained an open problem.

3 SING: An Improved, Backward-Search Based Heuristic Learning System

We describe **SI**ngle search space **N**eural heuristic **G**enerator (SING), a system which learns single-search space heuristics for domain-independent planning. SING learns a heuristic function $h_{nn}(s)$, which takes as input a vector representation of a state s, and returns a heuristic estimate of the distance from s to a closest goal. SING is implemented on top of the Fast Downward planner (FD) (Helmert 2006).

SING uses backward search to generate training data, similar to LHFCP, but incorporates several significant differences in the state representation, backward search space formulation, and backward search strategy. Below, we describe each of these in details:

3.1 State Representation

The input to h_{nn} is a vector representation of a state. LHFCP used a multivalued SAS+ vector representation of the state, which is a natural representation to use, as FD uses the SAS+ representation internally.

Another natural representation for the vector input to h_{nn} is based on the STRIPS propositional representation of the problem. A STRIPS planning task (Fikes and Nilsson 1971) is a 4-tuple, $\Pi = \langle F, I, G, A, \rangle$ where F is a set of propositional facts, $I \in 2^F$ is the initial state, $G \in 2^F$ is a set of

goal facts, and A is a set of actions. Each action $a \in A$ has preconditions $pre(a)$, add effects $add(a)$, and delete effects $del(a)$, which are sets of facts. A state $s \in 2^F$ is set of facts, and s is a goal state if $G \subseteq s$. Given a state $s \in 2^F$, a is applicable iff $pre(s) \subseteq s$. After applying a in s, s transitions to $s \cup add(a) \setminus del(a)$. A plan for Π is a sequence of applicable actions which make I transition to a goal state.

The STRIPS representation corresponds directly to the classical planning subset of the standard PDDL domain description language, as PDDL uses boolean facts to represent the world state. In the SAS+ representation used by FD, each possible value of a variable represents a mutually exclusive set of facts in the underlying propositional problem. Each variable-value pair in FD represents a fact, negation of a fact, or negation of all facts represented by other values in the variable. Preconditions and effects of actions are also represented as the set of variable-value pairs. Since the variable/value naming conventions used in the SAS+ generated by the FD PDDL-to-SAS+ translator, conversion between the SAS+ finite-domain representation and the STRIPS propositional state representation is easy. Thus, h_{nn} can use either the boolean (STRIPS) or multivalued (SAS+) state vector representation as input during training and search.

Since each input bit corresponds to a fact in the boolean encoding, it may enable a more accurate h_{nn} state evaluation function to be learned than the SAS+ multi-valued encoding. On the other hand, SAS+ encodings are more compact, which can significantly reduce the dimensionality of the state representation, which can result in faster NN evaluation, speeding up the search process. Thus, the choice of state vector representation poses a tradeoff between h_{nn} evaluation accuracy and h_{nn} evaluation speed, and SING can use either the multivalued SAS+ vector representation or the STRIPS boolean vector representation.

3.2 Search Space and Operators for Training Sample Generation

The task of training sample generation is to collect a training set $T = \{(s_1, e_1), ..., (s_r, e_r)\}$ a set of r states and their (estimated) distance to a goal. The basic idea is to repeatedly start at a goal g and generate a sequence of states heading away from it (using a directed search or random walk), adding such states to the training data.

In some search problems such as the sliding tiles puzzle, backward search is relatively straightforward as the goal state is given explicitly as input to the problem, and the operators available for the forward problem are sufficient to solve the backward problem.

In domain-independent planning, backward search based training sample generation poses several issues. First, a goal condition, possibly satisfied by many goal states, is given instead of an explicit, unique goal state, so in general, it is not possible to simply "search backward from the goal state". Second, in general, the operators for the forward problem are not sufficient for backward search. LHFCP generates a start state for backward search by generating a state which satisfies the goal conditions, with values unspecified by the goal condition filled in randomly. It relies on the operators in the original (forward) problem to perform backward search.

SING incorporates two approaches to backward search for training sample generation: (1) backward explicit search using derived inverse operators, and (2) regression.

Explicit Backward Search with Derived Inverse Operators As in LHFCP, a candidate start state for backward search is generated by first generating a partial state which satisfies all conditions in the goal condition, and then randomly assigning values to variables whose values are unspecified in the goal condition. Such a randomly generated candidate start state s might be invalid and unexpandable, i.e., no backward operators (see below) can be applied to s. In that case, we simply generate another candidate state. This random initialization is performed for each backward sampling search.

For the search operators, one simple approach is to use the same set of actions as for forward search, as in LHFCP (Geissman 2015). However, this fails in domains where actions are not invertible such as `visitall`.

Thus, operators for the backward search must be derived from the forward operators. Since preconditions and effects are represented as a set of variable-value pairs in FD, one naive method to generate inverse actions is to swap values of variables which appear in both preconditions and effects. Other variable-value pairs in preconditions and effects are treated as preconditions in the inverse action, because they must hold after application of the action. However, the inverse action does not change values of variables which appear in the original effects but not in the original preconditions. To address this issue, we use information available in the STRIPS formulation of the problem (as explained in Section 3.1, conversion among the PDDL problem description, its STRIPS formulation and the SAS+ formulation used internally by Fast Downward is straightforward). For action a, we generate an inverse action a' such that $pre(a') = (pre(a) \cup add(a)) \setminus del(a)$, $add(a') = del(a)$, and $del(a') = add(a)$. We identify variable-value pairs which represent propositions as add effects, and pairs which represent negation of facts as delete effects.

Regression Another approach to backward search is regression. In backward search using regression, we use the modified SAS+ representation by Alcázar et al. (2013). An action a is applicable to a state s if $add(a) \cap s \neq \emptyset \wedge del(a) \cap s = \emptyset$. If an action a is applied to a state s, s transitions to a state $s' = (s \setminus add(a)) \cup pre(a)$. Normally, SAS+ variables represent mutex groups of the corresponding STRIPS propositions. In regression planning with SAS+, each variable has an additional, *undefined* value. The starting node in regression space is the goal state, where variables unspecified by the goal condition have *undefined* values. When an action a is applied to a state s, if a variable v is included in $add(a)$ but not in $pre(a)$, v is set to *undefined*.

When generating training data, a bit vector representation of states needs to be generated (Section 3.1). When converting the SAS+-based representation used by Fast Downward into a bit vector, unlike the other possible state values in the mutex group, undefined values are not explicitly represented

in the bit vector. For example, suppose a state variable x in regression search has 2 possible actual values, v_1 and v_2, as well as "undefined". In the bit vector representation output for use as training data, x is represented by 2-bits, where the first bit represents x_1, and the second bit represents x_2, and there is no explicit third bit for the undefined value.

Regression vs. explicit search spaces The choice of regression vs. explicit spaces depends on the domain. Although regression is in a sense the "correct" way to perform backward search, the backward branching factor in regression space is very large in many domains. On the other hand, while explicit backward spaces sometimes have much smaller branching factor than regression, goal generation has risks. First, goal generation might fail to find a goal. Also, generated goals might not be true goal states reachable from the start state, and states reachable from such incorrect states are also unreachable from the start state. Such cliques (unreachable from the start state) can cause backward search to yield few or no states for training data.

However, although the training data may include states which are unreachable from the start state, these may nevertheless be useful for learning an effective h_{nn} which evaluates "real" states during search, somewhat similar to how synthetic data generated by the adversary during training is useful for learning networks which correctly classify real data in GAN learning (Goodfellow et al. 2014).

3.3 Backward Search Strategy

Given a start state for backward search (corresponding to a goal in the forward search space), we seek a set of training states T which are relatively far from goal g but with a reasonable estimate of their distance from g for training h_{nn}. Breadth-first search (BFS) from g could be used to generate states T for which $c(s, g)$, the exact distances from $s \in S$ to g (assuming unit-cost domains) are known, but would limit the training data to states which are very close to g. We need a search algorithm which can go much further from g than BFS, and for which the number of steps in the (inverted) path from $s \in T$ to g is an approximation of $c(s, g)$.

One natural sampling/search strategy is random walk, as in LHFCP (Geissman 2015). The number of steps from g at which s is encountered is used as an estimate of the true distance from g to s. Although random walk is fast, distance estimates from random walk may be inaccurate if cycles are not detected. Loop detection can be implemented easily using a hash table, but in domains with many cycles, it can be difficult to sample nodes far from g if the random walk is restarted whenever a previously visited node is generated.

Therefore, we use depth-first search (DFS) to extend a path from g, using the depth at which s is encountered is used as an estimate of the true distance from g to s, and all generated states are added to T. Random tie-breaking among s is used to select the nodes among successors of s, $Succ(s)$, to expand. A hash table is used to prune duplicate nodes and prevent cycles. In domains with many cycles and dead ends, by backtracking (instead of restarting search) when a duplicate is detected, DFS can potentially sample more states which are further from g than random walk. The best choice of sampling search strategy depends on the domain. In some domains, DFS generates more accurate samples than random walk due to duplicate detection and backtracking, while in other domains DFS may incur large overheads due to backtracking and Random walk allows faster searches.

In the experiments below, during training data generation we perform $nsearches$ backward searches, stopping each search after $nsamples$ states are collected, i.e., $nsearches \times nsamples$ states are collected.

3.4 Neural Network Architecture

We use a standard feedforward network for h_{nn}, using the ReLU activation function. Each layer is fully connected to the next layer. The input layer takes the state vector representing a state s as input. As discussed in Section 3.1, the state vector is either a boolean vector for the STRIPS representation of the problem instance, or a multivalued vector for the SAS+ representation of the instance, so the number of inputs is the same as the length of the state vector ($|F|$ for STRIPS propositional representation, $|V|$ for SAS+ multivalued representation). The output layer is a single node which returns $h_{nn}(s)$, the heuristic evaluation value of state s. Since h_{nn} will be called many times as the heuristic evaluation function for best-first search, a small network enabling fast evaluation is desirable.

PyTorch 1.2.0 is used for training h_{nn}, but for search, we use the Microsoft ONNX Runtime 0.4.0 to evaluate h_{nn}. Both training and search use a single CPU core. Due to the simple network architecture as well as accelerated evaluation using the ONNX Runtime, h_{nn} can be evaluated relatively quickly, significantly faster than h_{ff} on most IPC domains, (see node expansion rates in Table 2).

3.5 Loss Function

Previous work on learning neural nets for classical planning used the standard Mean Square Error (MSE) regression loss function (Geissman 2015; Ferber, Helmert, and Hoffmann 2020). Instead of MSE, we use a prediction *relative error* sum loss function, $f_{loss} = \sum_i abs(\hat{y}_i - y_i)/(y_i + 1)$, which is the sum of the *relative* error of the predicted ($\hat{y}_i$) values compared to the training data (y_i).

4 Evaluation: Domain-Specific Heuristic Learning on Shared Search Spaces

In domains where multiple instances share the same space, it is possible to learn reusable h_{nn} networks that can be used across many instances, so the cost of learning a heuristic can be amortized across instances. For example, all instances of the N-puzzle (for a particular value of N) share the same search space.

We evaluated SING as a shared search space, single model learner on the following PDDL domains:

- `24-puzzle`: PDDL encodings of the standard 50-instance benchmark set from (Korf and Felner 2002)
- `35-puzzle`: 50 randomly generated instances

name	state vector	backward space	rev. search	inversion	NN # of hidden	NN nodes hidden	samples #
C2	boolean	regression	DFS	yes	1	16	10^5
C3	SAS+	explicit	rand. walk	yes	1	16	10^5
C4	boolean	explicit	DFS	yes	4	64	10^5
C5	boolean	explicit	DFS	yes	1	16	4×10^5
SING/L	SAS+	explicit	rand. walk	no	1	16	10^5

Table 1: SING configurations used in experiments. "state vector": vector representation of states. "backward space": search space for training data generation backward search. "rev. search" : search strategy for Training data generation backward search. "NN # of hidden": # of hidden nodes in h_{nn}. "NN nodes hidden": # of nodes per hidden layer. "samples #" : # of sample states collected in the training data collection phase using the sampling search. C2, C3 and C4 perform 500 searches, with a limit of 200 samples/search (10^5 samples). C5 performs 800 searches, with a limit of 500 samples/search (4×10^5 samples).

- `blocks25`: 100 `blocks` instances with 25 blocks generated using the generator from (Hoffmann 2002)
- `pancake`: 100 randomly generated instances with 14 pancakes.

For each domain above, we ran the learning phase (training data generation and h_{nn} training) *once* to learn a heuristic h_{nn} for the domain. For `24-puzzle`, we used the C4 configuration (Table 1 shows configuration details), and training data generation took 7 seconds and training took 61 seconds. For `blocks25`, we used the C5 configuration, training data generation took 502 seconds and training took 228 seconds. For `pancake`, we used the C4 configuration, training data generation took 21 seconds and training took 377 seconds.

Note that for these 3 domains, we tried several SING configurations (i.e., manual tuning) and report the results for the best configuration. We are currently investigating automated tuning (hyperparameter optimization) to optimize the best configuration for a given domain.

Table 2 and Figures 1-2 compare the coverage, node expansions, and runtime (on solved instances) of GBFS using h_{nn}, h_{ff}, h_{lm} with a 30 min time limit per instance and 8GB RAM limit using an Intel(R) Xeon(R) CPU E5-2680 v2.

h_{nn} had or tied for the highest coverage on all 4 domains. On `blocks25` and `pancake`, h_{nn} had the highest coverage. On `24-puzzle`, `35-puzzle` and `pancake`, h_{nn} had the lowest median run time. Thus, h_{nn} achieved competitive performance on all of these domains compared to both h_{ff} and h_{lm} in this shared search space evaluation setting. Note that while h_{nn} and h_{ff} expanded a comparable number of nodes, h_{nn} had a significantly higher median node expansion rate than h_{ff} resulting in faster runtimes.

Figure 3 compares heuristic accuracy (h-value minus true distance) for a set of 4400 states for h_{nn}, h_{ff}, h_{lm}, and h_{gc} (goal count). For states with true distance $\leq$ 30 from the goal state, h_{nn} is fairly accurate. This accuracy and the fast evaluation speed due to the simple neural network enables efficient, fast search.

5 Evaluation: Instance-Specific Learning

In Section 4, we evaluated SING for learning domain-specific heuristics which could be reused on many instances sharing the same search space, so the evaluation focused on search time, assuming that the time spent learning h_{nn} can be amortized across multiple instances.

Next, we evaluate SING as an instance-specific learner in an IPC Satisficing track setting, where SING is given 30 minutes total for all phases, including learning (including training data collection and training) and search. Each run of SING starts from scratch – *nothing is reused across instances, learning costs are not amortized, and the heuristic is learned specifically for solving a given instance once*.

We evaluate SING on a large set of standard benchmarks from the IPC, all with unit-cost actions. All runs were given a total 30 minutes for time limit both learning and search (i.e., includes training data collection, training, and search using h_{nn}) and 8GB RAM per instance. We evaluated the SING configurations in Table 1. As baselines for comparison, we also evaluated blind search, the goal count heuristic (h_{gc}), the Fast Forward heuristic (h_{ff}) (Hoffmann and Nebel 2001), and the landmark count heuristic (h_{lm}) (Hoffmann, Porteous, and Sebastia 2004). As an additional baseline we also evaluate SING/L, a configuration of SING which is very similar to LHFCP (Geissman 2015) (see Table 1. This configuration is the same as C3, except that instead of the derived inverse operators (Section 3.2), SING/L uses only the actions available in the forward model.

Table 3 shows the coverage results (# of instances solved). SING configurations C2, C3, C4, C5 significantly outperform blind search, showing that SING successfully learned some useful heuristic information.

The SING/L (LHFCP) configuration performed comparably to blind search, consistent with the results in (Geissman 2015). Configuration C3, which differs from SING/L only in that action inversion is used, has much higher coverage than SING/L, showing the effectiveness of action inversion.

C2 outperforms h_{ff} on 5 domains and outperforms h_{lm} on 2 domains. C3 outperforms h_{ff} on 5 domains and h_{lm} on 1 domains. C4 outperforms h_{ff} on 4 domains, and C5 outperforms h_{ff} on 3 domains. Thus although none of the SING configurations are competitive with h_{ff} and h_{lm} with respect to overall coverage, these results indicate that there are some domains where competitive performance can be obtained with a 30 minute limit, including the time to learn an instance-specific heuristic function entirely from scratch without a teacher.

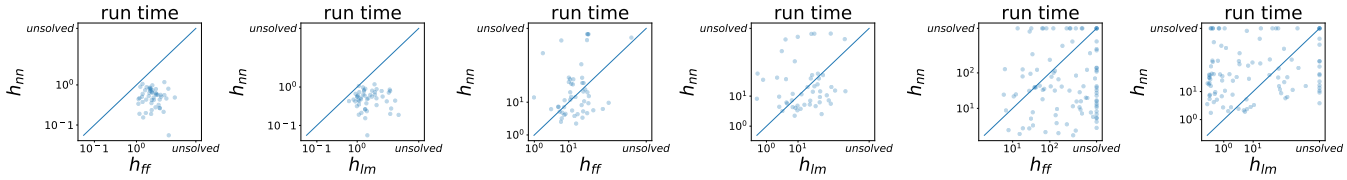
6 Ablation Study

To understand the relative impact of each of the new components of SING compared to LHFCP, we performed an ablation study comparing the following configurations:

(1) C5': Configuration C5 (Table 1) with fewer training samples (100k instead of 400k), (2) C5'/rw : same as C5', except using random walk instead of DFS, (3) C5'/sas : same as C5', except using SAS+ instead of boolean state representation, (4) C5'/reg : same as C5', except using regression instead of explicit search state, (5) C5'/orig : same as C5',

	coverage rate			median #expansions			median #exp. per second			median runtime		
	h_{nn}	h_{ff}	h_{lm}	h_{nn}	h_{ff}	h_{lm}	h_{nn}	h_{ff}	h_{lm}	h_{nn}	h_{ff}	h_{lm}
24-puzzle	**100.0**	**100.0**	**100.0**	**5,514**	9,232	67,859	10,649	3,862	**39,633**	**0.52**	2.31	1.59
35-puzzle	**100.0**	**100.0**	**100.0**	122,463	**57,045**	1,650,552	9,313	3,749	**74,487**	**12.95**	15.20	21.86
blocks	**84.0**	73.0	83.0	353,856	332,974	**33,658**	14,926	2,830	**24,054**	26.07	126.14	**1.56**
pancake	**100.0**	48.0	**100.0**	**74,873**	324,925	1,620,030	25,261	912	**134,248**	**2.93**	347.55	10.60
Average	**96.0**	80.2	95.8	**139,177**	181,044	843,025	15,038	2,838	**68,106**	10.61	122.80	**8.90**

Table 2: Domain-specific heuristics: Reusing a single learned model across many instances of the same shared search space domain. The (sampling, training) times were (28s, 210s) for `24-puzzle`, (276s, 764s) for `35-puzzle`, (502s, 228s) for `blocks25`, and (21s, 377s) for `pancake`.



(a) 24-puzzle (50 instances) (b) 35-puzzle (50 instances) (c) blocks25 (25 blocks, 100 instances)

Figure 1: Runtime (seconds) for `24-puzzle`, `35-puzzle`, and `blocks25`. h_{nn} vs. h_{ff} and h_{lm}.

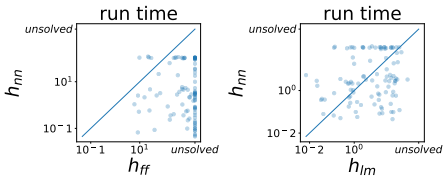


Figure 2: Runtime (seconds) for `pancake` (14 pancakes, 100 instances). h_{nn} vs. h_{ff} and h_{lm}.

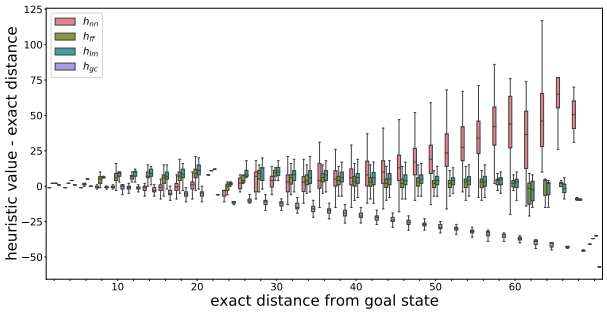


Figure 3: `24-puzzle` Heuristic accuracy: h_{nn}, h_{gc}, h_{ff}, h_{lm}

except using original operators only (no action inversion), and (6) C5'/mse : same as C5', except using MSE instead of the relative error sum loss function for NN training.

All configurations were run with a 30min, 2GB limit on the same IPC instances used in the above experiment. The coverages of the configurations were 604 for C5', 563 for C5'/rw, 481 for C5'/sas, 651 for C5'/reg, 420 for C5'/orig, and 559 for C5'/mse. This shows that the use of DFS in backward search, the use of boolean state representation, the use of action inversion, and the use of relative error sum loss function all have a significant positive impact on performance.

On the other hand, the effect of using regression vs explicit state search for the backward search during training data generation is highly domain-dependent, with regression performing better on some domains and explicit search on others, as can be seen by comparing configurations C2 (which is the same as C5'/reg) vs. C5 in Table 3.

7 Related Work

A broad survey of learning for domain-independent planning is (Celorrio et al. 2012). Satzger and Kramer (2013) developed a neural network based, domain-specific heuristic for classical planning. They used random problem generators to create instances for training the neural network. Their training process also relies on the use of an oracle (the FD planner with an admissible heuristic) to provide true distance from a state to a goal.

Shen et al. (2020) proposed an approach to learning domain-independent (as well as domain-dependent) heuristics using Hypergraph Networks. They showed that it was possible to successfully learn domain-independent heuristics which performed well even on domains which were not in the training data. As this approach uses a hypergraph based on the delete relaxation of the original planning instance, it is quite different from the minimalist approach taken in SING, which does not use any such derived features and uses only the raw state vector. The training data generation method is forward search based, similar to the forward approach of Ferber et al. described in Section 2 (Ferber, Helmert, and Hoffmann 2020). In addition, while their work focuses on generalization capability and search efficiency (node expansions) across domains, with runtime competitiveness left as future work, our work seeks to achieve runtime competitiveness using a simple NN architecture.

Random-walk sampling of the search space of determin-

istic planning problems for the purpose of learning a control policy for a reactive agent was proposed in (Fern, Yoon, and Givan 2004). This differs from SING in that SING learns a heuristic function which estimates distances to a goal state and and guide search (GBFS), instead of a reactive policy.

There is also a rapidly growing body of work on learning neural network based policies for probabilistic domains (c.f., (Toyer et al. 2018; Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Bajpai, Garg, and Mausam 2018; Garg, Bajpai, and Mausam 2019)), which is also related to learning heuristic evaluation functions for deterministic domains.

8 Conclusion

We investigated a supervised learning approach to learning a heuristic evaluation for search-based, domain-independent classical planning, where the training data is generated using backward search. Although LHFCP, a previous system, followed the same basic approach, it was performed comparably to blind search. SING pushes this approach much further using (1) backward search for training data generation using regression, as well as derived inverse operator for explicit space search, (2) DFS-based backward search for training data generation, (3) a propositional input vector representation, and (4) a relative error loss function.

We showed that SING can achieve performance competitive with h_{ff} and h_{lm} on several domains, both in shared search space scenarios where heuristics can be reused across domains, as well as single-instance learning where both learning and search using the learned heuristic must be performed within a given time limit.

SING is a relatively simple, minimalist system. SING uses *only* a PDDL description of a single problem instance as input. No additional problem generators or training instances are used. Learning is from scratch, and unlike the forward search based training data generation approach investigated by (Ferber, Helmert, and Hoffmann 2020), SING does not use any standard heuristics during training data generation. It uses a very simple feedforward neural network architecture, with no feature engineering. The only "features" used by SING are the raw state vectors. SING does not exploit any structures used by standard classical planning heuristics such as delete relaxations and causal graphs in either the learning or the search phases. Previous work used features derived/extracted from human-developed heuristics such as h_{ff} and explored how learning could be used to exploit such features in new ways (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009; de la Rosa et al. 2011; Garrett, Kaelbling, and Lozano-Pérez 2016; Shen, Trevizan, and Thiébaux 2020). By pushing the performance envelope for a more minimal approach our results provide a baseline for future work on heuristic learning using more sophisticated features and methods.

As discussed in Section 3.2, explicit backward search (as opposed to regression) for training data generation can generate states which are not reachable from the start state. Nevertheless, our results show that SING configurations which use explicit backward search perform quite well on some domains. In future work, we will investigate in detail how unreachable states in the training data affect the quality of the learned heuristic and the performance of the (forward) search using the learned heuristic.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting Regression in Planning. In *Proc. IJCAI*, 2254–2260.

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

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence* 11: 625–656.

Bajpai, A. N.; Garg, S.; and Mausam. 2018. Transfer of Deep Reactive Policies for MDP Planning. In *Proc. ICAPS*, 10988–10998.

Celorrio, S. J.; de la Rosa, T.; Fernández, S.; Fernández-Rebollo, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *Knowledge Eng. Review* 27(4): 433–467.

Coles, A.; and Smith, A. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *J. Artif. Intell. Res.* 28: 119–156. doi:10.1613/jair.2077.

de la Rosa, T.; Celorrio, S. J.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up Heuristic Planning with Relational Decision Trees. *J. Artif. Intell. Res.* 40: 767–813.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To Max or Not to Max: Online Learning for Speeding Up Optimal Planning. In *Proc. AAAI*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of the Hyperparameter Space. In *Proc. ECAI*.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning Domain-Specific Control Knowledge from Random Walks. In *Proc. ICAPS*, 191–199.

Fikes, R.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.* 2(3/4): 189–208.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proc. ICAPS*, 631–636.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. *CoRR* abs/1608.01302.

Geissman, C. 2015. *Learning Heuristic Functions in Classical Planning*. Master's thesis, University of Basel.

Gomoluch, P.; Alrajeh, D.; and Russo, A. 2019. Learning Classical Planning Strategies with Policy Gradient. In *Proc. ICAPS*, 637–645.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2017. Towards learning domain-independent planning heuristics. *CoRR* abs/1707.06895.

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 Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Proc. NIPS*, 2672–2680.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proc. ICAPS*, 408–416.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR* 26(1): 191–246. ISSN 1076-9757.

Hoffmann, J. 2002. FF Domain Collection. https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res.(JAIR)* 14: 253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered Landmarks in Planning. *J. Artif. Intell. Res.* 22: 215–278. doi:10.1613/jair.1492.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *Proc. ICAPS*, 422–430.

Korf, R. E.; and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2): 9–22.

Lelis, L. H. S.; Stern, R.; Arfaee, S. J.; Zilles, S.; Felner, A.; and Holte, R. C. 2016. Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces. *Artif. Intell.* 230: 51–73.

Satzger, B.; and Kramer, O. 2013. Goal distance estimation for automated planning using neural networks and support vector machines. *Natural Computing* 12(1): 87–100.

Shen, W.; Trevizan, F. W.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks 574–584.

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*, 7715–7723.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies With Deep Learning. In *Proc. AAAI*, 6294–6301.

Xu, Y.; Fern, A.; and Yoon, S. W. 2009. Learning Linear Ranking Functions for Beam Search with Application to Planning. *J. Mach. Learn. Res.* 10: 1571–1610.

Yoon, S. W.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *J. Mach. Learn. Res.* 9: 683–718.

	blind	gc	hff	hlm	SING/L (lhfcp)	c2	c3	c4	c5
agricola	0	0	10	**16**	1	13	12	11	9
airport	23	35	**36**	33	1	21	13	14	17
barman	0	0	12	**20**	0	0	0	0	0
blocks	18	**35**	**35**	**35**	21	33	27	**35**	**35**
childsnack	0	0	**1**	0	0	0	0	0	0
data-network	0	1	**5**	2	0	0	0	0	0
depot	4	14	**18**	**18**	6	5	5	15	13
driverlog	7	**19**	18	18	8	12	13	15	14
elevators	0	5	**20**	7	0	0	0	0	0
floortile	0	0	**2**	1	0	0	0	0	0
freecell	20	46	79	**80**	18	**80**	23	61	57
ged	0	**20**	**20**	**20**	0	**20**	0	0	0
grid	1	3	4	**5**	0	3	0	3	4
gripper	8	**20**	**20**	**20**	8	**20**	10	**20**	**20**
hiking	2	3	**20**	**20**	2	7	5	3	3
logistics	2	7	**29**	15	2	2	3	6	5
maintenance	0	**14**	11	**14**	0	0	0	0	0
miconic	55	**150**	**150**	**150**	71	**150**	146	**150**	**150**
movie	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime	20	21	**32**	22	5	18	13	18	17
mystery	15	15	**17**	14	9	6	12	10	9
openstacks	0	0	2	**20**	0	1	8	2	5
organic-synthesis	**3**	**3**	2	**3**	**3**	**3**	**3**	**3**	2
parcprinter	0	12	**20**	18	0	0	1	0	0
parking	0	0	**7**	0	0	0	0	0	0
pathways	4	5	**10**	8	4	4	4	4	4
pegsol	17	**20**	**20**	**20**	18	**20**	**20**	**20**	**20**
pipesworld	12	22	23	**27**	12	13	15	20	17
psr	49	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
rovers	6	21	**26**	25	6	14	16	16	16
satellite	6	15	**27**	12	7	15	8	8	9
scanalyzer	4	**20**	18	**20**	5	18	11	**20**	18
snake	3	4	5	**7**	4	3	3	**7**	6
sokoban	6	13	**19**	10	8	11	12	10	9
spider	1	12	9	**19**	0	5	7	5	9
storage	14	18	**19**	**19**	17	15	**19**	18	**19**
termes	0	10	14	**15**	3	4	2	7	3
tetris	0	**20**	9	**20**	2	11	19	2	18
thoughtful	5	5	8	**14**	5	12	5	5	5
tidybot	3	19	16	**20**	0	1	7	8	2
tpp	6	13	23	**29**	6	15	10	14	15
transport	0	5	0	**16**	0	0	0	0	0
trucks	6	9	**15**	9	6	6	7	7	7
visitall	0	**20**	0	**20**	0	0	9	0	0
woodworking	1	1	2	**4**	1	1	1	1	1
zenotravel	8	**20**	**20**	**20**	7	9	9	13	14
SUM	359	775	933	**965**	346	651	558	631	632
$>h_{\text{gc}}$					4	6	4	6	7
$>h_{\text{ff}}$					2	5	5	4	3
$>h_{\text{lm}}$					0	2	1	0	0

Table 3: Instance-Specific Learning: IPC Benchmark Instances, 8GB, 30min total limit (including training data generation, training h_{nn}, and search using h_{nn}) per instance: Coverage Results. In the bottom 3 rows, "$>h_{\text{heuristic}}$" indicates the count # of domains with higher coverage than $h_{\text{heuristic}}$.

Beating LM-cut with LM-cut: Quick Cutting and Practical Tie Breaking for the Precondition Choice Function

Pascal Lauer and **Maximilian Fickert**
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
s8palaue@stud.uni-saarland.de, fickert@cs.uni-saarland.de

Abstract

LM-cut is one of the most popular heuristics in optimal planning that computes strong admissible estimates of the perfect delete relaxation heuristic h^+. The heuristic iteratively computes disjunctive action landmarks for the current state, reducing their action costs until no more landmarks with remaining action costs can be found. These landmarks are generated by finding cuts in the justification graph, which depends on a precondition choice function mapping each action to its most expensive precondition according to $h^{\max}$. This precondition is not necessarily unique, yet the performance of the heuristic heavily depends on this choice. We introduce and analyze several new tie breaking strategies for the precondition choice function, and evaluate their effectiveness on the IPC benchmarks. Furthermore, we suggest a modification to the computation of the cut, which trades a negligible loss in heuristic accuracy for a significant speedup of the LM-cut computation.

Introduction

In optimal classical planning, strong admissible heuristics are desired to find solutions through heuristic search. One such heuristic is the landmarks cut heuristic $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009), which provides high-quality estimates of the perfect delete relaxation heuristic h^+. While LM-cut has been surpassed by recent advancements on abstraction heuristics and cost partitionings (e.g. Franco et al. 2017; Seipp and Helmert 2018; Seipp, Keller, and Helmert 2020), it remains a popular heuristic that does not require any precomputation before search.

LM-cut is defined by iteratively computing $h^{\max}$ (Bonet and Geffner 2001), finding a disjunctive action landmark, and reducing the cost of these actions until the $h^{\max}$-value becomes zero. A precondition choice function (pcf) maps each action to a precondition with maximal $h^{\max}$-value. The pcf defines the $h^{\max}$ justification graph, where the facts of the task form the vertices and there are edges from the precondition returned by the pcf to each effect of an action. A disjunctive action landmark is constructed by computing a cut in the graph through actions leading into the zero-cost goal zone (nodes from which the goal can be reached with a zero-cost path), and the cost of these actions are reduced by the minimal action cost in that landmark.

Bonet and Helmert (2010) have shown that the heuristic is equal to h^+ if the landmarks are computed via hitting sets. They implemented variants of the heuristic using polynomial-time approximations of the hitting set and showed that this approach does improve the accuracy of the heuristic, but the improvement is outweighed by the added computational overhead.

However, there is also some room to make LM-cut more accurate without changing the algorithm: There can be multiple preconditions with the same (maximal) $h^{\max}$-value (in particular after the first few iterations when several actions have reduced costs), and their tie breaking is left unspecified. This detail has been mostly neglected in the literature, but as we demonstrate in our experiments, the performance of the heuristic varies a lot depending on the tie breaking strategy. Bonet et al. (Bonet and Helmert 2010; Bonet and Castillo 2011) introduced a variant with random tie breaking, where the heuristic is computed multiple times in each state and the maximum heuristic value is used. While the estimates of the heuristic values improve, requiring repeated computation of the heuristic diminishes its practical use. We explore several new tie breaking strategies that aim to improve the heuristic by generating more effective landmarks.

LM-cut provides accurate admissible estimates, but is expensive to calculate. Pommerening and Helmert (2012; 2013) show that the computational effort can be reduced through incremental computation, by caching landmarks and re-using them for the successor states. We introduce a new idea to speed up the LM-cut computation, by computing the cuts in the justification graph in a simplified and faster way. While this can lead to overapproximated cuts and make the heuristic less informed, this drawback is greatly outweighed by the improved computation speed.

We first summarize the general planning background and recapitulate the details of the LM-cut algorithm. Next, we describe and evaluate our optimization to the computation of the cuts. Finally, we introduce our new tie breaking strategies for the precondition choice function, explain the intuition behind them, and analyze their effectiveness with an empirical evaluation on the IPC benchmarks.

Background

We first introduce the necessary background and notations, before reviewing the details of the LM-cut algorithm.

Preliminaries

We consider classical planning using the STRIPS representation with action costs (Fikes and Nilsson 1971). A planning task is a 5-tuple $\Pi = (\mathcal{F}, \mathcal{A}, c, \mathcal{I}, \mathcal{G})$, where

- $\mathcal{F}$ is a finite set of *facts*,
- $\mathcal{A}$ is a set of actions, each $a \in \mathcal{A}$ is a triple of fact sets *preconditions* (pre_a), *add effects* (add_a), and *delete effects* (del_a) with $\mathsf{add}_a \cap \mathsf{del}_a = \emptyset$,
- c is a cost function $\mathcal{A} \mapsto \mathbb{R}_0^+$,
- $\mathcal{I} \subseteq \mathcal{F}$ is the *initial state*,
- $\mathcal{G} \subseteq \mathcal{F}$ are the *goal facts*.

A *state* $s \subseteq \mathcal{F}$ is a set of facts. An action $a \in \mathcal{A}$ is *applicable* in a state s if $\mathsf{pre}_a \subseteq s$, and applying a in s leads to the state $s[\![a]\!] := (s \setminus \mathsf{del}_a) \cup \mathsf{add}_a$. A *plan* for s is a successively applicable action sequence leading from s to a goal state $s^* \supseteq \mathcal{G}$, and is called *optimal* if it has minimal cost among all plans for s. A plan for Π is a plan for its initial state $\mathcal{I}$.

The set of all states is denoted by $\mathcal{S}$. A *heuristic function* (short *heuristic*) $h : \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \infty$ estimates the cost of a plan for a state. The *perfect heuristic* h^* returns the cost of an optimal plan. A heuristic h is called *admissible* if $h(s) \leq h^*(s)$ for all $s \in \mathcal{S}$, and A* (Hart, Nilsson, and Raphael 1968) is guaranteed to find optimal solutions when using an admissible heuristic.

LM-cut

LM-cut (Helmert and Domshlak 2009) is based on an iterative computation of (action) landmarks. A *disjunctive action landmark* is a set of actions L, such that every plan must include at least one action from L. In the following, we assume that $\mathcal{I} = \{i\}$, $\mathcal{G} = \{g\}$, and $|\mathsf{pre}_a| \geq 1$ for all actions $a \in \mathcal{A}$ (this can be achieved with simple transformations). The heuristic is computed in an iterative procedure, where each iteration performs the following steps:

1. Compute $h^{\max}$ (Bonet and Geffner 2001) for all facts. If $h^{\max}(g) = \infty$, return ∞. If $h^{\max}(g) = 0$, return the computed heuristic value.
2. Define a *precondition choice function* (pcf), mapping each action to a precondition with maximal $h^{\max}$-value.
3. Construct the *justification graph*. The facts $\mathcal{F}$ of the planning task are the vertices of the graph, and there is an arc from the precondition chosen by the pcf to each of its add effects for all actions. The arc is labeled with the action.
4. Partition the vertices into three sets: (a) the 0-cost goal zone V^*, containing all facts from which g is reachable with a 0-cost path, (b) the before-goal zone V^0, containing all facts reachable from i without passing through a node in V^*, and (c) all other vertices V^b. The labels of the arcs leading from V^0 into V^* define a disjunctive action landmark L.
5. Reduce the cost of actions in L by $c_{\min} = \min_{a \in L} c(a)$, and add $c_{\min}$ to the heuristic value (which starts at 0).

After the computation of the heuristic, the action costs are reset to their original values.

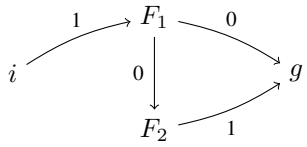


Figure 1: Example justification graph. A minimal cut only contains the action leading from i to F_1.

Our work addresses two aspects of LM-cut. In the next section, we describe a faster method to compute the cut (step 4). Afterwards, we introduce and evaluate tie breaking strategies for the precondition choice function (step 2).

Quick Cutting

In each iteration of LM-cut, the justification graph is built during the computation of $h^{\max}$: whenever a fact is assigned its final $h^{\max}$-value, (one of) its most expensive preconditions is stored as the predecessor in the justification graph. Afterwards, the zero-cost goal zone V^* is constructed using a backwards exploration from g considering only edges with cost 0. Finding the before-goal zone V^0 requires a separate forward exploration from i considering only arcs that do not lead into V^*, to ensure that all vertices found by this procedure are reachable without passing through V^*.

We suggest to skip the forward exploration phase, and instead just consider all non-zero cost arcs that lead from a vertex not in V^* to a vertex in V^* for the cut. These arcs can easily be identified at the end of the backward exploration phase, and iterating over those is generally much cheaper than the forward exploration. However, this strategy may overapproximate the cut, as for some of the actions in the cut, the precondition selected by the pcf may not be reachable from i without passing through the zero-cost zone.

Consider the example justification graph in Figure 1. The graph can be partitioned into $V^* = \{g, F_1\}$, $V^0 = \{i\}$, and $V^b = \{F_2\}$. According to the original algorithm, this would result in the cut containing only the arc from i to F_1, whereas with our method, the cut would additionally include the one from F_2 to g. In this example, the heuristic value would not be affected, but the heuristic value can potentially change if the additional action would otherwise be included in a cut in a later iteration of the LM-cut computation.

Note that, since our computation of the cut always contains the cut as computed by the original algorithm (but may potentially include more actions), it is still a disjunctive action landmark, and does not affect the properties of the heuristic (in particular admissibility).

Tie Breaking Strategies for LM-cut

The precondition choice function maps actions to a precondition with maximal $h^{\max}$-value, but in many cases there are multiple such preconditions which leaves room for tie breaking. Consider the justification graph shown in Figure 2. The action achieving g has three preconditions with maximal $h^{\max}$-values: v_1, v_2, or v_3 (indicated by the dashed arrows). If v_3 is selected by the precondition choice function,

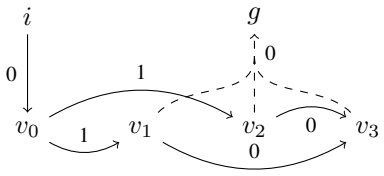


Figure 2: Example justification graph. The precondition choice function for the action achieving g can select any of v_1, v_2, or v_3.

then the cut will contain both remaining actions with cost one, and the computation will terminate. However, if v_1 or v_2 is selected, the cut only contains one of these actions, and an additional cut can be made afterwards, increasing the heuristic value. The example is inspired by the VisitAll domain, where a perfect tie breaking strategy will only select actions leading to a single location for the cut in each iteration (making $h^{\text{LM-cut}} = h^+$). Different tie breaking may result in significantly larger cuts (and thus smaller heuristic values), and similar cases where tie breaking is important appear on most other domains as well.

In the following, we first explain how tie breaking is applied in detail, and then introduce several tie breaking strategies that aim to improve the heuristic.

Tie Breaking in LM-cut

The pseudo code of the computation of the cut is shown in Algorithm 1. This shows our quick cutting method which only performs the backwards exploration from the goal; the original computation of the cut would perform the forward exploration afterwards to compute V^0, and would return $\{a \in \mathit{Discovered} \mid \mathit{pcf}(a) \in V^0\}$ instead.

It is important to note that for an action a, the tie breaking of the precondition choice function is only applied the first time $\mathit{pcf}(a)$ is evaluated — all subsequent calls to $\mathit{pcf}(a)$ will return the same selected precondition. Furthermore, at the time of the tie breaking, only a fragment of the final zero-cost goal zone V^* is known.

Our tie breaking methods aim to reduce the size of the cuts, i.e., generating smaller disjunctive action landmarks in step 4 of each LM-cut iteration. While this does not guarantee that the heuristic values will improve, it is reasonable to assume that reducing the cost of fewer actions will result in more cuts being made before $h^{\max}$ evaluates to zero.

In the following, we refer to the precondition selected by $\mathit{pcf}(a)$ as the *supporter* of a. We call an action a an *achiever* of a fact f if $f \in \mathsf{eff}_a$. Our first tie breaking strategies aim to reduce the size of V^*, with the assumption that this leads to fewer actions pointing into V^* and thereby smaller cuts.

Algorithm 1: L computation

```
1  V* ← {g}, Explored ← ∅, Discovered ← ∅
2  while Explored ≠ V* do
3      select f ∈ V* \ Explored
4      Explored ← Explored ∪ {f}
5      for each a ∈ A with f ∈ eff_a and
           h^max(p) < ∞ for all p ∈ pre_a do
6          if c(a) = 0 then
7              V* ← V* ∪ {pcf(a)}
8          else
9              Discovered ← Discovered ∪ {a}
10 return {a ∈ Discovered | pcf(a) ∉ V*}
```

$\mathbf{V^*}$ Detection (GZD) Prefer a precondition that is already in the zero-cost goal zone V^*. If such a precondition exists, then choosing it as the supporter can not increase V^*. Note that any further tie breaking among multiple potential supporters that are already in V^* has no effect.

Border Detection (BD) Prefer a precondition that has no zero-cost achievers. If such a supporter is selected, the zero-cost goal zone will not expand beyond that fact, as there are no further zero-cost actions to choose from. In our motivational example (Figure 2), this strategy would prefer v_1 and v_2 over v_3 as intended, minimizing V^*.

Zero-Cost Achievers (ZCA) Prefer a precondition with a minimal number of zero-cost achievers. This strategy is an extension of the previous one (BD), but imposes a ranking to the potential supporters if there is no precondition without zero-cost achievers. The idea is again to approximate how many additional facts may be added to V^* when recursively exploring the supporter. However, some of the zero-cost achievers may originate from other facts already in V^*, and these achievers would be beneficial in keeping V^* small, so this approximation may not always be accurate.

Value Decrease Minimization (VDM) Prefer a precondition of which the $h^{\max}$-value since the first iteration decreased the least. While the previous two strategies aim to reduce the breadth of the backward exploration beyond the supporter, this strategy aims to reduce the depth. If the $h^{\max}$-value of the precondition p is close to its value from the first iteration, then there should not be many zero-cost actions in the justification graph between i and p. In our example (Figure 2), assuming that the actions leading from v_1 to v_3 and v_2 to v_3 initially had a cost of 1, then this strategy would also prefer v_1 or v_2 over v_3.

Zero-Cost Path (ZCP) Prefer a precondition p that minimizes the number of zero-cost actions on a path from i to p. Note that when the precondition choice function is called, the justification graph is still being constructed backwards from the goal so we do not know the structure of the graph between i and p. Therefore, we partially re-use the justification graph from the previous iteration. More specifically, we keep track of the path information for each fact during the $h^{\max}$ computation: When the $h^{\max}$-value of a fact is set, we consider the justification graph from the previous LM-cut iteration including the incremental updates (with arbitrary tie breaking) of the current $h^{\max}$ computation so far. For the evaluation of the precondition choice function, we then select the candidate with the fewest zero-cost actions on its path. Like VDM, this strategy aims to reduce the depth of the

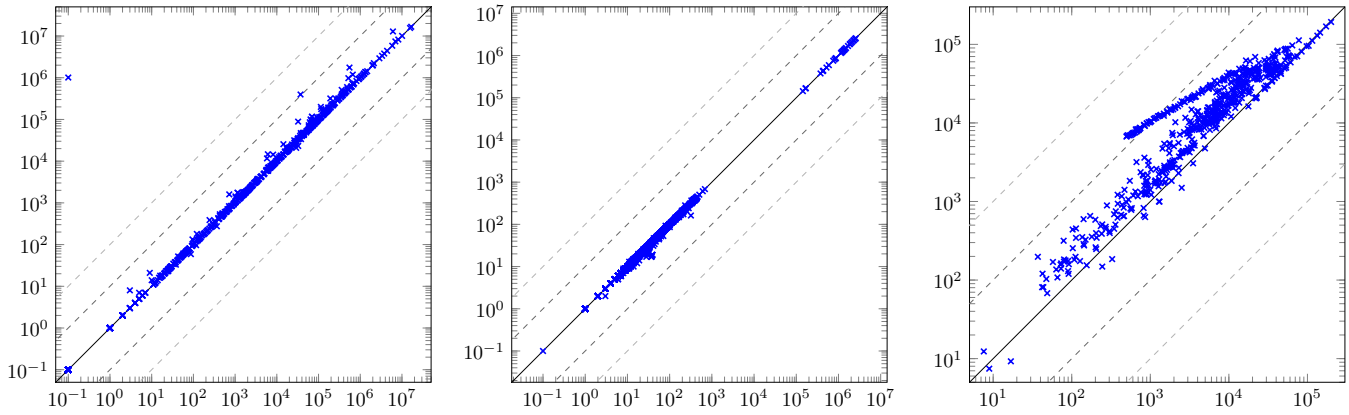


Figure 3: Number of expansions before the last f-layer (left), initial heuristic value (middle), and evaluations per second (right) for original (x-axis) vs. quick (y-axis) cutting. Expansions and evaluations per second are shown for commonly solved instances, and instances with a search time of less than 0.01 seconds are excluded for the latter to reduce noise.

backward exploration by reducing the number of zero-cost actions between this precondition and i, but uses a different approximation.

Achiever Minimization (AM) Prefer a precondition that has a minimal number of achievers with reachable preconditions under $h^{\max}$. Each achiever of a fact f adds an incoming arc to the corresponding node in the justification graph. Preferring preconditions with fewer achievers leads to fewer iterations in the backwards exploration (Algorithm 1, line 5). Thus, we can expect V^* and *Discovered* to stay smaller, which should lead to fewer actions in the cut.

Experiments

We implemented our techniques in Fast Downward (Helmert 2006), on top of the existing implementation of LM-cut. For our evaluation, we include all solvable instances from the optimal tracks of the IPCs up to 2018 that do not have conditional effects or axioms, resulting in a total of 1672 unique instances from 48 domains. The experiments were run using the Lab framework (Seipp et al. 2017) on a cluster of machines with Intel Xeon E5-2660 CPUs with a clock rate of 2.2 GHz. LM-cut is run in A* with time and memory limits of 30 minutes and 4 GB respectively. We first assess the impact of our alternative computation of the cut, before evaluating our new tie breaking strategies.

Quick Cutting

Our new method of computing the cut should improve the computational efficiency of the heuristic, but at a potential loss of informativeness.

The left plot of Figure 3 shows a comparison of the number of expansions to the last f-layer between the two methods of computing the cut. Across all commonly solved instances, our method of computing the cut leads to 6.6% more expansions on average. The domains where expansions increase the most are VisitAll ($+70\%$), Sokoban ($+40\%$), and Depot ($+23\%$), though the increase is typically low (less than 1% on most domains). The initial heuristic value does not decrease in 1396 of the 1672 instances (see the middle plot of Figure 3).

On the other hand, we obtain a huge speed-up in the computation of the heuristic as shown in the right plot of Figure 3. On average, with our method we can achieve 91% more heuristic evaluations per second (up to 586% in Miconic), which more than makes up for the comparatively small loss in informativeness. This leads to a decreased search time across almost all domains, and we are able to solve 28 more instances (856 vs. 828) on our benchmark set (the most significant gains are $+4$ in Parking and $+3$ in Floortile).

The single exception is Organic Synthesis (split). Since quick cutting overapproximates the cuts which leads to more actions having their cost reduced, intuitively, one would expect that this should lead to fewer iterations of $h^{\max}$ per computation of $h^{\text{LM-cut}}$ on average. However, this is not always the case, as demonstrated by the following example. Assume there are two actions a_1 and a_2 with cost 2, which, using the original cutting, would be included in a cut at some point, having both of their costs reduced by 2. If instead a_2 was included in an earlier (overapproximated) cut which only reduced its cost by 1, then reducing both a_1 and a_2 to 0 may now require two cuts (first decreasing a_1 and a_2 by 1, and then reducing only a_2 by 1). Such cases appear frequently in Organic Synthesis. While the resulting heuristic value is typically not affected, it can significantly diminish the computational advantage of our cutting method. The coverage on Organic Synthesis does not decrease, but the number of evaluations per second drops by 18%, increasing the search time by 24% on average.

Tie Breaking Strategies

Table 1 shows the coverage of LM-cut with several tie breaking strategies. All configurations use our new method of computing the cuts (we also ran the experiments with the original cutting which exhibited a similar relative performance of the different tie breaking strategies). As baselines, we consider the (arbitrary) tie breaking employed by the cur-

Coverage	arb	inv	rnd	GZD	BD	ZCA	VDM	ZCP	AM	GZD+BD	Scorpion
Airport (50)	28	27	23	**29**	27	28	24	24	27	28	**29**
Blocks (35)	**28**	27	**28**	**28**	**28**	**28**	**28**	**28**	**28**	**28**	**28**
DataNetwork (20)	12	12	12	13	12	12	12	12	12	13	**14**
Depot (22)	7	7	7	7	7	7	7	7	7	10	**13**
DriverLog (20)	13	14	13	14	13	13	13	13	13	13	**15**
Elevators (30)	22	22	20	22	22	22	22	22	22	22	**24**
Freecell (80)	15	15	15	24	16	12	15	15	21	33	**64**
Grid (5)	2	2	1	2	2	2	2	2	2	2	**3**
Hiking (20)	10	10	9	10	9	8	9	9	10	9	**14**
Logistics (63)	27	27	25	27	25	25	25	25	25	27	**34**
Mprime (35)	23	25	22	23	23	23	22	22	25	24	**31**
Mystery (19)	16	17	15	17	17	17	17	17	17	17	**19**
Nomystery (20)	16	16	14	17	15	14	18	18	16	18	**20**
Openstacks (80)	31	31	31	31	31	30	31	31	31	31	**34**
OrgSynth-split (20)	**15**	**15**	14	**15**	14	10	**15**	**15**	**15**	**15**	10
Parcprinter (30)	19	22	19	22	19	19	22	22	18	20	**30**
Parking (40)	9	9	6	9	10	10	12	12	8	**13**	**13**
Pegsol (36)	**35**	34	33	**35**	34	34	**35**	34	34	**35**	**35**
Pipes-notank (50)	18	18	17	18	17	17	18	18	17	18	**25**
Pipes-tank (50)	12	12	10	12	11	9	12	12	12	12	**18**
PNetAlignment (20)	**9**	**9**	7	**9**	**9**	**9**	**9**	**9**	**9**	**9**	0
Rovers (40)	9	**11**	9	9	9	9	9	9	9	9	9
Satellite (36)	8	12	7	8	14	13	**15**	**15**	10	14	8
Scanalyzer (30)	16	16	16	16	15	14	16	16	16	16	**18**
Snake (20)	6	6	4	6	4	4	6	6	6	7	**13**
Sokoban (30)	**30**	29	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
Spider (20)	11	11	9	12	11	9	11	11	10	12	**15**
Termes (20)	7	6	6	7	6	6	7	7	6	7	**13**
Tidybot (40)	**23**	22	20	**23**	20	15	22	22	22	**23**	22
VisitAll (40)	16	15	17	15	**36**	**36**	**36**	**36**	14	**36**	30
Woodworking (30)	19	22	19	19	20	22	20	20	22	20	**30**
Zenotravel (20)	**13**	**13**	12	**13**	12	12	**13**	**13**	**13**	12	**13**
Others (601)	331	331	331	331	331	331	331	331	331	331	**346**
Sum (1672)	856	865	821	873	869	850	884	883	858	914	**1020**

Table 1: Coverage for LM-cut with different tie breaking methods. Domains where coverage across tie breaking methods does not change are grouped to "Others".

rent implementation of Fast Downward ("arb")[1], its inverse ("inv"), and random tie breaking ("rnd"). The results for random tie breaking are averaged and rounded over 5 random seeds, though there is very little variance (coverage changes across different seeds on only 4 instances, and overall coverage was always between 820 and 822). We include results for all our tie breaking strategies as well as selected combinations thereof. Furthermore, we added Scorpion (Seipp 2018)[2] as a representative of the state of the art.

This first thing to note is that the performance of LM-cut is heavily dependent on the tie breaking strategy, as the overall coverage ranges from 821 (with random tie breaking) to 914 (with one of our combined methods). The biggest differences can be seen in Freecell and VisitAll: depending on the tie breaking, the initial heuristic value can change by over a factor of 3 (Freecell) respectively 5 (VisitAll), and coverage ranges from 12 to 33 respectively 14 to 36. Most of our introduced strategies outperform the baselines, and random tie breaking is particularly bad across most domains. Four of our tie breaking strategies (BD, ZCA, VDM, and ZCP) lead to significant gains on VisitAll (and similarly on Satellite), as these strategies effectively solve the issue described in our corresponding example (Figure 2). In Freecell on the other hand, our other two strategies (GZD and AM) work best, increasing coverage by 9 respectively 6 over the baselines.

Combined Tie Breaking Strategies In case of remaining ties, multiple tie breaking strategies can be used in sequence to break the remaining ties. In preliminary experiments, we had most success with combining tie breaking strategies that complement each other. For example, using any other strat-

[1]While there is no explicit tie breaking, typically the first one according to Fast Downward's variable ordering is selected.

[2]We disabled the h^2 preprocessor (Alcázar and Torralba 2015) to make it more comparable to our planner which is not using it.

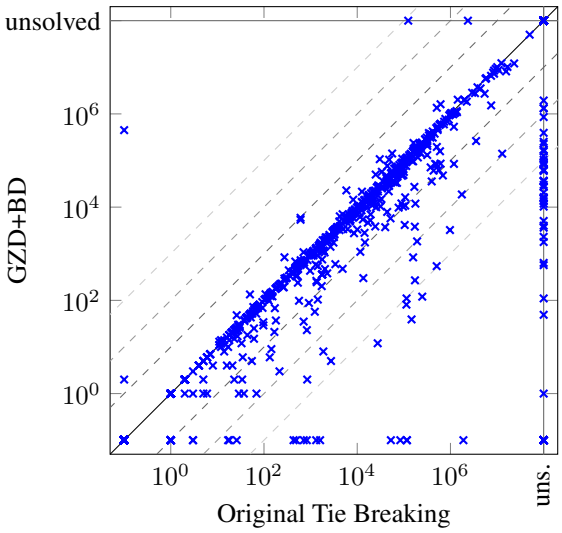


Figure 4: Number of expansions before the last f-layer for GZD+BD and original tie breaking.

egy after GZD or BD worked really well, as both strategies seem to make good tie breaking choices if their criterion applies, but yield no further information in case such a precondition does not exist. In that case, using a different strategy provides additional information on top, improving the performance of the heuristic. In contrast, VDM and ZCP both aim to reduce the depth of the backward exploration, and combining them had little effect.

The best configuration we have found so far prefers supporters that are already in V^*, and breaks the remaining ties according to BD (GZD+BD). This configuration retains the good performance on VisitAll, and further improves results on Freecell ($+9$ coverage compared to the best individual tie breaking method), Depot ($+3$), Parking ($+1$), and Snake ($+1$). Compared to the original tie breaking, coverage increases in 13 domains and decreases in only 2, solving 58 more instances overall. The number of expansions to the last f-layer decreases significantly (see Figure 4), sometimes by several orders of magnitude.

Compared to Scorpion, LM-cut is still worse on most domains. However, on Organic Synthesis (split) and Petri Net Alignment, the previous implementation of LM-cut already beat Scorpion; and with our improvements, LM-cut pulls ahead also in Sattelite ($+7$ coverage), VisitAll ($+6$), Rovers ($+2$), Tidybot ($+1$), and Miconic ($+1$).

Conclusion

In this work, we introduced an optimization to LM-cut addressing the computation of the cut in the justification graph, and introduced and evaluated an extensive set of tie breaking strategies for the precondition choice function. Both contributions significantly improve the performance of LM-cut on the IPC benchmarks: our best performing configuration beats the previous implementation by 914 vs. 828 solved instances, increasing coverage by 86 using the same heuristic and search engine.

For future work, we want to explore additional tie breaking strategies, in particular ones that consider *which* actions should be included in the cut. Our current strategies only aim to include as few actions in the cut as possible, however, there may be cases where cutting more actions may be preferable (for example, if there is a single action that, if cut, enables a zero-cost path to the goal, but multiple other actions could be cut instead without enabling such a path).

Additionally, we want to analyze the combinations of multiple tie breaking strategies in more depth. In preliminary experiments, we had some good results, yet some combinations had surprisingly adverse effects on performance. Understanding the cause of these effects may allow us to combine our methods more effectively, and find combinations that may further improve the heuristic.

Furthermore, our methods seem complementary to incremental computation of LM-cut (Pommerening and Helmert 2013), and could be easily combined. This should further boost the performance of the heuristic, and make it more competitive with the state of the art.

Acknowledgments

Maximilian Fickert was funded by DFG grant 389792660 as part of TRR 248 – CPEC (see https://perspicuous-computing.science).

References

[Alcázar and Torralba 2015] Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*, 2–6. AAAI Press.

[Bonet and Castillo 2011] Bonet, B., and Castillo, J. 2011. A complete algorithm for generating landmarks. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*. AAAI Press.

[Bonet and Geffner 2001] Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

[Bonet and Helmert 2010] Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10)*, 329–334. Lisbon, Portugal: IOS Press.

[Fikes and Nilsson 1971] Fikes, R. E., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

[Franco et al. 2017] Franco, S.; Torralba, A.; Lelis, L. H.; and Barley, M. 2017. On creating complementary pattern databases. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4302–4309. AAAI Press/IJCAI.

[Hart, Nilsson, and Raphael 1968] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic de-

termination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

[Helmert and Domshlak 2009] Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

[Helmert 2006] Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

[Pommerening and Helmert 2012] Pommerening, F., and Helmert, M. 2012. Optimal planning for delete-free tasks with incremental LM-Cut. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, 363–367. AAAI Press.

[Pommerening and Helmert 2013] Pommerening, F., and Helmert, M. 2013. Incremental lm-cut. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 162–170. Rome, Italy: AAAI Press.

[Seipp and Helmert 2018] Seipp, J., and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research* 62:535–577.

[Seipp et al. 2017] Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

[Seipp, Keller, and Helmert 2020] Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research* 67:129–167.

[Seipp 2018] Seipp, J. 2018. Scorpion. In *IPC 2018 planner abstracts*, 77–79.

Online Saturated Cost Partitioning for Classical Planning

Jendrik Seipp
University of Basel
Basel, Switzerland
jendrik.seipp@unibas.ch

Abstract

Saturated cost partitioning is a general method for admissibly adding heuristic estimates for optimal state-space search. The algorithm strongly depends on the order in which it considers the heuristics. The strongest previous approach precomputes a set of diverse orders and the corresponding saturated cost partitionings before the search. This makes evaluating the overall heuristic very fast, but requires a long precomputation phase. By diversifying the set of orders online during the search we drastically speed up the planning process and even solve slightly more tasks.

Saturated Cost Partitioning

One of the main approaches for solving classical planning tasks optimally is using the A* algorithm (Hart, Nilsson, and Raphael 1968) with an admissible heuristic (Pearl 1984). Since a single heuristic usually fails to capture enough details of the planning task, it is often beneficial to compute multiple heuristics and to combine their estimates (Holte et al. 2006). The preferable method for admissibly combining heuristic estimates is cost partitioning (Haslum, Bonet, and Geffner 2005; Haslum et al. 2007; Katz and Domshlak 2008, 2010; Pommerening, Röger, and Helmert 2013). By distributing the original costs among the heuristics, cost partitioning makes the sum of heuristic estimates (under the reduced cost functions) admissible.

Saturated cost partitioning (SCP) is one of the strongest methods for finding cost partitionings (Seipp, Keller, and Helmert 2020). At the core of the SCP algorithm lies the insight that we can often reduce the (action) cost function of a planning task and still obtain the same heuristic estimates. This notion is captured by so-called *saturated* cost functions. An (action) cost function *scf* is saturated for a heuristic h, an original cost function *cost* and a subset S' of states in the planning task, if $\mathit{scf}(a) \leq \mathit{cost}(a)$ for each action a and for all states $s \in S'$ the heuristic estimate by h for s is the same regardless of whether we evaluate h under *cost* or *scf*. We call a function that computes a saturated cost function for a given heuristic and cost function a *saturator*.

Algorithm 1 shows how the SCP procedure computes saturated cost functions that form a cost partitioning of a given cost function *cost* over an ordered sequence of heuristics ω. The algorithm starts by computing a saturated cost function for the first heuristic h in ω, i.e., it lets a saturator $\mathit{saturate}_h$ compute the fraction of the action costs that are needed to preserve the estimates by h for a subset of states under the original cost function (line 4). Afterwards, it iteratively subtracts the costs given to h from the original costs (line 6) and considers the next heuristic until all heuristics have been treated this way. The sequence of computed saturated cost functions forms the resulting cost partitioning C. We write h^{SCP}_ω for the cost partitioning heuristic that results from applying the SCP algorithm to the heuristic order ω.

Algorithm 1 Compute a saturated cost partitioning over an ordered sequence of heuristics ω for a cost function *cost*.

```
1: function SaturatedCostPartitioning(ω, cost)
2:     C ← ⟨⟩
3:     for all h ∈ ω do
4:         scf ← saturate_h(cost)
5:         append scf to C
6:         cost(a) ← cost(a) − scf(a) for all actions a
7:     return C
```

The original SCP formulation assumed S' to always be the set of all states. This definition has been generalized recently to allow preserving the estimates for a *subset* of states, giving rise to new saturator types (Seipp and Helmert 2019). One of the new saturators is *perim*, which preserves the estimates of all states within a given perimeter of the goal. For example, for a given heuristic h and a state s we can use perim to preserve the heuristic estimates of all states s' with $h(s') \leq h(s)$ (and reduce all higher estimates to $h(s)$). The perim saturator often yields higher estimates for a given state than the *all* saturator, which preserves all estimates. However, perim also often ignores costs that could be used to improve the heuristic estimates of other states. Therefore, the strongest method by Seipp and Helmert (2019), *perim**, first computes a saturated cost partitioning using perim and then uses the remaining costs to compute a saturated cost partitioning that preserves all estimates under the remaining cost function. We use *perim** in all experiments below.

The quality of an SCP heuristic greatly depends on the order in which the heuristics are considered. In this work, we use the *greedy* ordering method with the $\frac{h}{\mathit{stolen}}$ scoring function, the best ordering in previous work on saturated cost partitioning (Seipp, Keller, and Helmert 2020). For each

Algorithm 2 Offline diversification. Find a diverse set of heuristic orders Ω for SCP before the search.

```
1:  function OFFLINEDIVERSIFICATION
2:      Ω ← ∅
3:      Ŝ ← sample 1000 states
4:      repeat
5:          s ← sample state
6:          ω ← greedy order for s
7:          if ∃s' ∈ Ŝ : h^SCP_ω(s') > sup_{ω'∈Ω} h^SCP_ω'(s') then
8:              Ω ← Ω ∪ {ω}
9:      until time spent in function ≥ T
10:     return Ω
```

heuristic h and a given state s, it computes the fraction of $h(s)$ over the costs "stolen" by h, i.e., the amount of costs that h wants to steal from other heuristics for preserving its estimates. Then the greedy method orders heuristics by their $\frac{h}{stolen}$ fractions in decreasing order. As in previous work on SCP, we focus on *abstraction* heuristics (Helmert, Haslum, and Hoffmann 2007).

Offline Diversification of SCP Heuristics Most of the previous work on the topic precomputes SCPs *offline*, i.e., before the search and then computes the maximum over the SCP heuristic estimates for a given state during the search. Algorithm 2 shows the strongest offline SCP algorithm from the literature (Seipp, Keller, and Helmert 2020). It samples 1000 states $\hat{S}$ with random walks (line 3) and then iteratively samples a new state s (line 5), computes a greedy order ω for s (line 6) and keeps ω if it is *diverse*, that is, h^{SCP}_ω yields a higher heuristic estimate for any of the samples in $\hat{S}$ than all previously stored orders (lines 7–8). (The supremum of the empty set is $-\infty$.) The offline *diversification* procedure stops and returns the found set of orders Ω after reaching a given time limit. This last characteristic is the main drawback of the algorithm: the A* search can only start after the offline diversification finishes and so far there is no good stopping criterion except for a fixed time limit. Seipp, Keller, and Helmert (2020) showed that a limit of 1000 seconds leads to solving the highest number of IPC benchmarks in 30 minutes, but such a high time limit obviously bloats the solving time for many tasks, especially for those that blind search would solve instantly.

Online Computation of SCP Heuristics Instead of precomputing SCP heuristics before the search, we can also compute them *online*, i.e., during the search. This approach, which we call *online-nodiv*, computes a greedy order and the corresponding SCP heuristic for each state evaluated during the search. By design, online-nodiv can start the A* search immediately and it has access to the states that are actually evaluated by A* and not only to randomly sampled states like the offline diversification procedure. As a result, the online-nodiv method has been shown to work well for landmark heuristics (Seipp, Keller, and Helmert 2017). However, computing an SCP over abstraction heuristics for each evaluated state slows down the heuristic evaluation so much that the online variant solves much fewer tasks than precomputed SCP heuristics (Seipp, Keller, and Helmert 2020). This kind of result is typical for optimal classical planning: more work per evaluated state often results in better estimates but does not outweigh the slower evaluation speed (e.g., Karpas, Katz, and Markovitch 2011; Seipp, Pommerening, and Helmert 2015).

Algorithm 3 Online diversification. Simultaneously diversify a set of orders Ω for SCP and compute the maximum over all induced SCP heuristic values for a given state s.

```
1:  function COMPUTEHEURISTIC(Ω, s)
2:      if SELECT(s) and time spent in function < T then
3:          ω ← greedy order for s
4:          if h^SCP_ω(s) > sup_{ω∈Ω} h^SCP_ω(s) then
5:              Ω ← Ω ∪ {ω}
6:      return max_{ω∈Ω} h^SCP_ω(s)
```

Online Diversification of SCP Heuristics

In this work, we combine ingredients of the offline and online-nodiv variants to obtain the benefits of both, i.e., fast solving times and high total coverage. More precisely, we interleave heuristic diversification and the A* search: for a subset of the evaluated states, we compute a greedy order and store the corresponding SCP heuristic if it yields a more accurate estimate for the state at hand than all previously stored SCP heuristics.

Algorithm 3 shows pseudo-code for the approach, which adapts the COMPUTEHEURISTIC function used to evaluate a state. Before COMPUTEHEURISTIC is called for the first time, we initialize the set of heuristic orders Ω for SCP to be the empty set.[1] When evaluating a state s, we let the state selection function SELECT decide whether to use s for diversifying Ω (line 2). We discuss several state selection functions below, but all of them select the initial state for diversification. If s is selected, we compute a greedy order ω for s (line 3) and check whether ω induces an SCP heuristic h^{SCP}_ω with a higher estimate for s than all previously stored orders (line 4). If that is the case, we store ω (line 5). Finally, we return the maximum heuristic value for s over all SCP heuristics induced by the stored orders (line 6).

Compared to offline diversification, this online diversification algorithm has the advantage that it allows the A* search to start immediately and it doesn't need to sample states with random walks, but can judge the utility of storing an order based on states that are actually evaluated during the search. Compared to computing a saturated cost partitioning heuristic for each evaluated state (online-nodiv), online diversification evaluates states much faster and consequently solves many more tasks.

[1] Note that we could initialize Ω with a set of orders diversified offline. However, preliminary experiments showed that this only has a mild advantage over pure offline and pure online variants, so we only consider the pure variants here.

Time Limit

For abstraction heuristics, the offline diversification can perform two rather subtle optimizations compared to the online diversification: after precomputing all SCP heuristics, we can delete all abstract transition systems from memory, since during the search we only need the abstraction functions, which map from concrete to abstract states. Furthermore, for abstractions that never contribute any heuristic information under the set of precomputed orders, we can even delete the corresponding abstraction functions (Seipp 2018). While both optimizations often greatly reduce the memory footprint, the latter also speeds up the heuristic evaluation since we need to map the concrete state to its abstract counterpart for fewer abstractions.

To allow the online diversification to do these two optimizations, we need to stop the diversification eventually. We therefore introduce a time limit T and only select a state for diversification (line 2) if the total time spent in COMPUTEHEURISTIC is less than T.

State Selection Strategies

We now discuss three instantiations of the SELECT function, i.e., strategies for choosing the states for which to diversify the set of orders.

Interval The first strategy selects every i-th evaluated state for a given value of i. The motivation for this strategy is to distribute the time for diversification across the state space, in order to select states for diversification that are different enough from each other to let the corresponding SCP heuristics generalize to many unseen states. Note that for i=1 this strategy selects all states until hitting the diversification time limit T. For i=1 and T=∞ the resulting heuristic dominates the online SCP variant without diversification (online-nodiv), because both heuristics compute the same SCP heuristic for the currently evaluated state, but the variant with diversification also considers all previously stored orders.

Novelty This strategy makes the notion of "different states" explicit by building on the concept of *novelty* (Lipovetzky and Geffner 2012). Novelty is defined for factored states spaces, i.e., where each state s is defined by a set of atoms (atomic propositions) that hold in s. The novelty of a state s is the size of the smallest conjunction of atoms that is true in s and false in all states previously evaluated by the search. For a given value of k, the novelty strategy selects a state if it has a novelty of at most k.

Bellman The last strategy selects a state s if the maximum over the currently stored SCP heuristics h^{SCP}_Ω violates the Bellman optimality equation (1957) for s and its successor states, i.e., if $h^{\text{SCP}}_\Omega(\mathit{cost}, s) < \min_{s \xrightarrow{a} s' \in T} h^{\text{SCP}}_\Omega(\mathit{cost}, s') + \mathit{cost}(a)$, where T is the set of transitions in the planning task. Whenever the Bellman optimality equation is violated for a state s, we know that the current estimate for s is lower than the true goal distance of s, in which case it seems prudent to select s for diversification.

Experiments

We implemented online diversification for saturated cost partitioning in the Fast Downward planning system (Helmert 2006) and used the Downward Lab toolkit (Seipp et al. 2017) for running experiments on Intel Xeon Silver 4114 processors. Our benchmark set consists of all 1827 tasks without conditional effects from the optimal sequential tracks of the International Planning Competitions 1998–2018. We limit time by 30 minutes and memory by 3.5 GiB. All benchmarks, code and experiment data have been published online (Seipp 2020).

For the heuristic set on which SCP operates, we use the combination of pattern databases found by hill climbing (Haslum et al. 2007), systematic pattern databases of sizes 1 and 2 (Pommerening, Röger, and Helmert 2013) and Cartesian abstractions of *landmark* and *goal* task decompositions (Seipp and Helmert 2018). When comparing planning algorithms, we focus on the number of solved tasks, i.e., the *coverage* of a planner and its *time score* (used for the agile track of IPC 2018). The time score of a planner P for a task that P solves in t seconds is defined as $1 - \frac{\log(t)}{\log(T)}$, where T is the time limit, i.e., 1800 seconds in our case. The time score is 0 if P fails to solve the task within 1800 seconds. The total coverage and time score of a planner is the sum of its scores over all tasks.

Reevaluating States

The offline diversification algorithm finds a set of heuristic orders and maximizes over the corresponding SCP heuristics during the search. With such a fixed set of orders, the overall heuristic value of a state never changes. When we diversify the set of orders online during the search however, the heuristic estimate of a state s can increase after the time when s is generated, evaluated and added to the open list. Consequently, the heuristic estimates of states in the open list may be too low and it might be beneficial to reevaluate each state when retrieving it from the open list and postponing its expansion if its heuristic value has increased. To test this, we compare online diversification without and with state reevaluations in Table 1 (columns *on-stable* and *online*). The results show that reevaluating states increases the number of solved tasks in three domains (scanalyzer, tetris and tidybot) and never lets coverage decrease.

Our implementation uses the fact that we only need to reevaluate a state for the additional orders that we stored since its last evaluation. This minimizes the overhead incurred by the state reevaluations and makes the *online* variant solve tasks faster than *on-stable* in many domains (see right part of Table 1). Due to these results we let all online diversification variants reevaluate states in the experiments below.

State Selection Strategies

In the next experiment, we compare the different instantiations of the SELECT function. The left and middle parts of Table 2 hold per-domain and overall coverage results for the interval strategy with different intervals, the novelty strategy for k=1 and k=2 and the Bellman strategy. All strategies use

	Coverage				Time Score			
	offline	on-nodiv	on-stable	online	offline	on-nodiv	on-stable	online
agricola (20)	**0**	**0**	**0**	**0**	**0.0**	**0.0**	**0.0**	**0.0**
airport (50)	**34**	24	**34**	**34**	2.5	18.5	**26.1**	**26.1**
barman (34)	**4**	0	**4**	**4**	0.3	0.0	**2.0**	1.8
blocks (35)	**28**	20	**28**	**28**	2.2	19.5	**29.1**	**29.1**
childsnack (20)	**0**	**0**	**0**	**0**	**0.0**	**0.0**	**0.0**	**0.0**
data-network (20)	**14**	11	**14**	**14**	1.1	7.2	**12.1**	**12.1**
depot (22)	**13**	6	**13**	**13**	1.0	3.3	9.6	**9.8**
driverlog (20)	**15**	7	**15**	**15**	1.1	4.3	10.5	**10.7**
elevators (50)	**44**	12	**44**	**44**	3.4	2.6	25.9	**26.6**
floortile (40)	**6**	0	**6**	**6**	0.3	0.0	**0.8**	**0.8**
freecell (80)	**68**	30	**68**	**68**	4.8	10.9	35.8	**36.1**
ged (20)	**19**	7	**19**	**19**	1.5	4.7	12.1	**12.2**
grid (5)	**3**	1	**3**	**3**	0.2	1.0	2.2	**2.3**
gripper (20)	**8**	6	**8**	**8**	0.6	4.8	**7.0**	6.8
hiking (20)	14	8	**15**	**15**	1.0	5.1	**10.9**	10.7
logistics (63)	**39**	19	**39**	**39**	2.8	11.8	25.2	**26.0**
miconic (150)	**144**	133	**144**	**144**	11.2	80.6	131.6	**131.9**
movie (30)	**30**	**30**	**30**	**30**	2.4	**42.7**	42.4	42.6
mprime (35)	**29**	24	**29**	**29**	2.3	19.1	26.5	**26.7**
mystery (30)	**19**	15	**19**	**19**	1.5	12.6	**17.8**	**17.8**
nomystery (20)	**20**	12	**20**	**20**	1.5	8.0	15.3	**15.4**
openstacks (100)	**53**	21	**53**	**53**	3.8	12.0	29.8	**30.0**
organic (20)	**7**	**7**	**7**	**7**	0.5	6.0	**6.1**	**6.1**
organic-split (20)	**10**	6	**10**	**10**	0.7	1.9	**4.2**	**4.2**
parcprinter (50)	**38**	34	**38**	**38**	2.9	28.2	34.1	**34.2**
parking (40)	**13**	1	**13**	**13**	0.9	0.1	**5.4**	**5.4**
pathways (30)	**5**	4	**5**	**5**	0.3	5.1	**5.5**	5.4
pegsol (50)	**48**	42	**48**	**48**	3.7	22.8	**35.6**	35.4
petri-net (20)	**0**	**0**	**0**	**0**	**0.0**	**0.0**	**0.0**	**0.0**
pipes-nt (50)	**25**	14	**25**	**25**	1.8	10.4	**19.0**	18.7
pipes-t (50)	**18**	8	**18**	**18**	1.3	5.1	**12.1**	12.0
psr-small (50)	**50**	49	**50**	**50**	3.9	48.2	**54.6**	54.5
rovers (40)	**8**	7	**8**	**8**	0.6	6.6	**8.1**	**8.1**
satellite (36)	**7**	6	**7**	**7**	0.5	5.5	**7.3**	7.2
scanalyzer (50)	**35**	7	33	**35**	2.7	5.7	19.6	**21.2**
snake (20)	**12**	6	**12**	**12**	0.9	2.5	**7.6**	7.4
sokoban (50)	**50**	33	**50**	**50**	3.8	19.6	39.5	**39.9**
spider (20)	**15**	7	**15**	**15**	1.1	2.9	**8.6**	8.5
storage (30)	**16**	14	**16**	**16**	1.2	12.5	**17.1**	**17.1**
termes (20)	**12**	0	**12**	**12**	0.8	0.0	**3.2**	**3.2**
tetris (17)	**11**	3	10	**11**	0.8	1.3	5.4	**5.5**
tidybot (40)	**25**	18	24	**25**	1.8	5.8	15.3	**15.4**
tpp (30)	**8**	7	**8**	**8**	0.6	8.0	**8.9**	**8.9**
transport (70)	34	20	**36**	**36**	2.6	10.4	**22.4**	**22.4**
trucks (30)	**13**	9	**13**	**13**	0.9	5.3	8.6	**8.8**
visitall (40)	30	**33**	30	30	2.3	27.8	**30.3**	**30.3**
woodwork (50)	**49**	38	**49**	**49**	3.8	24.3	40.5	**44.7**
zenotravel (20)	**13**	7	**13**	**13**	1.0	4.9	8.7	**8.8**
Sum (1827)	1156	766	1155	**1159**	86.8	539.8	900.4	**908.7**

Table 1: Coverage and time scores of four SCP variants: offline diversification (*offline*), online computation without diversification (*on-nodiv*), and online diversification without (*on-stable*) and with state reevaluation (*online*). All diversifying variants use a time limit of 1000 seconds, and *on-stable* and *online* use interval selection with i=10K.

	bellman	novelty-1	interval-1	interval-10	novelty-2	interval-100K	interval-1K	interval-100	interval-10K	T=1000s	T=∞
bellman	–	**7**	5	5	4	5	5	4	4	1145	995
novelty-1	**7**	–	4	7	5	3	6	5	3	1153	1125
interval-1	**7**	**5**	–	**4**	5	4	5	4	2	1153	803
interval-10	**7**	**8**	**4**	–	6	4	4	2	3	1154	957
novelty-2	**7**	**6**	**7**	**8**	–	**6**	**4**	3	4	1157	1058
interval-100K	**8**	**6**	**7**	**7**	**6**	–	**5**	**4**	1	1156	1137
interval-1K	**10**	**8**	**9**	**8**	**4**	**5**	–	**4**	2	1157	1106
interval-100	**8**	**8**	**7**	**6**	**5**	**4**	**4**	–	3	1157	1061
interval-10K	**11**	**8**	**8**	**9**	**6**	**4**	**4**	**6**	–	**1159**	1125

Table 2: Left: per-domain coverage comparisons of different state selection strategies. Each variant uses at most 1000 seconds for online diversification. The entry in row r and column c shows the number of domains in which strategy r solves more tasks than strategy c. For each strategy pair we highlight the maximum of the entries (r, c) and (c, r) in bold. Middle: total number of solved tasks with a time limit of 1000 seconds for online diversification. Right: solved tasks without a diversification time limit.

a time limit of 1000 seconds for the online diversification. We see that overall coverage is similar for all interval and novelty variants (1153–1159 solved tasks) and that the Bellman strategy solves fewer tasks in total than the other strategies. We obtain the highest total coverage by selecting every ten thousandth evaluated state (*interval-10K*) and therefore we use this strategy in all other experiments.

Time Limit

The middle and right parts of Table 2 confirm that we need a time limit for the online diversification. For all state selection strategies total coverage decreases when the time limit of 1000 seconds for the diversification is lifted. The coverage loss is higher, the more states we may select for diversification. For example, the coverage of the *novelty-1* variant only decreases by 28 tasks, because the number of selected states is limited by the number of atoms A in the planning task. For *novelty-2* coverage decreases by 99 tasks, because at most $|A|^2$ states can be selected.

Samples

The offline and online diversification algorithms differ in two main respects: for which states they compute orders and which states they use to decide whether to store an order. For both of these decisions, the offline variant uses sample states obtained with random walks, whereas the online variant uses states evaluated during the search. In this subsection, we analyze whether online diversification benefits from considering randomly sampled states.

	samples	both	state	Coverage	Time Score
samples	–	**2**	2	1158	865.5
both	**2**	–	1	1158	865.5
state	**3**	**2**	–	**1159**	**908.7**

Table 3: Comparison of three different online diversification methods. All methods limit the time for diversification to 1000 seconds, compute an order for each ten thousandth evaluated state and reevaluate states before expanding them. They differ in the set of states $\hat{S}$ for which they diversify the set of stored orders Ω. For *samples* the set $\hat{S}$ contains 1000 sample states obtained with random walks before the search. When using the *state* method $\hat{S}$ only contains the currently evaluated state (as in Algorithm 3). The *both* method sets $\hat{S}$ to the union of the samples and the evaluated state. For an explanation of the data, see Table 2.

		1s	10s	100s	1000s	1200s	1500s
Coverage	offline	1056	1145	**1159**	1156	1148	1128
	online	1102	1135	1153	**1159**	1154	1146
Time Score	offline	**791.2**	690.7	420.3	86.8	59.2	25.9
	online	920.6	**929.7**	919.1	908.7	906.3	906.6

Table 4: Coverage and time scores for offline and online diversification using different time limits for diversification. The online variants use the *interval-10K* strategy.

It is unlikely that computing orders for randomly sampled states is preferable to computing orders for states that the search actually evaluates. However, it could be beneficial to use a set of sample states when judging whether an order should be stored. In Table 3, we evaluate this hypothesis by comparing three different choices for the question which states to consider when deciding whether an order should be stored. The data shows that we solve almost the same number of tasks in total and per domain regardless of whether we take into account only a single state, a set of 1000 samples or both. However, storing an order when it improves the heuristic value for the currently evaluated state results in shorter runtimes than for the other two variants in Table 3, which is why we only consider this variant in Algorithm 3 and all other experiments.

Offline vs. Online Diversification

We now evaluate different time limits and compare the resulting algorithms to their offline counterparts. The top part of Table 4 confirms the result from Seipp, Keller, and Helmert (2020) that we cannot simply reduce the time for offline diversification (to 1 or 10 seconds) in order to minimize overall runtime, without sacrificing total coverage. Offline diversification solves the highest number of tasks (1159) with a time limit T of 100 seconds and slightly fewer tasks (1156) with T=1000s. Using lower or higher time limits leads to solving much fewer tasks. The results are similar for

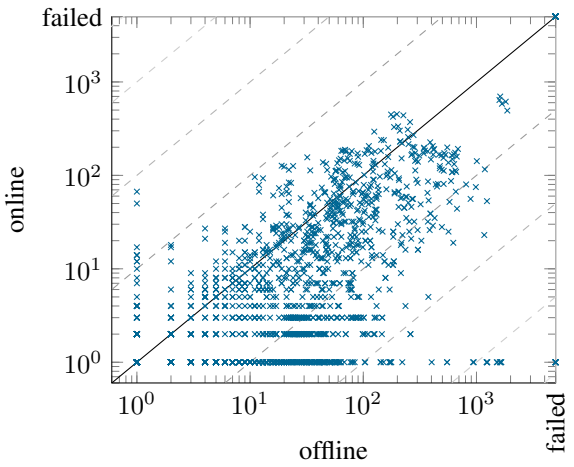


Figure 1: Number of stored orders by offline and online diversification. Both variants use a diversification time limit of 1000 seconds and the online variant uses the interval state selection strategy with i=10K.

online diversification, which solves the most tasks (1159) for T=1000s and slightly fewer tasks (1153–1154) for T=100s and T=1200s. Online diversification is less susceptible to the chosen time limit than offline diversification: while the difference between the maximum and minimum coverage score for offline diversification is 103 tasks, the corresponding value for online diversification is only 57 tasks. Table 1 shows detailed coverage and time score results for the offline and online variants that use at most 1000 seconds for diversification (among two other variants).

Before we analyze the runtimes of the different variants, we compare the number of orders stored by offline and online diversification (using 1000 seconds for diversification) in Figure 1. We can see that the online variant tends to store fewer orders than the offline counterpart, often by more than one order of magnitude. More precisely, online diversification stores fewer orders than offline diversification for 1221 tasks, while the opposite is the case for 296 tasks. For SCP the increased accuracy from using more orders usually outweighs the increased evaluation time (Seipp, Keller, and Helmert 2020). Therefore, Figure 1 suggests that the online diversification stores fewer redundant orders than the offline diversification, because otherwise the coverage gap between the two variants (3 tasks) would be larger.

Not only does online diversification select useful orders and obtain high coverage scores, but it also drastically reduces the overall runtime for many tasks compared to offline diversification. The bottom part of Table 4 reveals that the time score of all online variants is higher than the best time score of all offline variants. The time score gap between the two variants is 129.4 points for T=1s and it grows to 880.7 points for T=1500s.

Figure 2 shows the cumulative number of solved tasks over time by offline and online diversification (with T=1000s) and the variant that computes an SCP heuristic for each evaluated state without storing any orders. The lat-

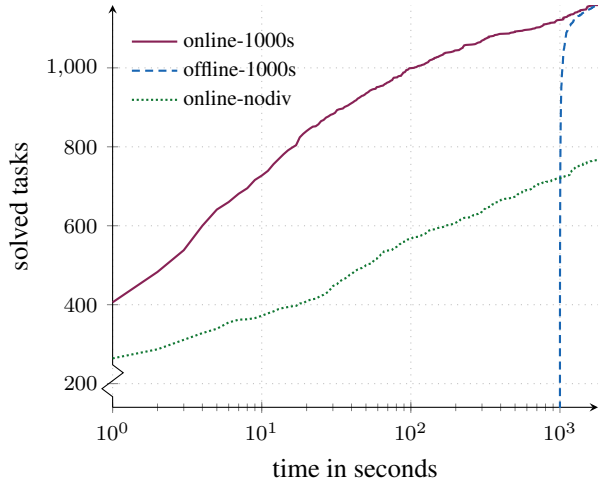


Figure 2: Number of solved tasks over time.

ter variant (online-nodiv) solves the simpler tasks quickly, but only reaches a total coverage of 766 tasks. The offline variant achieves a much higher total coverage (1156 tasks), but it can only start finding solutions after its diversification phase ended.

The online variant with diversification combines the advantages of the other two approaches and achieves both short runtimes and high total coverage (1159 tasks). For example, online-1000s solves 1121 tasks before offline-1000s even finishes the diversification phase. After reaching the diversification time limit, the online and offline variants solve roughly the same number of additional tasks per time step.

For all time limits between 1 and 1800 seconds, online diversification solves more tasks than offline diversification and the online-nodiv variant. The right part of Table 1 holds per-domain time scores for the algorithms in Figure 2. The numbers show that online diversification is faster than offline diversification in all domains, and usually achieves much higher time scores (columns *offline* and *online* in Table 1).

Related Work

The work that is most closely related to ours simultaneously refines a set of Cartesian abstraction heuristics and a set of SCP heuristics over them during an A* search (Eifler and Fickert 2018). Whenever the maximum over the SCP heuristics violates the Bellman optimality equation (1957) for a state s and its successor states, the authors either refine one of the abstractions until the heuristic estimate for s increases, merge two abstractions or compute a new greedy order ω for s (using the h scoring function, Seipp, Keller, and Helmert 2020) and add h^{SCP}_{ω} to the set of SCP heuristics. Their strongest algorithm compares favorably against a version that only refines the abstractions offline and only computes a single SCP heuristic over them. However, both the online and the offline version are outperformed by the version that diversifies a set of SCP heuristics over a *fixed* set of Cartesian abstraction heuristics, i.e., the offline SCP variant we describe in Algorithm 2.

The literature contains additional approaches that improve heuristics online during the search. For example, the SymBA* planner repeatedly switches between a symbolic forward search and symbolic backward searches in one of multiple abstractions (Torralba, Linares López, and Borrajo 2016). In the setting of satisficing planning, Fickert and Hoffmann (2017) refine the FF heuristic (Hoffmann and Nebel 2001) during enforced hill-climbing and greedy best-first searches.

As a final example, Franco and Torralba (2019) interleave the precomputation of a symbolic abstraction heuristic and the symbolic search that uses it, by iteratively switching between the two phases. In each round they double the amount of time given to each phase. Our work is orthogonal to theirs since the two approaches focus on interleaving two different types of precomputation with the search.

Conclusions

The best previously-known method for computing diverse SCP heuristics uses a fixed amount of time for sampling states and computing SCP heuristics for them. It yields strong heuristics, but needs a long precomputation phase. Computing an SCP heuristic for each evaluated state yields even better estimates and needs no precomputation phase, but it greatly slows down the search. We showed that by *diversifying* SCP heuristics *online*, we can combine the strengths of both approaches and obtain an algorithm that needs no sample states nor precomputation phase, evaluates states quickly and achieves high coverage.

Currently, the strongest optimal classical planners compute multiple cost partitionings over abstraction heuristics and use them in an A* search. There are three steps that can take long before these planners can start their search: deciding which abstractions to build (i.e., pattern selection for pattern database heuristics), building the abstractions and computing orders for cost partitioning algorithms. Franco and Torralba (2019) show how to interleave the search with building an abstraction and our paper shows how to efficiently compute orders online. It will be interesting to see how we can decide during the search which abstractions to build and how we can combine all of these techniques.

Acknowledgments

We thank Malte Helmert and the anonymous reviewers for their insightful comments. We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639).

References

Bellman, R. E. 1957. *Dynamic Programming*. Princeton University Press.

Eifler, R.; and Fickert, M. 2018. Online Refinement of Cartesian Abstraction Heuristics. In Bulitko, V.; and Storandt, S., eds., *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*, 46–54. AAAI Press.

Fickert, M.; and Hoffmann, J. 2017. Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 107–115. AAAI Press.

Franco, S.; and Torralba, Á. 2019. Interleaving Search and Heuristic Improvement. In Surynek, P.; and Yeoh, W., eds., *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS 2019)*, 130–134. AAAI Press.

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.; Bonet, B.; and Geffner, H. 2005. New Admissible Heuristics for Domain-Independent Planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, 1163–1168. AAAI Press.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over Multiple Pattern Databases Speeds up Heuristic Search. *Artificial Intelligence* 170(16–17): 1123–1136.

Karpas, E.; Katz, M.; and Markovitch, S. 2011. When Optimal Is Just Not Good Enough: Learning Fast Informative Action Cost Partitionings. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 122–129. AAAI Press.

Katz, M.; and Domshlak, C. 2008. Optimal Additive Composition of Abstraction-based Admissible Heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 174–181. AAAI Press.

Katz, M.; and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence* 174(12–13): 767–798.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In De Raedt, L.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P., eds., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, 540–545. IOS Press.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the Most Out of Pattern Databases for Classical Planning. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2357–2364. AAAI Press.

Seipp, J. 2018. *Counterexample-guided Cartesian Abstraction Refinement and Saturated Cost Partitioning for Optimal Classical Planning*. Ph.D. thesis, University of Basel.

Seipp, J. 2020. Code, benchmarks and experiment data for the HSDIP 2020 paper "Online Saturated Cost Partitioning for Classical Planning". https://doi.org/10.5281/zenodo.4068173.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research* 62: 535–577.

Seipp, J.; and Helmert, M. 2019. Subset-Saturated Cost Partitioning for Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 391–400. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2017. A Comparison of Cost Partitioning Algorithms for Optimal Classical Planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 259–268. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research* 67: 129–167.

Seipp, J.; Pommerening, F.; and Helmert, M. 2015. New Optimization Functions for Potential Heuristics. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 193–201. AAAI Press.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461. doi:10.5281/zenodo.790461. URL https://doi.org/10.5281/zenodo.790461.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2016. Abstraction Heuristics for Symbolic Bidirectional Search. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3272–3278. AAAI Press.

Subset-Saturated Transition Cost Partitioning for Optimal Classical Planning

Dominik Drexler and **David Speck** and **Robert Mattmüller**
University of Freiburg, Germany
{drexlerd, speckd, mattmuel}@informatik.uni-freiburg.de

Abstract

Cost partitioning admissibly combines the information from multiple heuristics for state-space search. We use a greedy method called saturated cost partitioning that considers the heuristics in sequence and assigns the minimal fraction of the remaining costs that it needs to preserve the heuristic estimates. In this work, we address the problem of using more expressive transition cost functions with saturated cost partitioning to obtain stronger heuristics. Our contribution is subset-saturated transition cost partitioning that combines the concepts of using transition cost functions and prioritizing states that look more important during the search. Our empirical evaluation shows that this approach still causes too much computational overhead but leads to more informed heuristics.

Introduction

Heuristic search with a lower bounding heuristic is one of the most promising techniques to solve challenging planning problems optimally. We consider cost partitioned heuristics that combine information from multiple sources. Cost partitioned heuristics use the theory of cost partitioning (Katz and Domshlak 2008). A cost partitioning distributes each transition's cost over multiple heuristics such that the sum does not exceed the original cost. If each heuristic is admissible for the assigned transition costs, then the sum of heuristic estimates is admissible. The most effective approach to compute cost partitioned heuristics is saturated cost partitioning (Seipp, Keller, and Helmert 2017). Saturated cost partitioning considers the heuristics in sequence and assigns the minimum fraction of the remaining costs that it needs to preserve the heuristic estimates. Recently, saturated cost partitioning with restrictive operator cost functions yielded state-of-the-art performance (Seipp and Helmert 2019).

In this work, we address the problem of using more expressive transition cost functions with saturated cost partitioning to obtain stronger heuristics. Transition cost functions require computationally more demanding representations compared to operator cost functions but allow computing more informed cost partitioned heuristics. The baseline approach is saturated transition cost partitioning by Keller et al. (2016). It uses the minimal fraction of the remaining cost required to preserve the heuristic estimates, which often results in an increased computational effort but also in more informed heuristics.

We can decrease the computational effort by considering a larger solution set. Moving from operator to transition cost functions typically already increases the solution set because every operator cost function is a transition cost function (but not vice versa). Another way of increasing the size of the solution set is by preserving the heuristic estimates of a subset of states. The concepts of preserving the heuristic estimates of a subset of states were introduced with subset-saturated (operator) cost partitioning on restrictive operator cost functions (Seipp and Helmert 2019). Preserving the heuristic estimates of a subset of states has shown to yield more accurate cost partitioned heuristics.

Our contribution is subset-saturated transition cost partitioning that combines saturated transition cost partitioning with the concepts of preserving the heuristic estimates of a subset of states. Even though subset-saturated transition cost partitioning does not necessarily compute better cost partitioned heuristics, we provide an empirical analysis to show that this is often the case in practice. We derive a mechanism for selecting transition cost functions from the solution set that trades heuristic accuracy with performance and follows the principle of prioritizing a subset of states.

Background

In this chapter we define, describe and discuss the concepts and ideas that form the foundation for subset-saturated transition cost partitioning.

Planning Tasks

We consider the SAS^+ planning formalism (Bäckström and Nebel 1995). A SAS^+ planning task is a 5-tuple that consists of a set of finite domain variables that induce a set of states, a finite set of operators (or actions) that induce a finite set of transitions, an initial state, a goal condition and a function that describes the cost of applying each operator. Each planning task compactly encodes a transition system with weights assigned to transitions.

Transition Systems

A transition system describes the dynamics of a state-based system. Transition systems (Seipp and Helmert 2019) are also called state spaces.

Definition 1 (Transition System)**.** *A Transition System $\mathcal{T}$ is a directed, labeled graph defined by a finite set of states $S(\mathcal{T})$, a finite set of labels $L(\mathcal{T})$, a finite set $T(\mathcal{T})$ of labeled transitions $s \xrightarrow{l} s$ with $s, s' \in S(\mathcal{T})$ and $l \in L(\mathcal{T})$, an initial state $s_0(\mathcal{T}) \in S(\mathcal{T})$ and a set $S_\star(\mathcal{T}) \subseteq S(\mathcal{T})$ of goal states.*

The objective in state space search is to find paths from the initial state to a goal state.

Definition 2 (Paths and Goal Paths (Seipp and Helmert 2019))**.** *Let $\mathcal{T}$ be a transition system. A path from $s \in S(\mathcal{T})$ to $s' \in S(\mathcal{T})$ is a sequence of transitions from $T(\mathcal{T})$ of the form $\pi = \left\langle s^0 \xrightarrow{l_1} s^1, \ldots, s^{n-1} \xrightarrow{l_n} s^n \right\rangle$, where $s^0 = s$ and $s^n = s'$. The length of π denoted by $|\pi|$, is n. The empty path (of length 0) is permitted if $s = s'$. π is called a goal path, if it is a path to any goal state $s' \in S_\star(\mathcal{T})$.*

In the context of classical planning and cost partitioning, it is convenient to allow assignment of costs to transitions that are not necessarily unit costs. Transition cost functions allow a different assignment for each transition and therefore, allows taking the application contexts of each action into account.

Definition 3 (Transition Cost Function)**.** *Let $\mathcal{T}$ be a transition system with transitions $T(\mathcal{T})$. A transition cost function for $\mathcal{T}$ is a function $tcf : T(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$ that assigns costs to transitions. A transition cost function is finite if $-\infty < tcf(t) < \infty$ for all transitions $t \in T(\mathcal{T})$. It is nonnegative if $0 \leq tcf(t)$ for all transitions $t \in T(\mathcal{T})$. We write $\mathcal{C}(T(\mathcal{T}))$ for the set of all transition cost functions for $\mathcal{T}$ and $\mathcal{C}_{\geq 0}(T(\mathcal{T}))$ for the set of all nonnegative transition cost functions for $\mathcal{T}$.*

A special case of a transition cost function is an operator cost function. It is a transition cost function where every transition with the same label is assigned the same cost value. Therefore, an operator cost function is a mapping from labels (or operators) to our considered codomain. The representation of an operator cost function requires worst-case space of $\Theta(|L(\mathcal{T})|)$ or $\Theta(||\Pi||)$ where $||\Pi||$ denotes the input size of a planning task. The representation of a transition cost function requires worst-case space of $\Theta(||\Pi|| \cdot 2^{poly(||\Pi||)})$.

In the context of cost partitioning, it is convenient to work with a collection of transition systems where each transition system is associated with a transition cost function. The definition of a weighted transition system captures the notion of this pairing (Seipp and Helmert 2019).

Definition 4 (Weighted Transition System)**.** *A weighted transition system is a tuple $\langle \mathcal{T}, tcf \rangle$ where $\mathcal{T}$ is a transition system and tcf is a transition cost function for $\mathcal{T}$.*

Allowing infinities to be assigned to transitions through either operator or transition cost functions means that we must take care of arithmetic expressions that involve $+\infty$ and $-\infty$ to make the theory of cost partitioning work. Seipp and Helmert (2019) defined two kinds of addition that make it possible to handle mixed infinities in cost partitioning.

The symbols $+$ (infix) and $\sum$ (prefix) denote the left-addition operation. Left-addition over integers is the usual addition. Expressions that contain infinities are defined as $\infty + x = \infty$ and $-\infty + x = -\infty$ for all integers x, including x being ∞ or $-\infty$. This operation is associative but not commutative. Intuitively, a left addition sum that contains mixed infinities evaluates to the leftmost infinity. In cost partitioning, we use left-addition for summing up multiple heuristic values and partitioned costs (Seipp and Helmert 2019).

The symbols $\oplus$ (infix) and $\bigoplus$ (prefix) denote the path-addition operation. Path addition over integers is the usual addition. Expressions that contain infinities are defined as $x \oplus y = \infty$ iff $x = \infty$ or $y = \infty$ and $x \oplus (-\infty) = -\infty$ for all $x \neq \infty$. This operation is associative and commutative. Intuitively, a path addition sum evaluates to $+\infty$ if the path addition sum involves at least one $+\infty$. We use path-addition to define the cost of a path in a transition system (Seipp and Helmert 2019).

Definition 5 (Cost of a path)**.** *Let $\langle \mathcal{T}, tcf \rangle$ be a weighted transition system. The cost of a path $\pi = \left\langle s^0 \xrightarrow{l_1} s^1, \ldots, s^{n-1} \xrightarrow{l_n} s^n \right\rangle$ with $t_i = s^{i-1} \xrightarrow{l_i} s^i$ is defined as $cost(tcf, \pi) = \bigoplus_{i=1}^{n} tcf(t_i)$.*

Intuitively, a path of cost $-\infty$ is infinitely cheap, and a path of cost ∞ is non-existent.

In optimal classical planning, we are interested in finding paths with the cheapest cost to a goal. The following definition captures goal distances as functions that depend on a given state and a transition cost function. This notion follows the pairing of weighted transition systems in cost partitioning.

Definition 6 (Goal Distances and Optimal Paths (Seipp and Helmert 2019))**.** *Let $\langle \mathcal{T}, tcf \rangle$ be a weighted transition system. The goal distance of a state $s \in S(\mathcal{T})$ in $\mathcal{T}$ under cost function tcf is defined as $\inf_{\pi \in \Pi_\star(\mathcal{T}, s)} cost(tcf, \pi)$ where $\Pi_\star(\mathcal{T}, s)$ is the set of goal paths from s in $\mathcal{T}$.*

We write $h^*_{\mathcal{T}}(tcf, s)$ for the goal distance of s in $\mathcal{T}$ under transition cost function tcf.

A goal path π from s in $\mathcal{T}$ is optimal under the given transition cost function tcf if $cost(tcf, \pi) = h^*_{\mathcal{T}}(tcf, s)$.

The empty infimum is defined as ∞ and follows the notion of a non-existent goal path. The infimum ensures that repeatedly taking a negative cost cycle in the transition system will evaluate to $-\infty$.

Abstractions

Abstractions are relaxations of the behavior of a state-based system where multiple states collapse into a single abstract state (Helmert, Haslum, and Hoffmann 2007).

Definition 7 (Abstraction)**.** *Let $\mathcal{T}, \mathcal{T}'$ be two transition systems with the same label sets $L(\mathcal{T}) = L(\mathcal{T}')$ and let $\alpha : S(\mathcal{T}) \to S(\mathcal{T}')$, $\beta : T(\mathcal{T}) \to T(\mathcal{T}')$ be surjective functions. We say that $\mathcal{T}'$ is an abstraction of $\mathcal{T}$ with abstraction mappings α, β if (1) $\alpha(s_0(\mathcal{T})) = s_0(\mathcal{T}')$, (2) $\alpha(s_\star) \in$*

$S_\star(\mathcal{T}')$ *for all* $s_\star \in S_\star(\mathcal{T})$*, and (3)* $\alpha(s) \xrightarrow{l} \alpha(s') \in T(\mathcal{T}')$ *and* $\beta(s \xrightarrow{l} s') = \alpha(s) \xrightarrow{l} \alpha(s')$ *for all* $s \xrightarrow{l} s' \in T(\mathcal{T})$.

We refer to $\mathcal{T}$ as the concrete transition system and $\mathcal{T}'$ as the abstract transition system. We consider a special type of abstraction where every abstract state is cartesian (Seipp and Helmert 2013).

An abstraction heuristic is a function that maps each concrete state to the goal distance of its corresponding abstract state under a given transition cost function. Goal distances in the abstract transition system require an abstract transition cost function that maps every abstract transition to a value in the codomain. The abstract transition cost function maps an abstract transition to the minimal cost of every concrete transition that induces it (Keller et al. 2016).

Definition 8 (Abstraction heuristic)**.** *Let* $\langle \mathcal{T}, tcf \rangle$ *and* $\langle \mathcal{T}', tcf' \rangle$ *be two weighted transition systems. Let* $\mathcal{T}'$ *be an abstraction of* $\mathcal{T}$ *with abstraction mappings* α, β*.*

We say that tcf' *is the abstract transition cost function of* tcf *that describes the cost of each abstract transition* $t' \in T(\mathcal{T}')$ *in the abstraction with*

$$tcf'(t') = \min\{tcf(t) \mid t \in T(\mathcal{T}) \wedge \beta(t) = t'\}$$

The abstraction heuristic of a concrete state $s \in S(\mathcal{T})$ *is the goal distance of the corresponding abstract state* $\alpha(s)$ *in the abstraction* $\mathcal{T}'$ *under the abstract transition cost function* tcf'*, i.e.* $h(tcf, s) = h^*_{\mathcal{T}'}(tcf', \alpha(s))$.

The definition of the abstract transition cost function reveals that the computation of an abstract transition cost involves a minimization over all (exponentially many) transitions of the concrete transition system that are responsible for this abstract transition. In the case of cartesian abstractions this minimization of exponentially many concrete transitions can be carried out in time that is often polynomial in the number of variables of the planning task, albeit worst-case exponential (Geißer, Keller, and Mattmüller 2016).

A heuristic is admissible if it never overestimates the goal distances in the weighted concrete transition system. Abstraction heuristics are admissible because every goal path in the concrete transition system corresponds to a goal path in the abstract transition system. The minimization in the abstract transition cost function ensures that the abstraction heuristic does not overestimate any goal distance (Keller et al. 2016). We use the A^* algorithm with an admissible heuristic to find optimal goal paths (Hart, Nilsson, and Raphael 1968).

Transition Cost Partitioning

A transition cost partitioning splits a given transition cost function into a sequence of transition cost functions such that the sum (left addition) of all transition cost functions in the sequence is upper bounded by the original transition cost function (Keller et al. 2016; Pommerening 2017).

Definition 9 (Transition Cost Partitioning)**.** *A transition cost partitioning for a weighted transition system* $\langle \mathcal{T}, tcf \rangle$ *with transition cost function* $tcf \in \mathcal{C}(T(\mathcal{T}))$ *is a tuple* $\langle tcf_1, \ldots, tcf_n \rangle \in \mathcal{C}(T(\mathcal{T}))^n$ *whose sum is bounded by* tcf*, i.e.* $\sum_{i=1}^n tcf_i(t) \leq tcf(t)$ *for all* $t \in T(\mathcal{T})$.

A transition cost partitioning induces a cost partitioned heuristic by associating each transition cost function with a heuristic and summing up the heuristic estimates of individual states. The following theorem states that the cost partitioned heuristic is admissible. Katz and Domshlak (2008) introduced cost partitioning that works on nonnegative operator cost functions. Pommerening et al. (2015) showed that general operator cost functions can be used for cost partitioning and Keller et al. (2016) defined general transition cost function for cost partitioning. Finally, Seipp and Helmert (2019) introduced left addition rules to handle mixed infinities, which led to Theorem 1 and shows that cost partitioned heuristics are admissible.

Theorem 1. *Admissibility Let* $\mathcal{T}$ *be a transition system with admissible heuristics* $\langle h_1, \ldots, h_n \rangle$ *and a transition cost partitioning* $P(T(\mathcal{T})) = \langle tcf_1, \ldots, tcf_n \rangle \in \mathcal{C}(T(\mathcal{T}))^n$*. Then* $h_{P(T(\mathcal{T}))}(s) = \sum_{i=1}^n h_i(tcf_i, s)$ *is an admissible heuristic.*

An optimal transition cost partitionings for a set of heuristics is a transition cost partitioning that provides the best heuristic estimate for a given state. The best known algorithm to compute optimal transition cost partitioning works in exponential time and is not useful in practice. In the next section, we focus on greedy algorithms based on cost saturation that consider the heuristic in sequence and builds the transition cost partitioning sequentially.

Subset-Saturated Transition Cost Partitioning

In this section, we combine saturated transition cost partitioning (Keller et al. 2016) with subset-saturation known from subset-saturated operator cost partitioning (Seipp and Helmert 2019). We first generalize dominating cost functions (Seipp and Helmert 2019).

Definition 10 (Dominating Transition Cost Function)**.** *Consider two transition cost functions* tcf *and* tcf' *defined on the same set of transitions. We say that* tcf *dominates* tcf'*, in symbols* $tcf \leq tcf'$*, if* $tcf(t) \leq tcf'(t)$ *for all transitions* t*.*

We say that tcf *is the unique minimum of a set of transition cost functions* $Cost$ *if it dominates all transition cost functions in* $Cost$*. (Not all sets* $Cost$ *have a unique minimum.)*

A saturated transition cost function for a subset of states dominates a given transition cost function and preserves the heuristic estimates of a given subset of states (Seipp and Helmert 2019).

Definition 11 (Saturated Transition Cost Function)**.** *Consider a weighted transition system* $\langle \mathcal{T}, tcf \rangle$*, a set of states* $S' \subseteq S(\mathcal{T})$ *and a heuristic* h *for* $\mathcal{T}$*. A transition cost function* $stcf \in \mathcal{C}(T(\mathcal{T}))$ *is saturated for* S', h *and* tcf *if*

1. $stcf \leq tcf$ *and*
2. $h(stcf, s) = h(tcf, s)$*, for all states* $s \in S'$.

A saturated transition cost function always exists because tcf itself is a saturated transition cost function. Our definition of the saturated transition cost function allows selecting from a typically larger set of possible saturated transition

cost functions because of allowing transition cost functions instead of operator cost functions and preserving the heuristic estimates of a subset of states. We formalize this selection mechanism with a generalization of operator saturators (Seipp and Helmert 2019) to transition saturators. A transition saturator is a function that takes as an input a heuristic, the remaining transition cost function, and a subset of states and outputs a saturated transition cost function. In contrast, an operator saturator only allows operator cost functions in the input and output.

Definition 12 (Transition Saturator)**.** *Consider a transition system $\mathcal{T}$, a set of states $S' \subseteq S(\mathcal{T})$ and a heuristic h for $\mathcal{T}$.*

A transition saturator for S' and h is a partial function $saturate : \mathcal{C}(T(\mathcal{T})) \to \mathcal{C}(T(\mathcal{T}))$ such that whenever $saturate(tcf)$ is defined, it is a saturated transition cost function for S', h and tcf.

A transition saturator is general if its domain of definition is $\mathcal{C}(T(\mathcal{T}))$. It is nonnegative to general (NNG) if its domain of definition is $\mathcal{C}_{\geq 0}(T(\mathcal{T}))$. It is nonnegative if its domain of definition is $\mathcal{C}_{\geq 0}(T(\mathcal{T}))$ and it only produces transition cost functions in $\mathcal{C}_{\geq 0}(T(\mathcal{T}))$.

The following definition generalizes subset-saturated operator cost partitioning by exchanging operator saturators with transition saturators. Alternatively speaking, we parameterize saturated transition cost partitioning with transition saturators and allow saturation for a subset of states.

Definition 13 (Subset-Saturated Transition Cost Partitioning)**.** *Consider a weighted transition system $\langle \mathcal{T}, tcf \rangle$, a set of states $S' \subseteq S(\mathcal{T})$, a nonempty sequence of heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ for $\mathcal{T}$ and a sequence $Saturate = \langle saturate_1, \ldots, saturate_n \rangle$ such that $saturate_i$ is a saturator for $S' \subseteq S(\mathcal{T})$ and h_i for all $1 \leq i \leq n$.*

The saturated transition cost partitioning $\langle tcf_1, \ldots, tcf_n \rangle$ of the transition cost function tcf induced by $Saturate$ is defined as:

$$
\begin{aligned}
remain_0 &= tcf \\
tcf_i &= saturate_i(remain_{i-1}) \text{ for all } 1 \leq i \leq n \\
remain_i &= remain_{i-1} - tcf_i \text{ for all } 1 \leq i \leq n
\end{aligned}
$$

The subtraction in the definition of $remain_i$ is defined in terms of left addition, i.e., $a - b := a + (-b)$ and corresponds the definition from subset-saturated operator cost partitioning.

The saturated transition cost partitioning (Definition 13) is a transition cost partitioning. The result follows from the proof that subset-saturated operator cost partitioning produces operator cost partitionings and exchanges labels with transitions (Seipp and Helmert 2019).

In general, subset-saturated cost partitioning has three major choice points that influence the accuracy of the cost partitioned heuristic. These are the set of heuristics, the order of the heuristics, and the transition saturators. In the remaining part of this section, we define generalizations of the four operator saturators and an additional transition saturator that allows avoiding computations of abstract transition weights (Definition 8).

When comparing transition saturators, we use the same concept of domination between saturated cost functions. A dominating saturated cost function is "more economical" than the cost function that it dominates because it achieves the same objective of preserving heuristic estimates but leaves a larger fraction of the remaining costs for later heuristics in sequence. This often gives better heuristic estimates but is not guaranteed due to the greediness of saturated cost partitioning (Seipp and Helmert 2019).

We generalize the comparison results of operator saturators to allow for comparison of transition saturators. We provide an additional result for comparing operator saturators with transition saturators.

Theorem 2 (Domination by Subsets)**.** *For a given transition system $\mathcal{T}$, heuristic h for $\mathcal{T}$ and transition cost function $tcf \in \mathcal{C}(T(\mathcal{T}))$, let $STCF(X)$ be the set of saturated transition cost functions for the set of states $X \subseteq S(\mathcal{T})$, h and tcf.*

Let $S'' \subseteq S' \subseteq S(\mathcal{T})$. Then:

1. *For all transition cost functions $tcf' \in STCF(S')$, there exists a transition cost function $tcf'' \in STCF(S'')$ that dominates tcf'.*
2. *If a transition cost function $tcf'' \in STCF(S'')$ is the unique minimum of $STCF(S'')$, then tcf'' dominates all transition cost functions in $STCF(S')$.*

Proof: Statement 1 follows from Definition 11 where we allow assigning lower saturated costs to transitions that start at or end at states outside the subset if it does not conflict with preserving the heuristic estimates of states within the subset. Such transitions may exist exclusively in S'', because $S'' \subseteq S'$. Statement 2 follows from Definition 10 of a unique minimum and the first statement. ∎

The theorem is an analog extension of the theorem about domination by subsets on operator saturators (Seipp and Helmert 2019). It shows that transition saturators for $S'' \subseteq S'$ are more economical than transition saturators for S'. Since sets of saturated transition cost functions do not necessarily have a unique minimum, it is not guaranteed that a minimal saturated transition cost function for $STCF(S'')$ dominates all saturated transition cost functions in $STCF(S')$. This stronger notion of dominance requires that $STCF(S'')$ has a unique minimum and is part 2 of the theorem.

The second way to obtain more economical transition saturators uses saturator composition where the output of a saturator is applied to the input of another saturator (Seipp and Helmert 2019).

Theorem 3 (Domination by Composition)**.** *Let $saturate_1$ and $saturate_2$ be transition saturators for the same transition system $\mathcal{T}$, state set $S' \subseteq S(\mathcal{T})$ and heuristic h. Let $saturate_{12} : \mathcal{C}(T(\mathcal{T})) \to \mathcal{C}(T(\mathcal{T}))$ be the composition of these saturators, i.e. $saturate_{12}(tcf) = saturate_2(saturate_1(tcf))$ for all transition cost functions $tcf \in \mathcal{C}(T(\mathcal{T}))$.*

Then $saturate_{12}$ is a transition saturator for $\mathcal{T}$, S' and h, and for all transition cost functions $tcf \in \mathcal{C}(T(\mathcal{T}))$, $saturate_{12}(tcf)$ dominates $saturate_1(tcf)$.

Proof: Follows directly from Definition 12, where we require the output of a transition saturator to dominate its input. ∎

The composition is the reason why we consider saturators in the general case. The inner saturator can output negative costs that the outer saturator has to handle. The composition with $saturate_2(\max(0, saturate_1(tcf)))$ ensures that $saturate_2$ is considered in the NNG case. This does not violate Definition 11 statement 1 because tcf is the remaining cost function and nonnegative. The following theorem states that allowing saturators to output transition cost functions makes them more economical.

Theorem 4 (Domination by Expressiveness)**.** *Let $saturate_o$ be an operator saturator for transition system $\mathcal{T}$, state set $S' \subseteq S(\mathcal{T})$ and heuristic h. Then there exists a transition saturator $saturate_t$ for $\mathcal{T}$, S', and h such that $saturate_t(ocf)$ dominates $saturate_o(ocf)$ for all operator cost functions $ocf \in \mathcal{C}(L(\mathcal{T}))$.*

Proof: The output of any operator saturator is an operator cost function and a special case of the output of a transition saturator. Hence, there exists a transition saturator that returns the same saturated transition cost function as the operator saturator. ∎

In other words, for each operator saturator, we can construct a transition saturator that produces a dominating saturated transition cost function. In the rest of this chapter, we define such a transition saturator for each known operator saturator. We introduce a new transition saturator that is explicitly used for transition cost functions and improves the performance of computing heuristic estimates. Whenever we provide additional information about the experimental setup, we describe the setup that allows for a fair comparison with operator saturators.

General transition saturators

In the definition of each transition saturator, we consider a weighted concrete transition system $\langle \mathcal{T}, tcf \rangle$ where tcf describes the current remaining transition cost function and an abstraction heuristic h for $\langle \mathcal{T}, tcf \rangle$ with underlying weighted abstract transition system $\langle \mathcal{T}', tcf' \rangle$.

Saturate for all states ($\mathbf{all_t}$) The all_t transition saturator preserves the heuristic estimates of all states and is the one that was considered previously by Keller et al. (2016). It computes the unique minimum saturated transition cost function $mstcf$ by setting the consistency constraint $h(tcf, s) \leq h(tcf, s') + mstcf(t)$ tight for all transitions $t = s \xrightarrow{l} s' \in T(\mathcal{T})$. This can be enforced by setting

$$mstcf(t) = h(tcf, s) \ominus h(tcf, s')$$

Seipp, Keller, and Helmert (2020) defined the operator $\ominus$ in the context of computing the minimum saturated operator cost function $msocf$. The operator $\ominus$ behaves like the regular subtraction in the finite case and handles infinities as $x \ominus y = -\infty$ iff $x = -\infty$ or $y = \infty$ and $x \ominus y = \infty$ iff $x = \infty \neq y$ or $x \neq -\infty = y$. The minimum saturated operator cost function $msocf$ sets the consistency constraint tight for at least one transition of each operator, i.e.,

$$msocf(l) = \sup_{s \xrightarrow{l} s' \in T(\mathcal{T})} h(tcf, s) \ominus h(tcf, s')$$

where the empty supremum is defined as $-\infty$ and tcf is an operator cost function in saturated operator cost partitioning. Seipp, Keller, and Helmert (2020) show that the operator $\ominus$ computes the minimal saturated cost for each transition and the supremum generalizes over all transitions with the same label such that context information does not need to be tracked. Hence, the $mstcf$ computes the unique minimum among all saturated transition cost functions that preserve the heuristic estimates of all states.

Saturate for all states ($\mathbf{spd_t}$) The transition saturator spd_t is nonnegative and preserves the heuristic estimates of all states. Its name is derived from the Shortest Path Discovery (SPD) problem (Szepesvári 2004), where we are given a transition system, a function $query$ that returns the cost of a transition, and a lower bound on the true transition weights. The objective is to find the exact goal distance of each state[1], such that the number of evaluations of the function $query$ is as small as possible.

The SPD problem occurs in the saturated transition cost partitioning algorithm as follows: For the next abstraction heuristic in sequence, we do not know the abstract transition weights. But we can compute their weights using Definition 8. Our experiments have shown that computing all abstract transition weights is a performance bottleneck. According to Definition 13, we know that $remain_i$ is nonnegative if $remain_0$ is nonnegative (as in classical planning). Therefore, a lower bound for each abstract transition weight is zero[2].

A nonnegative lower bound is important because it allows using Dijkstra's algorithm for goal distance analysis. The lower bound can avoid the computation of the exact abstract transition weight if it does not shorten goal distances during goal distance analysis. Consider the case that Dijkstra's expands an abstract transition. If the lower bound on the abstract transition weight decreases the currently known goal distance of the source state, then we have to compute the exact transition weight. Otherwise, we keep the lower bound and proceed with the next abstract transition. The lower bound in the else case does not introduce shortcuts because: If the goal path that uses the transition with the lower bound on the cost is not the current cheapest path for the source state, then the same path that uses the exact transition weight is also not the current cheapest path for the source state.

Saturate for reachable states ($\mathbf{reach_t}$) The transition saturator $reach_t$ is a generalization of the operator saturator

[1] s-t-path in the original problem definition

[2] It is possible to extract a more accurate lower bound, i.e., an operator cost function of the remaining transition cost function in $\mathcal{O}(|L(\mathcal{T})|)$ time when using decision diagrams.

reach$_o$ (Seipp and Helmert 2019). It preserves the heuristic estimates of all states that are reachable in a forward search. The concrete preimage S' of all states that are reachable in the abstract transition system overapproximates the set of reachable states. The set of unreachable states is $S(\mathcal{T}) \setminus S'$. We set the heuristic estimate of each state $s \in S(\mathcal{T}) \setminus S'$ to $h(tcf, s) = -\infty$ because they are never visited in a forward search (Seipp and Helmert 2019). These modified heuristic estimates will remain for all subsequent saturators in a composition to exclude unreachable states from the subset of states. Finally, apply *all*$_t$ on the modified heuristic estimates to obtain the unique minimum saturated transition cost function for S'.

Saturate for a perimeter (perim$_t$) The transition saturator *perim*$_t$ is a generalization of the operator saturator *perim*$_o$ (Seipp and Helmert 2019) and preserves the heuristic estimates of all states that are within a perimeter of k to a goal. The idea is that it is more important to preserve heuristic estimates of states that are closer to a goal (e.g., Holte et al., 2004; Torralba, Linares López, and Borrajo, 2018). The set of states within perimeter $k > 0$ is $S_k = \{s \in S(\mathcal{T}) \mid h(tcf, s) \leq k\}$. In the abstract transition system $\mathcal{T}'$ the set of states S_k corresponds to all abstract states $s \in \mathcal{T}'$ with $h^*_{\mathcal{T}'}(tcf', s) \leq k$.

To efficiently compute a saturated transition cost function, we only allow for nonnegative transition cost functions in the input and cap the heuristic estimates at $k = h(tcf, s_{\mathcal{I}})$ where $s_{\mathcal{I}}$ is the initial state (Seipp and Helmert 2019). Finally, apply *all*$_t$ on the capped heuristic estimates to obtain a saturated transition cost function for S_k.

Saturate for a single state (lp$_t$) The transition saturator *lp*$_t$ is a generalization of the operator saturator *lp*$_o$ (Seipp and Helmert 2019) and preserves the heuristic estimate of only a single state $s \in S(\mathcal{T})$. The set of possible saturated transition cost functions to pick from does not have a unique minimum. We use an adapted version of the linear programming formulation of the *lp*$_o$ operator saturator to choose from the set of possible saturated transition cost functions. The LP uses the same LP-trick to encode the heuristic estimates from the saturated transition cost function.

$$H_a \leq 0 \qquad \text{for all } a \in S_\star(\mathcal{T}') \tag{1}$$

$$H_a \leq C_t + H_b \text{ for all } a \xrightarrow{l} b = t \in T(\mathcal{T}') \tag{2}$$

$$C_t \leq tcf'(t) \quad \text{for all } t \in T(\mathcal{T}') \tag{3}$$

$$C_t \leq C_l \qquad \text{for all } a \xrightarrow{l} b = t \in T(\mathcal{T}') \tag{4}$$

$$H_{\alpha(s)} = h(s) \tag{5}$$

The variables C_t encode the saturated transition cost function, the variables C_l encode the saturated operator cost function, and the variables H_a encode the heuristic estimate $h^*_{\mathcal{T}'}$. We also choose the objective of minimizing the sum of used operator costs $\min \sum_{l \in L(\mathcal{T})} C_l$ where we only include labels with finite saturated operator costs. Our LP formulation differs from the LP formulation of the operator saturator *lp*$_o$ by allowing transition cost function in the input and the output.

The objective function requires a preprocessing, where we compute the saturated transition cost of all transitions that result in either ∞ or $-\infty$. We first set the heuristic estimates of every state in the preimage of each abstract state that is unreachable from the abstract state $\alpha(s)$ to $-\infty$ (similar idea as described in *reach*$_t$). Then, we apply $mstcf$ to all transitions t that evaluate to $mstcf(t) \in \{-\infty, \infty\}$. Finally, we restrict the LP to contain only abstract states with finite heuristic estimates together with their incident abstract transitions. If there is an abstract transition with label l and a constraint of type 2, then this label l is part of the objective function. After solving the LP, we extract the saturated cost of every transition from the variable C_t of the corresponding abstract transition.

In our experiments, we preserve the heuristic estimate of the initial state.

Nonnegative transition saturators

We obtain a nonnegative transition saturator by applying $stcf(t) = \max(0, stcf(t))$ for each transition $t \in T(\mathcal{T})$ on the output $stcf$ of the NNG transition saturator. We denote the nonnegative transition saturator with an additional superscript, e.g., *perim*$_t^+$ for the nonnegative transition saturator of the transition saturator *perim*$_t$. It is possible to obtain similar theoretical comparison results on nonnegative transition saturators.

Selecting runtime efficient saturator outputs

A problem with the current definitions of our transition saturators is that their output does often not perform well in our experiments. The runtime of the subtraction in Definition 13 and the size of the representation of $remain_i$ depends on the saturated transition cost function. Therefore, selecting a saturated transition cost function that performs better in the tradeoff between leaving more remaining costs for subsequent heuristics and lowering the computational overhead is necessary.

The first solution idea is to generalize saturated transition costs over operators as described in the minimum saturated operator cost function (section *all*$_t$). If we do this for all operators, then we obtain subset-saturated operator cost partitioning. Operator costs can be subtracted efficiently from a transition cost function when using decision diagrams.

The second solution idea is to decrease the number of transitions for which the saturator output $stcf(t) \neq 0$ because subtraction of zero is trivial. We can achieve this by considering the nonnegative case or considering other subsets of states. Purely considering the nonnegative case is not an option because negative costs have shown to make the heuristics stronger (Pommerening et al. 2015).

In our experiments, we use the following mechanism to select a saturated transition cost function: If an operator o holds that the saturated transition cost function assigns the same value to every transition with label o, then we use the first solution idea because the same remaining cost is available for subsequent heuristics. Otherwise, we use a weaker notion of the second solution idea where we replace each

$stcf(t) = -\infty$ by 0. We found it prohibitively expensive in our experiments to saturate with $stcf(t) = -\infty$. The saturated transition cost $stcf(t) = -\infty$ typically occurs in dead end states or unreachable states. Intuitively speaking, this replacement corresponds to focus the computational effort on the subset of states S'. It preserves the same heuristic estimates but leaves fewer remaining costs for subsequent heuristics. We denote transition saturators that use this selection mechanism with additional superscript r, e.g. $reach_t^r$.

The proposed solution ideas enable many options to design selection mechanisms, which we leave open for future work.

Negative costs in saturator inputs

We conclude the section of transition saturators with a discussion of the concern about the heuristic reevaluation under different types of transition cost functions. Transition saturators also require the ability to reevaluate the heuristic under the given input transition cost function. The first reevaluation under $remain_i$ is generally not too costly because $remain_i$ is always nonnegative if the cost function given with the planning task is nonnegative (as in classical planning). However, if a transition saturator saturates for a subset of states S' and we later reevaluate the heuristic for states $s \notin S'$, then this requires algorithms like Bellman-Ford if the saturated transition cost function contains negative costs (Seipp and Helmert 2019).

In contrast to the operator saturators, a transition saturator has the ability to tighten the consistency constraint set to each transition. In this case, a heuristic does not change its estimates after reevaluation under the saturated transition cost function. Therefore, we can directly extract the heuristic from the output of the saturator. However, when selecting a saturated transition cost function that does not tighten each consistency constraint, then the heuristic might change after reevaluation. Seipp and Helmert (2019) found it prohibitively expensive to reevaluate the heuristic with Bellman-Ford and they suggest directly extracting a heuristic lower bound from the output of the saturator, trading heuristic accuracy with performance.

Experiments

We implemented all transition saturators into the Fast Downward planning system (Helmert 2006). Similar to modern symbolic search planners, we utilize decision diagrams for compact representation and computation of sets of states (Kissmann and Edelkamp 2014; Torralba et al. 2014; Speck, Geißer, and Mattmüller 2018). More specifically, we use the CUDD library (Somenzi 1994) for Binary Decision Diagrams with the default fast-downward variable order to represent and compute transition cost functions. We conducted experiments with the Downward Lab toolkit (Seipp et al. 2017) on Intel Xeon E5-2650v2 2.60GHz processors with 64GB DDR3 1866MHz ECC registered memory. The benchmark set consists of all 1827 instances from the optimization track of the 1998-2018 International Planning Competitions that do not have conditional effects. Each task was limited to a single core with 4GB of memory and a time limit of 30 minutes. We consider the same set of abstraction as the initial work on cost saturation with operator saturators (Seipp and Helmert 2019) that consists of all CEGAR abstractions with the goal and landmark diversification technique (Seipp and Helmert 2014), the systematic pattern databases (Pommerening, Röger, and Helmert 2013) and the pattern databases found by hill-climbing (Haslum et al. 2007). We order the abstractions using the static greedy order with the scoring function $\frac{h}{stolen}$, that measures how well a heuristic balances the two objectives of having a high heuristic estimate and stealing low costs from other heuristics (Seipp, Keller, and Helmert 2020). We consider only the single order that is optimized for a high heuristic estimate of the initial state. We optimize the order independent of the transition saturator for a fair comparison. All benchmarks, code, and experimental data have been published online[3].

We write $saturate_1$,$saturate_2$ for the composition $saturate_{12}$ of saturators where the output of $saturate_1$ is applied to the input of $saturate_2$.

We write $saturate_1$+$saturate_2$ to denote that $saturate_2$ is an additional run of cost saturation on the remaining cost after computing a saturated transitioncost partitioning with $saturate_1$. The saturated transition cost functions of both runs form a cost partitioning. Hence, summing up the heuristic estimates of both runs yields an admissible heuristic. Postponing additional runs of cost saturation makes sense if there are potentially unused costs available.

If not explicitly mentioned, a transition saturator composition always uses the transition saturator spd_t first. For example all_t corresponds to spd_t,all_t.

Comparison of transition saturators Table 1a shows the pairwise comparison of a set of transition saturators that we obtain using Theorems 2, 3, 4, and the knowledge from previous work about operator saturators. We simulate the saturated transition cost partitioning with the transition saturator all_t (without spd_t) and subset-saturated transition cost partitioning and obtain coverage of 974 tasks. Composition with spd_t clearly pays off because all_t has a coverage of 984 tasks and guarantees computing the same heuristics. Selecting the saturated transition cost function that replaces negative infinities by zero all_t^r also pays off because the coverage increases to 1003 tasks, and the estimate of the initial state is worse in only 4 tasks. Preserving the heuristic estimates of reachable state states $reach_t^r$ shows small improvements. Preserving the heuristic estimates of only the initial state lp_t^r+$reach_t^r$ wins the most pairwise comparison for the heuristic estimate of the initial state but solves only 766 tasks. Our best transition saturator is $reach_t^+$,$perim_t^r$+$reach_t^r$ that solves 1016 tasks and wins the second most pairwise comparisons for the heuristic estimate of the initial state. The transition saturator $perim_t$ most effectively improves heuristic accuracy.

Comparison with operator saturators Table 1b shows the pairwise comparison of a set of operator saturators

[3] https://doi.org/10.5281/zenodo.4065414

	all_t (without spd_t)	all_t	all_t^r	reach_t^r	lp_t^r+reach_t^r	perim_t^r+all_t^r	reach_t^+, perim_t^r+reach_t^r
all_t (without spd_t)	-	**0**	**4**	17	36	31	33
all_t	**0**	-	**4**	17	36	32	33
all_t^r	0	0	-	13	34	33	34
reach_t^r	**49**	**49**	**69**	-	30	42	35
lp_t^r+reach_t^r	**370**	**370**	**400**	**397**	-	**137**	**131**
perim_t^r+all_t^r	**444**	**447**	**475**	**468**	68	-	8
reach_t^+, perim_t^r+reach_t^r	**445**	**448**	**477**	**474**	67	**20**	-
Coverage	974	984	1003	1003	766	1014	**1016**

(a) Comparison of transition saturators

	all_o	perim_o+all_o	all_t (without spd_t)	reach_t^+, perim_t^r+reach_t^r
all_o	-	53	153	51
perim_o+all_o	**485**	-	**370**	49
all_t (without spd_t)	**330**	219	-	33
reach_t^+, perim_t^r+reach_t^r	**645**	**384**	**445**	-
Coverage	1035	**1042**	974	1016

(b) Comparison with operator saturators

Table 1: Per-task comparison of the initial h-value for a subset of all saturator compositions. In every pairwise comparison we consider the tasks for which both transition saturators finished computing the initial heuristic estimate. The entry in row r and column c indicates the number of tasks where r returns a transition cost partitioning with a better initial heuristic value than c. Boldface is used to indicate the winner in the pairwise comparison (r, c) and (c, r).

and transition saturators. We simulate saturated operator cost partitioning with the operator saturator all_o. We choose the best operator saturator $perim_o$+all_o for subset-saturated operator cost partitioning. We simulate saturated transition cost partitioning with the transition saturator all_t (without spd_t). We choose the best transition saturator $reach_t^+$,$perim_t^r$+$reach_t^r$ for subset-saturated transition cost partitioning.

Operator saturators solve more tasks compared to transition saturators with 1042 against 1016 solved tasks. This shows that selecting more efficient saturated transition cost functions is crucial for further improvements. The best transition saturator computes significantly fewer heuristics with a lower estimate for the initial state than other saturators (rightmost column). Furthermore, the best transition saturator wins the most pairwise comparisons for the heuristic estimate of the initial state and shows that subset-saturated transition cost partitioning computes more informed heuristics than previous saturators.

Coverage	all_o	perim_o+all_o	all_t (without spd_t)	reach_t^+, perim_t^r+reach_t^r
driverlog (20)	**15**	**15**	13	14
elevators-opt08-strips (30)	20	20	**21**	18
elevators-opt11-strips (20)	17	17	**18**	15
floortile-opt11-strips (20)	**4**	3	0	1
freecell (80)	**65**	**65**	47	47
ged-opt14-strips (20)	15	15	**19**	**19**
miconic (150)	110	110	109	**113**
mprime (35)	**28**	27	**28**	27
nomystery-opt11-strips (20)	**20**	**20**	13	13
parcprinter-08-strips (30)	17	18	16	**19**
parcprinter-opt11-strips (20)	13	14	12	**15**
pipesworld-notankage (50)	22	22	22	**23**
scanalyzer-08-strips (30)	**13**	**13**	10	10
scanalyzer-opt11-strips (20)	**10**	**10**	7	7
snake-opt18-strips (20)	**13**	12	**13**	**13**
tetris-opt14-strips (17)	**11**	10	**11**	**11**
transport-opt08-strips (30)	**14**	**14**	13	13
transport-opt11-strips (20)	**10**	**10**	9	8
transport-opt14-strips (20)	**8**	**8**	**8**	7
woodworking-opt08-strips (30)	20	20	19	**26**
woodworking-opt11-strips (20)	14	14	13	**18**
zenotravel (20)	**13**	**13**	7	7
Sum (1827)	1035	**1042**	974	1016

Table 2: Per domain coverage. Contains all domains where the best operator saturator $perim_o$+all_o solves more tasks than the best transition saturator $reach_t^+$,$perim_t^r$+$reach_t^r$ or vice versa.

Per domain coverage Table 2 shows the per domain coverage of a relevant subset of all domains. The best transition saturator $reach_t^+$,$perim_t^r$+$reach_t^r$ performs badly in the domains freecell, zenotravel, and nomystery, and it performs well in domains woodworking, ged and miconic. Furthermore, we see a favor for transition cost partitioning in domains where optimal plans contain an action multiple times. Intuitively, if an optimal plan contains an action multiple times, it applies the action in different states. Otherwise, the plan would not be optimal because it contains a cycle. However, duplicate actions in optimal plans are not necessary for transition cost partitioning to give more informed heuristics.

Future work

Finding better variable orderings for decision diagrams can further improve the performance and lower the risk of unmanageable large decision diagrams (Keller et al. 2016).

Another important problem to solve is finding mechanisms to select saturated transition cost functions that are computationally easier to handle but still allow us to profit

from more expressive cost assignments. Other selection mechanisms can provide us with polynomial-size guarantees for the representations of transition cost functions during saturated cost partitioning. The question is whether or not such smaller representations are capable of carrying enough context information that allows computing better cost partitioned heuristics.

Conclusion

We introduced subset-saturated transition cost partitioning that combines saturated transition cost partitioning with the concepts of preserving the heuristic estimates of a subset of states.

Our empirical evaluation shows that more expressive transition cost functions still require too much computational overhead but leads to more informed heuristics. Furthermore, subset-saturated transition cost partitioning lowers the risk of getting heuristics that are worse than heuristics of subset-saturated operator cost partitioning. In other words, the greediness of cost saturation becomes less problematic.

Subset-saturated transition cost partitioning allows selecting from a larger solution set of saturated transition cost functions. Crucial for further improvements is selecting saturated transition cost functions that are computationally easier to handle and still allowing us to obtain better cost partitioned heuristics by considering more expressive cost assignments.

Acknowledgements

David Speck was supported by the German Research Foundation (DFG) as part of the project EPSDAC (MA 7790/1-1). We sincerely thank the anonymous reviewers for their insightful and detailed comments.

References

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

Geißer, F.; Keller, T.; and Mattmüller, R. 2016. Abstractions for planning with state-dependent action costs. In *Proc. ICAPS 2016*, 140–148.

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.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS 2007*, 176–183.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. In *Proc. ICAPS 2004*, 122–131.

2014. *IPC-8 planner abstracts*.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *Proc. ICAPS 2008*, 174–181.

Keller, T.; Pommerening, F.; Seipp, J.; Geißer, F.; and Mattmüller, R. 2016. State-dependent cost partitionings for Cartesian abstractions in classical planning. In *Proc. IJCAI 2016*, 3161–3169.

Kissmann, P., and Edelkamp, S. 2014. Gamer and dynamic-gamer – symbolic search at ipc 2014. In *IPC-8 planner abstracts* (2014), 77–84.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Proc. AAAI 2015*, 3335–3341.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *Proc. IJCAI 2013*, 2357–2364.

Pommerening, F. 2017. *New Perspectives on Cost Partitioning for Optimal Classical Planning*. Ph.D. Dissertation, University of Basel.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In *Proc. ICAPS 2013*, 347–351.

Seipp, J., and Helmert, M. 2014. Diverse and additive Cartesian abstraction heuristics. In *Proc. ICAPS 2014*, 289–297.

Seipp, J., and Helmert, M. 2019. Subset-saturated cost partitioning for optimal classical planning. In *Proc. ICAPS 2019*, 391–400.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J.; Keller, T.; and Helmert, M. 2017. A comparison of cost partitioning algorithms for optimal classical planning. In *Proc. ICAPS 2017*, 259–268.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated cost partitioning for optimal classical planning. *JAIR* 67:129–167.

Somenzi, F. 1994. Cudd: colorado university decision diagram package - release 3.0.0.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018. SYMPLE: Symbolic Planning based on EVMDDs. In *IPC-9 planner abstracts*, 91–94.

Szepesvári, C. 2004. Shortest path discovery problems: A framework, algorithms and experimental results. In *Proc. AAAI 2004*, 550–555.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A symbolic bidirectional A* planner. In *IPC-8 planner abstracts* (2014), 105–109.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2018. Symbolic perimeter abstraction heuristics for cost-optimal planning. *AIJ* 259:1–31.

Investigating Lifted Heuristics for Timeline-based Planning

Riccardo De Benedictis and **Amedeo Cesta**
CNR - Italian National Research Council, Institute of Cognitive Sciences and Technologies
Via San Martino della Battaglia 44, 00185, Rome, Italy
{name.surname}@istc.cnr.it

Abstract

This paper investigates the use of lifted heuristics, inspired by the more classical ones for the resolution of STRIPS-like problems, for the efficient resolution of *timeline-based* planning problems. We propose, in particular, a new heuristic strategy which, while maintaining the variables lifted, allows more accurate decisions. Furthermore, the concepts presented in this work pave the way for a new type of heuristics which, at present, allow this kind of solvers a significant performance improvement.

Introduction

Since their early introduction, domain-independent heuristics have immediately proven to be a fundamental ally in solving difficult combinatorial problems such as those related to automated planning. The number of heuristics, introduced in recent years, for the efficient resolution of these problems has grown significantly to the point of constituting a research field (called *heuristic planning*) in its own. The different approaches that make up a solver's paraphernalia, range from the seminal h_{add} and h_{max} (Bonet and Geffner 2001) to the more recent developments relying on *delete-relaxation*, like the h^{FF} heuristic (Hoffmann and Nebel 2001) and the *causal graph* heuristics (Helmert 2006), on *landmarks*, like in (Hoffmann, Porteous, and Sebastia 2004; Porteous, Sebastia, and Hoffmann 2014), on the *critical path*, like the h^m heuristic (Haslum and Geffner 2000; Haslum, Bonet, and Geffner 2005) or, lastly, on *abstraction*, like in (Edelkamp 2014) or in (Helmert, Haslum, and Hoffmann 2007; Helmert et al. 2014).

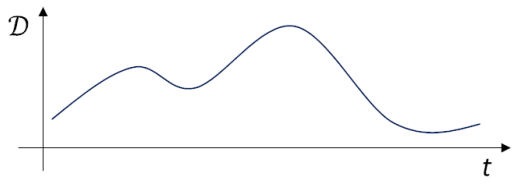
While the above heuristics are significantly heterogeneous among them (although, often, they share some commonalities), they have in common the fact that they have been developed specifically for the resolution of a particular type of problem, characterized by a specific modeling language called PDDL (Ghallab et al. 1998), representing a natural evolution of the most long-lived STRIPS (Fikes and Nilsson 1971) formalism. Despite the PDDL, over the years, has been extended through different directions by introducing *durative-actions* and *numeric fluents* (Fox and Long 2003), *derived predicates* and *timed initial literals* (Edelkamp and Hoffmann 2004), *continuous changes* (Fox and Long 2006), *state-trajectory constraints* and *preferences* (Gerevini et al. 2009) and *object-fluents*[1], the development of heuristics for reasoning with these more expressive formal systems has remained relatively limited to a few cases (e.g., (Piotrowski et al. 2016; Franco et al. 2019)).

Although it significantly departs from the previous ones, the *timeline-based* approach represents a different formalism that, already in its original formulation (Muscettola et al. 1992), is able to cover a large part of the above features. Although introduced before the aforementioned formalisms, this specific planning paradigm has always remained a niche within the automated planning community. The fragmentation of the different timeline-based formalisms, indeed, did not allow the emergence of a common language which would have enabled a fair comparison among the different reasoners. Furthermore, analogously to the solvers reasoning upon the previous PDDL extensions, timeline-based planners have to cope with the high expressiveness of the formalisms which, despite making them particularly suited at addressing real-world applications, unavoidably leads to performance issues. The contribution of this paper, a slightly modified version of (De Benedictis and Cesta 2020), is, hence, twofold: after providing a new formalization of the timeline-based problem, aiming to embracing the different aspects of the previous formalisms, we propose a new domain-independent heuristic which, inspired by the more classical ones, aims at improving the resolution efficiency.

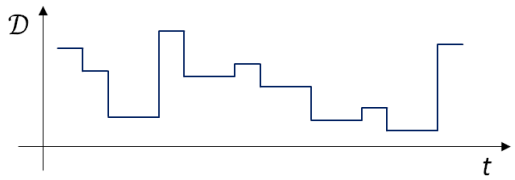
Timeline-based planning

Timeline-based planning was first introduced in (Muscettola et al. 1992; Muscettola 1994) and, since then, many solvers have been proposed like, for example, IxTeT (Ghallab and Laruelle 1994), EUROPA (Jonsson et al. 2000), ASPEN (Chien et al. 2010), the TRF (Fratini, Pecora, and Cesta 2008; Cesta et al. 2009) on which the APSI framework (Fratini et al. 2011) relies and, more recently, PLATINUm (Umbrico et al. 2017). Some theoretical work on timeline-based planning like (Frank and Jónsson 2003; Jonsson et al. 2000) was mostly dedicated to identifying connections with classical planning a-la PDDL (Fox and Long

[1] http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/kovacs-pddl-3.1-2011.pdf



(a) A continuous timeline.



(b) A step-wise timeline.

Figure 1: A continuous and a step-wise timeline.

2003). A recent new formalization of timeline-based planning has been proposed in (Cialdea Mayer, Orlandini, and Umbrico 2016), while (Gigante et al. 2020) studied its properties from a computational complexity point of view. The work on IxTeT and TRF has tried to clarify some key underlying principles but mostly succeeded in underscoring the role of time and resource reasoning (Cesta and Oddi 1996; Laborie 2003). The planner CHIMP (Stock et al. 2015) follows a Meta-CSP approach having meta-Constraints which havely resembles timelines. The Flexible Acting and Planning Environment (FAPE) (Dvorák et al. 2014) tightly integrates timelines with acting. The Action Notation Modeling Language (ANML) (Smith, Frank, and Cushing 2008) is an interesting development which combines the HTN decomposition methods with the expressiveness of the timeline representation. Finally, it is worth mentioning that the timeline-based approaches have been often associated to resource managing capabilities. By leveraging on constraint-based approaches, most of the above approaches like IxTeT (Laborie and Ghallab 1995; Laborie 2003), (Cesta, Oddi, and Smith 2002), (Smith, Frank, and Jónsson 2000) or (Verfaillie, Pralet, and Lemaître 2010) integrate planning and scheduling capabilities.

In order to better understand what we are talking about when discussing about timeline-based planning, it is important to introduce, without going into too much formal details, some basic concepts about *constraint networks* (Dechter 2003; Lecoutre 2009). Some of the timeline-based frameworks like, for example, those described in (Smith, Frank, and Jónsson 2000; Frank and Jónsson 2003), refer to timeline-based planning in terms of constraint-based planning, further emphasizing the central role that constraints take on within this type of planning. Formally,

Definition 1. *A* constraint network $\mathcal{N}$ *is composed of a finite set of variables, denoted by* $vars(\mathcal{N})$*, and a finite set of constraints, denoted by* $cons(\mathcal{N})$*.*

Specifically, constraint networks represent the lowest level elements on which timeline-based planning relies. The main data structure for the timeline-based paradigm is, indeed, the *timeline* which, in generic terms, is a function of time, either discrete or continuous, over a given domain. Formally,

Definition 2. *A* timeline T *is a function*

$$\mathsf{T} : \mathbb{T} \rightarrow \mathcal{D}$$

where $\mathbb{T}$ *is the (either discrete or continuous) domain of* time *and* $\mathcal{D}$ *is the (possibly infinite) domain of the timeline.*

It is worth noticing that the previous definition is quite general, not specifying any limitation neither on the time, which can be either discrete or continuous, nor on the domain which can be, in general, of any kind. Specifically, the domain of a timeline can be either symbolic (e.g., "a", "b", "c", etc.) or numeric (e.g., "1", "2", "3", etc.). Additionally, numeric domains can be either integer (e.g., "10", "12", "25", etc.) or real (e.g., "1.23", "2.17", "3.14", etc.). While integer domains can change in time only step-wise, real domains can change both step-wise and continuously. Finally, continuous changes can happen both linearly or non-linearly. Figure 1 (a), for example, represents a continuously updating non-linear timeline over reals. Figure 1 (b), on the contrary, shows a step-wise updating timeline.

Since the definition of timeline is completely general, it is possible to represent, through these, extremely heterogeneous concepts. We need, therefore, a unifying element that allows to represent contents homogeneously, in a way which is agnostic from the nature of the timeline. To this end, we introduce the concept of *token* and establish that values on timelines are a direct consequence of tokens through a timeline extraction procedure (more details soon). Without loss of generality, a token is an "assertion over a temporal interval". Formally,

Definition 3. *A* token *is an expression of the form:*

$$n\left(x_0, \ldots, x_i\right)_{\chi} @ \left[s, e, \tau\right]$$

where n *is a predicate name,* $x_0, \ldots, x_i$ *are the* parameters *of the predicate (i.e., constants, numeric variables or symbolic variables),* χ *is the* class *of the token (i.e., either a* fact *or a* goal*),* s *and* e *are the* temporal parameters *of the token (i.e., constants or variables) belonging to* $\mathbb{T}$ *such that* $s \leq e$ *and* τ *is the* scope parameter *of the token (i.e., a constant or a symbolic variable) representing the timeline on which the token apply.*

Roughly speaking, the expression on the left of the "@" symbol represents the "assertion" while the expression at its right represents the "interval". In other words, a token $n\left(x_0, \ldots, x_i\right)_{\chi} @ \left[s, e, \tau\right]$ asserts that $\forall t$ such that $s \leq t \leq e$, the relation $n\left(x_0, \ldots, x_i\right)$ holds at the time t on the timeline τ. Furthermore, given a token η, we call $pars\left(\eta\right)$ its parameters $x_0, \ldots, x_i, s, e, \tau$.

Tokens constitute the main building blocks of timeline-based plans. Regardless of the resolution procedure, indeed, the role of any timeline-based solver consists in introducing new tokens and/or establishing the values of their parameters. A critical aspect to keep in mind, when talking about

tokens, is that, in general, their parameters are variables of a constraint network and, as such, can be constrained. In other words, in order to reduce the allowed values for the tokens' constituting parameters, and thus decreasing the modeled system's allowed behaviors, it is possible to impose *constraints* among them (and/or among the parameters and other possible variables). Such constraints include temporal constraints, binding constraints between symbolic variables as well as (non)linear constraints among numerical variables (possibly including temporal variables).

The set of tokens and constraints is used to describe the main data structure that is used to represent (partial) plans of the timeline-based approach: the *token network*. Formally,

Definition 4. *A* token network *is a tuple* $\pi = (\mathcal{T}, \mathcal{N})$*, where:*

- $\mathcal{T} = \{\eta_0, \ldots, \eta_j\}$ *is a set of tokens, such that* $\forall \eta \in \mathcal{T}, pars(\eta) \subseteq vars(\mathcal{N})$.
- $\mathcal{N}$ *is a constraint network.*

Finally, as already mentioned, tokens can be partitioned into two classes: *facts* and *goals*. While facts are, by definition, inherently true, goals have to be achieved. Causality, in particular, in the timeline-based approach, is defined by means of a set o *rules* indicating how to achieve goals. Formally,

Definition 5. *A* rule *is an expression of the form*

$$n(x_0, \ldots, x_k) @ [s, e, \tau] \leftarrow \boldsymbol{r}$$

where:

- $n(x_0, \ldots, x_k) @ [s, e, \tau]$ *is the* head *of the rule, i.e. an expression in which* n *is a predicate name,* $x_0, \ldots, x_k$ *are the parameters of the head (i.e., numeric variables or symbolic variables),* s *and* e *are the* temporal parameters *of the head (i.e., constants or variables) belonging to* $\mathbb{T}$ *such that* $s \leq e$ *and* τ *is the* scope parameter *of the head (i.e., a constant or a symbolic variable) representing the timeline on which the rule apply.*
- $\boldsymbol{r}$ *is the* body *of the rule (or the* requirement*), i.e. either another token, a* constraint *among tokens (possibly including the* $x_0, \ldots, x_k, s, e, \tau$ *variables), a* conjunction *of requirements or a (priced*[2]*)* disjunction[3] *of requirements.*

Specifically, rules define causal relations that must be complied to in order for a given goal to be achieved. Roughly speaking, for each goal having the "form" of the head of a rule, the body of the rule (i.e., a logic combination of further tokens and constraints) must also be present in the token network. An example of rule is given by

$$At(?x) @ [s, e, \tau] \leftarrow \left\{ \left\{ \begin{array}{c} [e - s \geq 1] \wedge \\ \left\{ \begin{array}{c} dt : DriveTo(?x)_g @ [s, e, \tau] \wedge \\ [\tau == dt.\tau] \wedge [s == dt.e] \wedge \\ [?x == dt.?x] \end{array} \right\} \vee \\ \left\{ \begin{array}{c} ft : FlyTo(?x)_g @ [s, e, \tau] \wedge \\ [\tau == ft.\tau] \wedge [s == ft.e] \wedge \\ [?x == ft.?x] \end{array} \right\} \end{array} \right\} \right\}$$

By combining tokens, constraints, conjunctions and disjunctions, the above rule states that, in order to be in a given position, our agent must reach it either by driving or by flying.

We have now all the ingredients to define a timeline-based planning problem. In particular, the definition can rely on the above concept of requirement.

Definition 6. *A timeline-based* planning problem *is a triple* $\mathcal{P} = (\mathbf{T}, \mathcal{R}, \boldsymbol{r})$*, where:*

- $\mathbf{T}$ *is a set of timelines.*
- $\mathcal{R}$ *is a set of rules.*
- $\boldsymbol{r}$ *is a requirement, i.e. either a (fact or goal) token, a constraint among tokens, a conjunction of requirements or a (priced) disjunction of requirements.*

It is worth highlighting that, conversely to other timeline-based approaches, our formalism makes a clear distinction between tokens and values on timelines. This difference aims at guaranteeing us a further element of generality. The transition from tokens to timelines, however, requires the introduction of a further function which allows to *extract* the timelines from the tokens. Specifically,

Definition 7. *An* extraction function X_T *is a function for a timeline* T

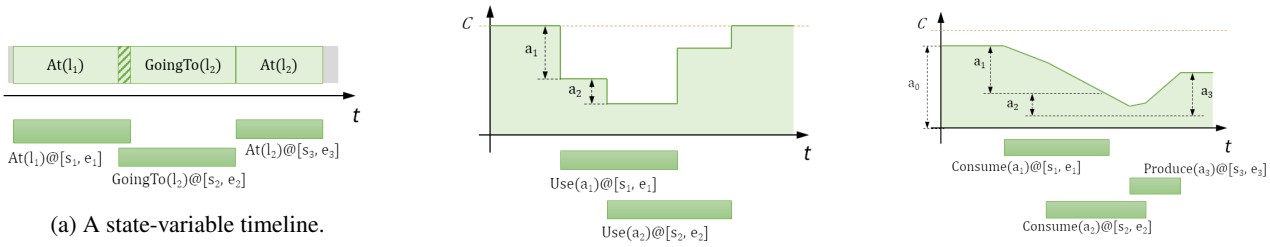
$$\mathsf{X}_\mathsf{T} : \mathbb{T} \times 2^{\mathcal{T}_\mathsf{T}} \rightarrow \mathcal{D}$$

where $\mathbb{T}$ *is the (either discrete or continuous) domain of* time*,* $\mathcal{T}_\mathsf{T}$ *is the set of tokens in the token network, having* T *in the domain of their* τ *variable, and* $\mathcal{D}$ *is the domain of the timeline.*

As can be easily seen by comparing Definition 2 with Definition 7, the result of the extraction function is, basically, a timeline. Each type of timeline, indeed, has associated its own timeline extraction procedure which allows to pass from the associated tokens to the resulting timelines. In other words, the timeline extraction procedure assigns to the tokens a higher-level semantic: according to the nature of the timeline, the procedure is able to "recognize the meaning" of the involved tokens. Note that, thanks to the introduction of the above higher-level semantic, not all token configurations lead to consistent timelines. According to the nature of the timeline, indeed, some configurations of tokens might lead to inconsistencies. It is responsibility of the solver to introduce further constraints so as to avoid such inconsistencies. Another way to see a timeline, indeed, is in terms of a *global constraint* (refer, for example, to (Dechter 2003; Lecoutre 2009)) over those tokens of the token network which assume the same value for their τ variables. Such global constraints, in particular, depend on the nature of the timeline, hence justifying the introduction of this concept within the formalism.

[2] It is possible, if needed, to associate a cost to the different disjuncts of a disjunction so as to model preferences.

[3] Some formalisms allow the definition of different rules having the same head, thus modeling the disjunctions. We preferred to replace this possibility by explicitly representing disjunctions. This choice can, in cases where these rules share some of the requirements, favor the modeler by reducing the size of the domain.



(a) A state-variable timeline.

(b) A reusable-resource timeline.

(c) A consumable-resource timeline.

Figure 2: Different timelines extracted by tokens.

Examples of timelines, extracted from tokens, are shown in the Figure 2. Specifically, Figure 2a shows a *state-variable* timeline, a step-wise timeline whose domain depends from the tokens which can be assigned, by means of the τ variable (omitted, for simplicity), to it. This type of timeline, in particular, introduces an additional global constraint that guarantees that different values, on the same timeline, cannot overlap in time. The state-variable of Figure 2a, as an example, has two values that overlap as a consequence of the overlapping of the $At\,(l_1)$ and the $GoingTo\,(l_2)$ tokens. Such an inconsistency can be solved, for example, by imposing an ordering constraint between the tokens (e.g., $e_1 \leq s_2$). Another type of timeline, typically used in pure scheduling problems, is the *reusable-resource* (see Figure 2b). This step-wise timeline is characterized by a maximum capacity and by a resource level which changes over time according to how the tokens, representing resource usages, overlap. The resource constraint guarantees that concurrent uses of the resource do not exceed its capacity. Finally, as example of a continuous timeline, the *consumable-resource* timeline (see Figure 2c) is characterized by a maximum capacity and by an initial amount. Similarly to reusable-resources, the resource level changes over time according to how the tokens, representing resource productions and consumptions, overlap, while the resource constraint guarantees that the level never exceeds the resource capacity nor goes below zero.

It is worth noticing that, unlike existing formalizations, by enabling any implementing solver to reason about timelines agnostically from their specific nature, the above definition allows us to maintain a certain generality. Furthermore, once provided an extraction function and the algorithms for managing the specific global constraint, new types of timelines can be introduced without affecting the solvers' resolution procedures.

The last aspect to consider regards the solution of a timeline-based planning problem. Roughly speaking, a solution is a token network whose all goals have been achieved. Furthermore, at least one *consistent* (i.e., does not violate any constraint) and *complete* (i.e., it includes all the variables) assignment of values to the variables of the underlying constraint network must be available. Notice that, among the constraints of the constraint network, there are also those which are imposed by the timelines. Formally,

Definition 8. *A token network $\pi = (\mathcal{T}, \mathcal{N})$ is a* solution *for a timeline-based planning problem $\mathcal{P} = (\mathbf{T}, \mathcal{R}, \boldsymbol{r})$ if:*

- *there exists a complete and consistent assignment of values to the variables of the constraint network $\mathcal{N}$.*
- *every goal $g \in \mathcal{T}$ is achieved (i.e., either the goal g is recognized as semantically equivalent to another token, or a rule, whose head is compatible with the token g, is applied).*

Reasoning with timelines

Unfortunately, the above definitions do not provide a computable test for building and verifying solutions. This section, therefore, introduces the typical approach for solving timeline-based planning problems. Specifically, common timeline-based solvers strongly rely on partial-order planning (Weld 1994) for reasoning, generalizing the concept of *threat* for including any possible inconsistency which might arise as a consequence of the timeline constraints (e.g., different states overlapping on the same state-variable, resources overusages, etc.). Despite this generalization, the search space (and, consequently, the solving algorithm) remains substantially unchanged. In particular, timeline-based solvers rely on the concept of *flaws*, that a token network has, and on the concept of *resolvers*, for solving them. Formally,

Definition 9. *A* flaw *in a token network $\pi = (\mathcal{T}, \mathcal{N})$ is either: (i) an* open goal *(i.e., a goal whose associated rule has not yet been applied or which has not yet been recognized as semantically equivalent to another token), (ii) a* threat *(i.e., any possible inconsistency arising as a consequence of the timeline constraints) or (iii) a* disjunction*.*

Intuitively, the main resolution principle consists in refining the token network π, identifying its flaws and applying resolvers for solving them, while maintaining the constraints $cons\,(\mathcal{N})$ consistent, until the token network π has no more flaws.

Figure 3 specifies a recursive non-deterministic procedure called TP (for Timeline-based Planning) for resolving timeline-based planning problems. Specifically:

- *flaws* denotes the set of all flaws in π provided by procedures `OpenGoals`, `Threats` and `Disjunctions`; φ is a particular flaw in this set.

procedure TP(π)
  $flaws \leftarrow$ OpenGoals (π) $\cup$ Threats (π) $\cup$ Disjunctions (π)
  if $flaws = \emptyset$ **then return** π
  end if
  select any flaw $\varphi \in flaws$
  $resolvers \leftarrow$ Resolve (φ, π)
  if $resolvers = \emptyset$ **then return** failure
  end if
  non-deterministically choose a resolver $\rho \in resolvers$
  $\pi' \leftarrow$ Refine (ρ, π)
  return TP (π')
end procedure

Figure 3: The TP procedure for solving timeline-based planning problems.

– *resolvers* denotes the set of all possible ways to resolve a specific flaw φ in a plan π and is given by the procedure `Resolve`. The resolver ρ is a particular element of this set.
– π' is the new plan obtained by refining π according to the resolver ρ as a consequence of the procedure `Refine`.

The TP procedure is called with an initial token network π_0, characterized by the problem's requirement. Each successful recursion is a refinement of the current plan according to the chosen resolver. In particular, the `Resolve` procedure returns all the resolvers that, in the token network π, solve the φ flaw. These resolvers depend, necessarily, on the type of flaw φ and on the current token network π. In the case of open goals, for example, resolvers represent the application of the corresponding rule or the *unification* (i.e., same predicate name and same, pairwise, parameter values, hence recognizing the tokens as semantically equivalent) with another already achieved goal or fact. In the case, for example, of excessive concurrent resource usages, conversely, resolvers could represent ordering constraints between couples of tokens. As a consequence, each invocation of the `Refine` procedure might introduce new tokens, new variables and/or new constraints to the token network. Intuitively, refinement operations should be chosen so as to avoid adding to the token network any constraint that is not strictly needed (this is called the *least commitment principle*).

Toward more effective heuristics

Reasoning within the above formal system is not at all simple[4]. It is worth noting that while the choice of the resolver is a *non-deterministic* step (i.e., it may be required to backtrack on this choice), the selection of a flaw is a *deterministic* step (i.e., there is no reason to backtrack on this choice) as all flaws need to be solved before or later in order to reach a solution plan. Despite the order in which flaws are processed is very important for the efficiency of the procedure, it is unimportant for its soundness and completeness. A deterministic implementation of the TP procedure should rely on algorithms like A* or IDA* so as to avoid that the search may keep exploring deeper and deeper a single path in the search space, adding indefinitely new tokens to the partial plan and never backtracking. As a consequence, choosing the *right* flaw and the *right* resolver becomes a crucial aspect for coping with the computational complexity and hence efficiently generating solutions.

The main difficulty derives from the impossibility of i) having a perfectly defined current state and ii) measuring the distance between this state and a desired state indicated in the formulation of the planning problem. For these reasons it becomes particularly inconvenient to use or even adapt, directly, the heuristics developed for classical formalisms. What we propose in this document is, somehow, to separate the temporal aspects from the purely causal ones, which in classical planning are strongly linked to be almost the same thing, and to apply classical heuristics only to the latter. In doing so, the rules of the timeline formalism become the equivalent of the PDDL operators, having the requirements as preconditions and the head of the rule as the only positive effect. Once this paradigm shift has been made, it becomes possible to adapt the heuristics of classical planning. Note that, however, this translation is not trivial: if, on the one hand, there is the simplification of having, for each operator, only a single positive effect (i.e., the solved flaw), on the other hand there is the difficulty of rendering atoms "ground" due to the presence of numerical parameters (representing, for example, the starting and the ending times of the tokens). We are therefore forced to reason about a sort of causal graph having lifted variables.

The overall proposed idea consists in applying, in a coarse way, *all* the possible resolvers for *all* the possible flaws until some termination criteria, i.e., unifications and resolvers which do not add further flaws, is met. Specifically, since flaws and resolvers are causally related (i.e., resolvers might introduce flaws which are solved by other resolvers, etc.) it is possible to build an AND/OR graph for representing such causal relations. By doing so, instead of searching in the space of the token networks, we have a single disjunctive token network containing all the possible plans (or, hopefully, only the "most interesting" ones) that can be found starting from the initial token network π_0. By exploiting the topology of such a graph it is possible to generate an estimation of "how far" a flaw is from being solved and exploit this estimation for guiding the resolution process. Specifically, taking inspiration from the h_{add} and the h_{max} heuristics introduced in (Bonet and Geffner 2001), the cost of a resolver, which can be seen as an AND node, can be estimated as the *maximum* (in case of h_{max} heuristic, or the *sum*, in case of the h_{add} heuristic) of the estimated costs of the flaws introduced by the resolver itself plus an intrinsic resolver's cost, while the estimated cost of a flaw, which can be seen as an OR node, can be estimated as the *minimum* of the estimated costs of its possible resolvers. Since all flaws must be solved, the solver chooses, among those that have to yet been solved, the most expensive one (i.e., the one that, most likely, will detect an inconsistency earlier) and will solve it with the least expensive resolver (i.e., the one that, more likely, will lead to a solution).

[4]Note that it is possible, in general, to represent through this formalism a self-referential proposition P, whose meaning is "P is false", hence showing the formalism's undecidability.

The lifted heuristic formulation

Before formally introducing the proposed heuristics, it is worth providing some definitions. Specifically, since the presence of flaws and resolvers, within the current partial solution, is controlled by a set of propositional variables, we refer to flaws by means of φ variables (we will use subscripts to describe specific flaws, e.g., φ_0, φ_1, etc.) and to resolvers by means of ρ variables (similarly to flaws, we will use subscripts to describe specific resolvers, e.g., ρ_0, ρ_1, etc.). Specifically, the value of such variables will be used to recognize active flaws that have to be solved (i.e., those flaws whose φ variables assume the $true$ value) and applied resolvers (i.e., those resolvers whose ρ variables assume the $true$ value). Additionally, given a flaw φ, we refer to the set of its possible resolvers by means of $res(\varphi)$ and to the (possibly empty) set of resolvers which are responsible for introducing it by means of $cause(\varphi)$. The latter set is usually constituted by the sole resolver representing the application of the rule which introduced the flaw. Nonetheless, this set can also be empty in case of top-level flaws, in which case the $true$ value is assigned to their controlling φ variables or, also, can contain more than one resolver in case the flaw is a consequence of their simultaneous application (e.g., a flaw representing two states overlapping on the same state-variable is activated whenever the rules that introduce the two states are applied simultaneously). Finally, given a resolver ρ, we refer to the set of its preconditions (e.g., the set of tokens introduced by the application of a rule) by means of $precs(\rho)$ and to the flaw solved through its application by means of $eff(\rho)$.

The above definitions allow us to formally introduce our heuristics. Specifically, let G be the estimated cost function, the estimated cost of a flaw φ and of a resolver ρ are characterized by the following equations:

$$G(\varphi) = min_{\rho \in res(\varphi)} G(\rho)$$
$$G(\rho) = c(\rho) + max_{\varphi \in precs(\rho)} G(\varphi)$$

where $c(\rho)$ is the *intrinsic cost* of the ρ resolver, i.e., a positive number representing the "cost" of disjuncts, in case of priced disjunctions, or the value 1, in other cases.

Similar to planning models based on satisfability (Kautz and Selman 1992), it is possible to introduce propositional constraints to the φ and ρ variables so as to guarantee the causal relations. By doing so, once the graph has been built, it is possible to frame the search space within a given boundary, dropping the computational complexity of the search procedure to a "simpler" NP-hard[5]. Furthermore, the introduction of these variables allows the use of propagation techniques and, in the event of inconsistencies, conflict analysis (and, hence, non-chronological backtracking) techniques, typical of SAT/SMT based solvers. The planning problem is therefore reduced to the assignment of $true$ values to the variables associated to the resolvers while observing the assignment, as a consequence of constraint propagation, of $true$ values to the variables associated to the flaws.

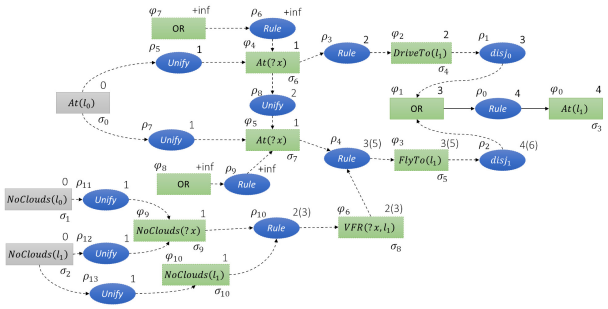


Figure 4: An example of causal graph with lifted variables.

Additionally, in order to establish the presence or not of the tokens inside the solution, a state variable $\sigma \in \{inactive, active, unified\}$ is associated to each token. A partial solution will hence consist solely of those tokens of the token network which are *active*. Moreover, in case such tokens are goals, the bodies of the associated rules must also be present within the solution. The *unified* tokens do not participate directly in the partial solution, since they are recognized as semantically equivalent to other active tokens, yet, since possibly subject to constraints, they might indirectly influence the "shape" of the solution. Finally, *inactive* tokens do not participate at all in the solution. We refer to tokens, later on, by means of σ variables (we will use subscripts to describe specific tokens, e.g., σ_0, σ_1, etc.) and to the flaws introduced by tokens by means of the $\varphi(\sigma)$ function.

An explanatory example

In order to better understand how the heuristic and causality constraints work we introduce, in this section, a very simple example involving an agent moving between different locations either by driving or by flying (which, in turn, requires good weather). Figure 4 shows an example of the graph which is generated for solving the problem of going from l_0 (a fact) to l_1 (a goal).

Estimated costs for flaws (boxes) and resolvers (ovals) are on their upper right. Notice that, in the example, the flaw φ_0 can only be solved by resolver ρ_0 which is directly applied (solid lines represent what is in the current partial solution). Additionally, since $\varphi_0 = \varphi(\sigma_3)$, the $active$ value is assigned to σ_3. The first flaw to be solved is, hence, φ_1, which can be solved either with resolver ρ_1, having an estimated cost of 3, or with resolver ρ_2 having an estimated cost of 4[6]. Applying, for example, the least expensive resolver ρ_1 would lead, as a consequence of constraint propagation, to the activation of the flaw φ_2 (notice that $precs(\rho_1) = \{\varphi_2\}$ and $cause(\varphi_2) = \{\rho_1\}$) which can be solved with the sole resolver ρ_3, which in turn activates the flaw φ_4 which is

[5]There is, intuitively, no guarantee that the built graph contains a solution. Similarly to what happens in Graphplan (Blum and Furst 1997), indeed, it might be required the addition of a "layer" to the graph.

[6]In the figure, the estimated costs are represented in the upper right of the flaws/resolvers and are computed through the h_{max} heuristic. Whenever they do not coincide, in parenthesis is also represented the value from the h_{add} heuristic.

solved with resolver ρ_5 leading to a solution. Finally, since $\varphi_4 = \varphi(\sigma_6)$, the $unified$ value is assigned to σ_6.

Current results

The causal graph, described in the previous section, has been implemented within the ORATIO solver[7]. In order to show the effectiveness of the proposed approach, we tested the solver, enhanced with the above heuristic, on different instances of the GOAC domain. Specifically, the Goal Oriented Autonomous Controller (GOAC) was an ESA initiative aimed at defining a new generation of software autonomous controllers to support increasing levels of autonomy for robotic task achievement. In particular, the domain, initially defined in (Fratini et al. 2011) and more recently cited in (Coles et al. 2019), aims at controlling a rover to take a set of pictures, store them on board and dump the pictures when a given communication channel was available. The interesting aspect of this domain is that communication can only take place within specific visibility windows that take into account the astronomical motions of the planets/satellites which, in some cases, may stand between the transmitting and receiving stations. The presence of these visibility windows, in particular, requires an explicit modeling of temporal aspects in order to adequately plan the transmission of information and can hence easily be modeled through, and solved by, timeline-based planners. The problem is made more interesting by the presence of constraints which include the available resources (e.g., memory and battery) as well as by having a distance matrix, among the possible locations, which might be not completely connected.

Figure 5 shows the execution times of different timeline-based solvers (i.e., EPSL (Cesta, Orlandini, and Umbrico 2013), AP2 (Fratini et al. 2011), J-TRE (De Benedictis and Cesta 2012), one of the precursors of ORATIO using a less accurate heuristic (De Benedictis and Cesta 2016), and the more recent PLATINUm (Umbrico et al. 2017)) as well as a couple of temporal-planning solvers (i.e., OPTIC (Benton, Coles, and Coles 2012) and COLIN (see (Coles et al. 2012)), both based on a classic FF-style forward chaining search (Hoffmann 2001)) in solving different instances of the GOAC problem. In particular, problems are obtained by varying the problem complexity along the number of pictures to be taken and the number of communication windows. Among all the generated problem instances, in particular, the ones with higher number of required pictures and higher number of visibility windows result as the hardest ones. The right mix of causal and temporal aspects makes the GOAC problem particularly complex to the point that some of the planners, beyond a certain number of pictures to collect and data dumps, show serious scalability issues. As shown in the figure, besides being considerably more efficient, compared to other timeline-based planners, ORATIO is also able to solve more complex instances. Compared to the temporal-planning solvers, however, it is clear that, despite significant improvements, there is still a performance gap to fill. Possible explanations of this gap include the maintenance, in the current state of the solvers, of the consistency between the various constraints (which is not required in the forward state space search planners), in addition to the greater effectiveness of the FF heuristics. Another aspect to take into consideration regards the possibility of making the graph more accurate, so as to be able to represent heuristics as h^2 (Haslum and Geffner 2000; Haslum, Bonet, and Geffner 2005). Since it is not possible to recognize the mutual exclusivity between the resolvers directly from the rules' structure, we have not yet found an effective approach for implementing it.

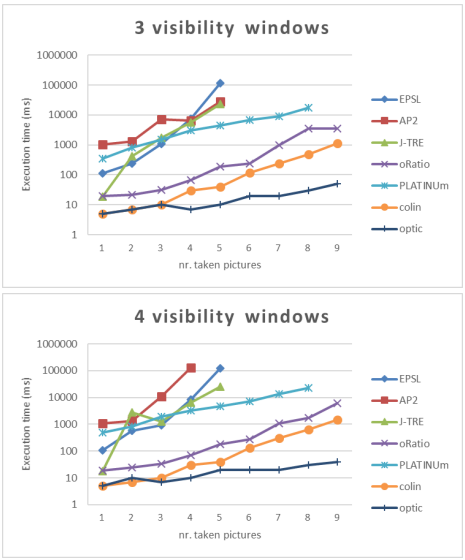


Figure 5: Execution times of different solvers to instances, of increasing complexity, of the GOAC problem.

Comments on the results. Although, for the moment, there are solvers able to solve the GOAC problem more efficiently than the ORATIO solver, we believe that the current results are nevertheless significant. In the first place, indeed, the heuristic described in this document proposes a complete paradigm shift for timeline-base planners: we pass from heuristics based on the current partial solution (i.e., the current token network π) to heuristics based on all possible plans that can be generated from starting from the planning problem. In so doing, we have the possibility to anticipate the consequences of decisions before they are even taken and this results in more accurate plan synthesis. A second aspect to consider regards the possibility of modeling (and, above all, integrating) different kinds of reasoning which depart from those more closely related to automated planning. By removing the temporal parameters from the tokens, indeed, we obtain a form of reasoning which is similar to constrained logic programming. The proposed heuristics, in particular, still remain valid, and paves the way for the efficient integration of different forms of reasoning such as, for example, automated planning and semantic reasoning. To better understand this aspect we can consider as an example the execution of Prolog program, whose efficiency strongly de-

[7] https://github.com/pstlab/oRatio

pends on the order in which the goals are defined within the rules as well as on the order in which the rules are defined. Different rules having the same goals defined in a different order are, indeed, semantically equivalent. The programmer, however, could be wrong at defining such orders or, even worse, the most efficient order could depend on the value of the parameters, unavoidably affecting the performance of the resolution process. The introduction of heuristics such as those presented would alleviate these types of problems.

Conclusions

The reasons for introducing a new timeline-based formalism are manifold and range from the possibility to model, through a uniform formalism, continuous changes over time (see, for example, Figure 1a) to make the plans more flexible in the execution phase (relaxing the constraint, present in some formalisms, that forces the timelines to be completely filled over time). Whatever the formalism, reasoning upon these systems remains particularly challenging from a computational point of view. For this reason we have introduced a new heuristic that takes into account, before starting the search, *all* possible resolvers for *all* possible flaws that may emerge from the resolution process, so as to be able to make choices according to a more accurate criterion. Although encouraging, the results show that there is still work to be done. As an example, since it is possible to recognize mutex resolvers by propagating constraints, it is worth to investigate different approaches for representing the h^2 heuristic. Analogously, the proper adaptation of landmark-based heuristics, might represent a fruitful path toward the resolution efficiency. We hence believe that, through this document, we can lay the foundations for the definition of a new typology of heuristics for the efficient resolution of timeline-based planning problems.

Acknowledgments. Authors work is partially supported by the INdAM-GNCS project *Metodi formali per tecniche di verifica combinata*, and by SI-ROBOTICS [8]. They are members of the OVERLAY[9] network.

References

Benton, J.; Coles, A.; and Coles, A. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

Blum, A. L., and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90(1-2):281–300.

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1-2):5–33.

Cesta, A., and Oddi, A. 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem. In Chittaro, L.; Goodwin, S.; Hamilton, H.; and Montanari, A., eds., *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*. IEEE Computer Society Press: Los Alamitos, CA.

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2009. Developing an End-to-End Planning Application from a Timeline Representation Framework. In *IAAI-09. Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference, Pasadena, CA, USA*.

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A Constraint-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics* 8(1):109–136.

Cesta, A.; Orlandini, A.; and Umbrico, A. 2013. Toward a general purpose software environment for timeline-based planning. In *20th RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*.

Chien, S.; Tran, D.; Rabideau, G.; Schaffer, S.; Mandl, D.; and Frye, S. 2010. Timeline-Based Space Operations Scheduling with External Constraints. In *ICAPS-10. Proc. of the 20th Int. Conf. on Automated Planning and Scheduling*.

Cialdea Mayer, M.; Orlandini, A.; and Umbrico, A. 2016. Planning and execution with flexible timelines: a formal account. *Acta Informatica* 53(6):649–680.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2012. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research* 44:1–96.

Coles, A.; Coles, A.; Martinez Munoz, M.; Savas, O.; Delfa, J.; de la Rosa, T.; E-Martín, Y.; and García Olaya, A. 2019. Efficiently Reasoning with Interval Constraints in Forward Search Planning. In *Proceedings of the Thirty Third AAAI Conference on Artificial Intelligence*. AAAI Press.

De Benedictis, R., and Cesta, A. 2012. New Reasoning for Timeline based Planning - An Introduction to J-TRE and its Features. In *ICAART 2012 - 4th International Conference on Agents and Artificial Intelligence*, 144–153. SciTePress.

De Benedictis, R., and Cesta, A. 2016. Investigating domain independent heuristics in a timeline-based planner. *Intelligenza Artificiale* 10(2):129–145.

De Benedictis, R., and Cesta, A. 2020. Lifted Heuristics for Timeline-Based Planning. In *ECAI 2020 - 24th European Conference on Artificial Intelligence, Santiago de Compostela, Spain*, volume 325, 2330–2337.

Dechter, R. 2003. *Constraint Processing*. Elsevier Morgan Kaufmann.

Dvorák, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Plan-Space Hierarchical Planning with the Action Notation Modeling Language. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Institut für Informatik.

Edelkamp, S. 2014. Planning with Pattern Databases. In *Sixth European Conference on Planning*.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *IJCAI*, 608–620.

[8] PON 676–Ricerca e Innovazione 2014-2020–G.A. ARS01_01120

[9] https://overlay.uniud.it

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fox, M., and Long, D. 2006. Modelling Mixed Discrete-continuous Domains for Planning. *Journal Of Artificial Intelligence Research* 27(1):235–297.

Franco, S.; Vallati, M.; Lindsay, A.; and McCluskey, T. L. 2019. Improving Planning Performance in PDDL+ Domains via Automated Predicate Reformulation. In *Computational Science – ICCS 2019*, 491–498. Springer International Publishing.

Frank, J., and Jónsson, A. K. 2003. Constraint-Based Attribute and Interval Planning. *Constraints* 8(4):339–364.

Fratini, S.; Cesta, A.; De Benedictis, R.; Orlandini, A.; and Rasconi, R. 2011. APSI-based Deliberation in Goal Oriented Autonomous Controllers. *ASTRA* 11.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):231–271.

Gerevini, A. E.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: {PDDL3} and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668. Advances in Automated Plan Generation.

Ghallab, M., and Laruelle, H. 1994. Representation and Control in IxTeT, a Temporal Planner. In *AIPS-94. Proceedings of the 2nd Int. Conf. on AI Planning and Scheduling*, 61–67.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—The Planning Domain Definition Language.

Gigante, N.; Montanari, A.; Orlandini, A.; Cialdea Mayer, M.; and Reynolds, M. 2020. On timeline-based games and their complexity. *Theoretical Computer Science* 815:247–269.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, 140–149. AAAI Press.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New Admissible Heuristics for Domain-Independent Planning. In *AAAI*, volume 5, 9–13.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM (JACM)* 61(3):16.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *ICAPS*, 176–183.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26(1):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.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research* 22:215–278.

Hoffmann, J. 2001. FF: The Fast-Forward Planning System. *AI Magazine* 22(3):57–62.

Jonsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in Interplanetary Space: Theory and Practice. In *AIPS-00. Proceedings of the Fifth Int. Conf. on AI Planning and Scheduling*.

Kautz, H., and Selman, B. 1992. Planning as Satisfiability. In *ECAI*, volume 92, 359–363.

Laborie, P., and Ghallab, M. 1995. Planning with Sharable Resource Constraints. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, 1643–1649. Morgan Kaufmann Publishers Inc.

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artificial Intelligence* 143:151–188.

Lecoutre, C. 2009. *Constraint Networks: Techniques and Algorithms*. Wiley-IEEE Press.

Muscettola, N.; Smith, S.; Cesta, A.; and D'Aloisi, D. 1992. Coordinating Space Telescope Operations in an Integrated Planning and Scheduling Architecture. *IEEE Control Systems* 12.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., ed., *Intelligent Scheduling*. Morgan Kauffmann.

Piotrowski, W.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic Planning for PDDL+ Domains. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, 3213–3219. AAAI Press.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2014. On the Extraction, Ordering, and Usage of Landmarks in Planning. In *Sixth European Conference on Planning*.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Smith, D. E.; Frank, J.; and Jónsson, A. K. 2000. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Hierarchical hybrid planning in a mobile service robot. In *KI 2015 Proceedings*, 309–315.

Umbrico, A.; Cesta, A.; Cialdea Mayer, M.; and Orlandini, A. 2017. Platinum: A new framework for planning and acting. In *AI*IA 2017 Proceedings*, 498–512.

Verfaillie, G.; Pralet, C.; and Lemaître, M. 2010. How to model planning and scheduling problems using constraint networks on timelines. *The Knowledge Engineering Review* 25(3):319–336.

Weld, D. S. 1994. An Introduction to Least Commitment Planning. *AI Magazine* 15(4):27–61.

Generating Data In Planning: SAS$^+$ Planning Tasks of a Given Causal Structure

Michael Katz and **Shirin Sohrabi**
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
michael.katz1@ibm.com, ssohrab@us.ibm.com

Abstract

The need for data in planning has long been established, by, e.g., machine learning based approaches. The existing data, however, is quite limited. There exists only a relatively small amount of hand-crafted planning domains, mostly introduced through International Planning Competitions. Further, this collection of domains is not necessarily diverse: many of these domains are some variants of the transportation problem.

In this work, we alleviate the shortage in existing planning tasks by automatically generating tasks of a particular causal structure. Given any graph G, we show how to create a SAS$^+$ planning task with the causal graph isomorphic to G. We create a large collection of planning tasks by randomly generating graphs of various structural restrictions and creating planning tasks for these graphs, but also, more importantly, we provide the community with a tool that allows for on-demand generation of additional, possibly larger tasks. Our experimental evaluation ensures that the generated collection is interesting for the current state of affairs in classical cost-optimal planning, showing the performance of state-of-the-art symbolic search and explicit heuristic search based planners.

Introduction

Since the first planning tasks encoded in STRIPS language back in 1971 (Fikes and Nilsson 1971), data, a.k.a. planning tasks, was the corner stone and one of the main drivers of research in planning. With the beginning of International Planning Competitions (IPC) in 1998 (McDermott 2000) came the increase in the availability of planning tasks, with the current estimate of slightly over 70 domains, including some variants in different formalisms. All these domains are hand-crafted, although some correspond to machine translation from a different problem (Palacios and Geffner 2009; Bonet, Palacios, and Geffner 2009; Grastien and Scala 2018; Sohrabi et al. 2018). Not only that most of these domains are hand-crafted, the collection is not necessarily diverse. Many of these domains are some variants of the transportation problem.

A major focus in classical planning was on heuristic search, with heuristics automatically obtained for planning tasks, exploiting the task structure. Few examples that explicitly exploit the causal structure include the *causal graph heuristic* (Helmert 2004) and the *structural pattern heuristics* (Katz and Domshlak 2010; Katz and Keyder 2012). Others, such as *pattern databases* (PDBs) (Edelkamp 2001) exploit the causal information in e.g., pattern selection (Haslum et al. 2007). *Merge-and-shrink heuristics* (Helmert, Haslum, and Hoffmann 2007) use the causal graph for guiding the merge process. Most existing heuristics that work on the multi-valued representation exploit the causal information in one way or another. Further, starting with the seminal work of Bäckström and Nebel (1995), the research on the complexity of planning tasks had a major focus on the characterization of planning fragments by their causal graph structure (Domshlak and Brafman 2002; Katz and Domshlak 2007; 2008; Giménez and Jonsson 2008; 2009; Katz and Keyder 2012; Bäckström and Jonsson 2013; Aghighi, Jonsson, and Ståhlberg 2015; Bäckström, Jonsson, and Ordyniak 2019), as well as some local structural characteristics, such as *k-dependence* (Katz and Domshlak 2007; Giménez and Jonsson 2012), showing these fragments to belong to a variety of complexity classes. For these two reasons, various planners performance heavily relies on the various structural characteristics of the input planning task.

The aim of this work is to generate planning tasks of a specific predefined structure. Here, we focus on the characterization of planning tasks by the structure of their causal graphs. Given a collection of multi-valued variables and a graph representing causal connections between these variables, we propose a way of generating SAS$^+$ actions, initial state and a goal, in a way that the causal graph of the resulting task will match the input graph. Our aim is to be able to automatically generate a diverse collection of planning tasks, as large as needed for various purposes. One such example purpose is learning a good planner selection strategy (Sievers et al. 2019; Ma et al. 2020). Another possible purpose is an additional source of benchmarks for empirical evaluation of new planning algorithms. We test the generated collection with two modern cost-optimal planners that represent the two popular state-of-the-art approaches to cost-optimal planning. For symbolic search, we chose the planner SYMBA* (Torralba et al. 2014), winner of the sequential optimal track of International Planning Competition 2014 and one of the planners in the winning portfolio of IPC 2018

(Katz et al. 2018). For heuristic search, we chose A^* with LM-cut heuristic (Helmert and Domshlak 2009), a component of many modern heuristics search based planners. Our experiments confirm that (i) the generated collection is challenging for both heuristic search and symbolic search based planners, and (ii) there is no clear dominance to any of the techniques.

The rest of the paper is structured as follows. We start with introducing the planning formalism and the notation used throughout the paper. We then move to construction, where we first describe various causal graph structures and the way these graphs can be constructed, and then describe the construction of planning tasks given a causal graph. Next, we present the experimental evaluation, including describing the way we have created our collection. Finally, we discuss the related work, and conclude with the summary of our results and future work.

Preliminaries

A SAS$^+$ *planning task* (Bäckström and Nebel 1995) is given by a tuple $\langle \mathcal{V}, A, s_0, s_* \rangle$, where $\mathcal{V}$ is a set of *state variables*, A is a finite set of *actions*. Each state variable $v \in \mathcal{V}$ has a finite domain $dom(v)$. Each pair $\langle v, \vartheta \rangle$ of variable $v \in \mathcal{V}$ and its value $\vartheta \in dom(v)$ is called a *fact*. By F_v we denote the set $\{\langle v, \vartheta \rangle \mid \vartheta \in dom(v)\}$ of facts for the variable v, and the set of all facts is denoted by $F := \bigcup_{v \in \mathcal{V}} F_v$. A (partial) assignment to the variables $\mathcal{V}$ is called a *(partial) state*. Often it is convenient to view partial state p as a set of facts with $\langle v, \vartheta \rangle \in p$ if and only if $p[v] = \vartheta$. For a partial assignment p, $\mathcal{V}(p) \subseteq \mathcal{V}$ denotes the subset of state variables instantiated by p. Partial state p is *consistent* with state s if $p \subseteq s$. We denote the set of states of a planning task by S. s_0 is the *initial state*, and the partial state s_* is the *goal*. Each *action* a is a pair $\langle pre(a), eff(a) \rangle$ of partial states called *preconditions* and *effects*. By $prv(a)$ we denote the part of the precondition that corresponds to variables that do not participate in action's effects, $prv(a) = \{\langle v, \vartheta \rangle \in pre(a) \mid v \notin \mathcal{V}(eff(a))\}$, also called *prevail condition*. An *action cost* is a mapping $C : A \to \mathbb{R}^{0+}$. An action a is applicable in a state $s \in S$ if and only if $pre(a)$ is consistent with s. Applying a changes the value of $v \in \mathcal{V}(eff(a))$ to $eff(a)[v]$. The resulting state is denoted by $s[\![a]\!]$. An action sequence $\pi = \langle a_1, \ldots, a_k \rangle$ is applicable in s if there exist states $s_0, \cdots, s_k$ such that (i) $s_0 = s$, and (ii) for each $1 \leq i \leq k$, a_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![a_i]\!]$. We denote the state s_k by $s[\![\pi]\!]$. π is a plan iff π is applicable in s_0 and s_* is consistent with $s_0[\![\pi]\!]$. We denote by $\mathcal{P}(\Pi)$ (or just $\mathcal{P}$ when the task is clear from the context) the set of all plans of Π. The cost of a plan π, denoted by $C(\pi)$ is the summed cost of the actions in the plan.

A central role in what follows is played by a standard structure in classical planning, called *causal graph* (Helmert 2004). The causal graph CG_Π of a task Π is a digraph with vertices $\mathcal{V}$. An arc (v, v') is in CG_Π iff $v \neq v'$ and there exists an action $a \in A$ such that $(v, v') \in [\mathcal{V}(eff(a)) \cup \mathcal{V}(pre(a))] \times \mathcal{V}(eff(a))$. For an action a, by E_a we denote the set of all such arcs, and by $E_{A'}$ we denote the union of all sets of arcs E_a for $a \in A'$.

Another structure typically used in planning for computing relaxation based heuristics is *relaxed planning graph* (Hoffmann and Nebel 2001), which is a layered graph of facts and actions, describing action application in the planning task, under value accumulating semantic. The layers are added until a fixpoint is reached, that is no new fact can be achieved. The first fact layer F_1 thus corresponds to the facts from the initial state, and the last layer is also a fact layer, and it is equal to the preceding fact layer. Each action layer A_i consists of all actions from A that are applicable in F_i, that is $A_i = \{a \in A \mid pre(a) \subseteq F_i\}$. The next fact layer F_{i+1} is then constructed by adding to F_i all facts achieved by the actions in A_i, namely $F_{i+1} = F_i \cup \bigcup_{a \in A_i} eff(a)$.

Finally, SAS$^+$ representation is often not provided directly, and is *translated* from STRIPS representation (Helmert 2006). The multi-valued variables in SAS$^+$ then correspond to invariant groups of pairwise mutually exclusive facts (mutexes), where exactly one such fact is true in any state reachable from the initial state. Each such invariant group over STRIPS facts corresponds to a set of facts *at most one* of which can be true in any given state that is reachable from the initial state. If there exist such states where no facts are true, then an additional value is added, representing that none of the facts in the invariant group is true.

Construction

We start our construction by defining a graph to be served as the causal graph of the constructed task. For that, we focus here on the following causal graph structures: *chain*, *directed chain*, *fork*, *inverted fork*, *star*, *bipartite graph*, *directed bipartite graph*, *tree*, *polytree*, and *directed acyclic graph*, *complete graph*, and *random graph*. For some of these structures, namely *directed chain*, *fork*, *inverted fork*, and *complete graph* the graphs are fully defined by the number of nodes (modulo automorphisms). In other cases, we introduce randomness into the graph construction. In what follows, we first describe how we handle these cases, and then how a task with the given causal structure is constructed.

Directed Bipartite Graph: A full directed bipartite graph is constructed by first randomly partitioning the nodes into left and right and then introducing an edge from each node on the left to each node on the right.

Bipartite Graph: A full undirected bipartite graph is constructed by first randomly partitioning the nodes into left and right and then introducing an edge from each node on the left to each node on the right, and vice versa.

Directed Chain: A *directed* chain of n nodes $v_1, \ldots, v_n$ is created by adding the edges (v_i, v_{i+1}) for each $1 \leq i < n$.

Chain: An *undirected* chain of n nodes $v_1, \ldots, v_n$ is created as follows. For each $1 \leq i < n$, we randomly decide whether to add an edge (v_i, v_{i+1}), with probability p. If no such edge is added, we add the edge (v_{i+1}, v_i) and if the edge (v_i, v_{i+1}) was added, we decide with probability p whether to add the edge (v_{i+1}, v_i).

Tree: A directed tree of n nodes $v_1, \ldots, v_n$ is constructed by choosing for each node v_i a parent randomly out of the

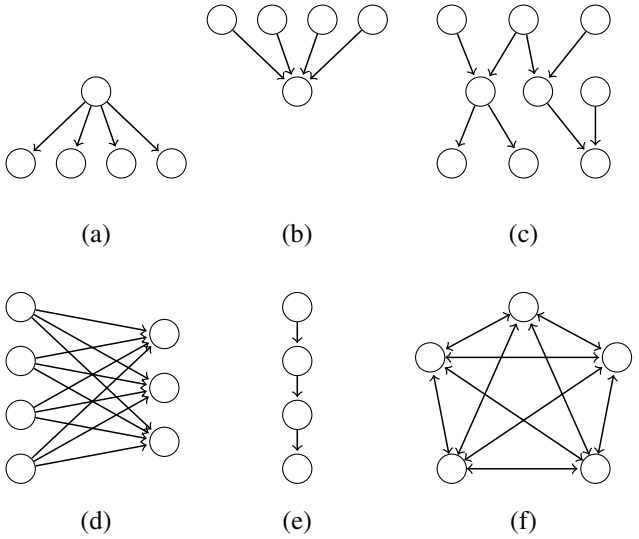


Figure 1: Selected causal graph structures: (a) fork, (b) inverted fork, (c) polytree, (d) directed bipartite graph, (e) chain, and (f) complete graph .

nodes $v_1, \ldots, v_{i-1}$.

Polytree: For a polytree, we start with a tree constructed as above, and then for each edge switch its direction with probability p.

Directed Acyclic Graph: A directed acyclic graph of n nodes $v_1, \ldots, v_n$ is constructed by choosing for each node v_i at least one parent randomly out of the nodes $v_1, \ldots, v_{i-1}$. We do that by going over all the preceding nodes and deciding with probability p whether to add an edge from the preceding node to the current node. If no edges were added, we repeat until at least one edge is added for each node (except the first one).

Random Graph: For each pair of nodes v_i and v_j we randomly decide whether to add a directed edge from v_i to v_j.

Fork: A fork is a directed tree with all non-root nodes being leafs, with their parent being the root node. A fork over nodes $v_1, \ldots, v_n$ is created by adding the edges (v_1, v_i) for each $1 < i \leq n$.

Inverted Fork: An inverted fork is a directed polytree with one leaf node and all non-leaf nodes being roots, with their only child node being the leaf node. An inverted fork over nodes $v_1, \ldots, v_n$ is created by adding the edges (v_i, v_1) for each $1 < i \leq n$.

Star: A star structure has one central node with all other nodes connected with the central node only. A star over nodes $v_1, \ldots, v_n$ is created as follows. For each $1 < i \leq n$, we randomly decide whether to add an edge (v_1, v_i), with probability p. If no such edge is added, we add the edge (v_i, v_1) and if the edge (v_1, v_i) was added, we decide with probability p whether to add the edge (v_i, v_1).

Algorithm 1 Construction of a planning task according to a given causal graph structure.

Input: Graph $G = (\mathcal{V}, E)$, number of facts $n \geq 2|\mathcal{V}|$

Partition n into $|\mathcal{V}|$ values $d_v \geq 2$ such that $\sum_{v \in \mathcal{V}} d_v = n$
$F_v \leftarrow \{\langle v, \vartheta \rangle \mid 0 \leq \vartheta < d_v\}$ for all $v \in \mathcal{V}$
$s_0[v] \leftarrow 0$ for all $v \in \mathcal{V}$
$k \leftarrow 0$
$A \leftarrow \emptyset$
$F_k \leftarrow s_0$
while $|F_k| < n$ **or** $E \setminus E_A \neq \emptyset$ **do**
    $k \leftarrow k + 1$
    $m_k \leftarrow$ number of new facts for layer k
    $F_k, A_{k-1} \leftarrow$ CREATELAYER(m_k, F_{k-1}, A)
    $A \leftarrow A \cup A_{k-1}$
Select $s_* \subseteq F_k$ such that $\forall v \in \mathcal{V}, |s_* \cap F_v| \leq 1$ and $s_* \cap (F_k \setminus F_{k-1}) \neq \emptyset$
return $\Pi = \langle \mathcal{V}, A, s_0, s_* \rangle$

function CREATELAYER(m, F, A)
    $A' \leftarrow \emptyset$
    $F' \leftarrow F$
    while $|F' \setminus F| < m$ **or** ($m = 0$ **and** $E \setminus E_{A \cup A'} \neq \emptyset$) **do**
        $a \leftarrow$ CREATEACTION
        if $E_a \subseteq E$ **then**
            if $E_a \setminus E_{A \cup A'} = \emptyset$ **and** Random(p) **then**
                Continue
            $A' \leftarrow A' \cup \{a\}$
            $F' \leftarrow F' \cup \mathit{eff}(a)$
    return F', A'

Complete Graph: A complete graph over nodes $v_1, \ldots, v_n$ is created by adding the edges (v_i, v_j) and (v_j, v_i) for all $1 \leq i < j \leq n$.

Figure 1 exemplifies selected graph structures. Having described how a causal graph for the future planning task is constructed, we now switch to the next step, showing how to construct a planning task with that causal graph.

Planning Task Construction

Given a graph $G = (\mathcal{V}, E)$ and a number of facts $n \geq 2|\mathcal{V}|$, we construct the SAS$^+$ planning task $\Pi = \langle \mathcal{V}, A, s_0, s_* \rangle$ with the causal graph G as follows. First, we choose the domain size $d_v \geq 2$ for each of the multi-valued variables $v \in \mathcal{V}$ and assume w.l.o.g. the values to be $dom(v) = \{0, \ldots, d_v - 1\}$. The variables represent sets of mutually exclusive facts (mutexes), and each such set corresponds to one of the two types of mutexes, namely, either exactly one or at most one of the values is true in all reachable states. We randomly decide which variables belong to which category. For the variables that represent the at-most-one case, we dedicate the last domain value to represent the case when none of the other facts are true. Next, w.l.o.g. we assume $s_0[v] = 0$ for all $v \in \mathcal{V}$. Then, we construct the actions, in layers, while constructing the relaxed planning graph. Fi-

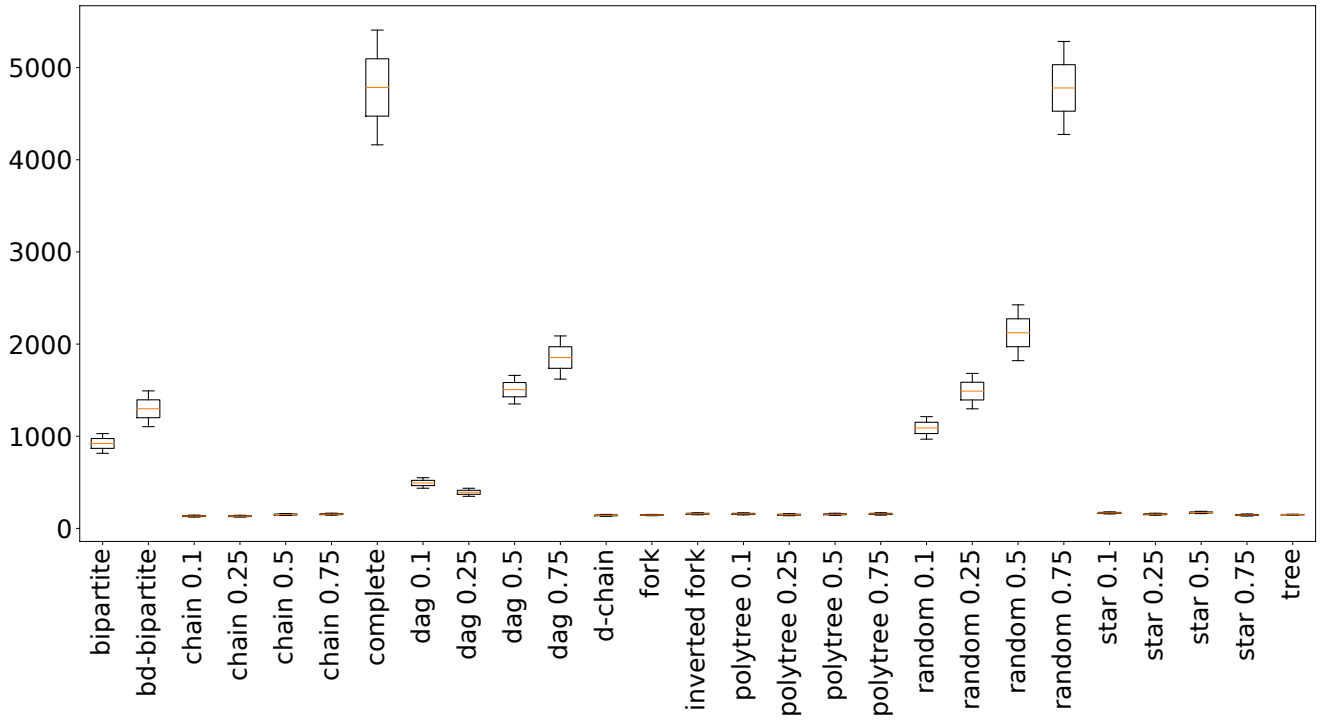


Figure 2: Mean generation time and 95% confidence intervals for each collection of tasks.

nally, the goal is chosen from the last fact layer of the relaxed planning graph, making sure that at least one of the chosen facts is unique to the last fact layer and that at most one fact is chosen per variable. In what follows, we describe how actions are constructed. Starting with the initial state as the first fact layer F_0, we create actions for an action layer L_i by

(I) selecting a subset of facts from the fact layer F_i, ensuring at most one fact is selected per variable,

(II) partitioning the selected set of facts into prevail condition and non-prevail precondition, and

(III) choosing for all[1] the variables of the precondition facts a different value as its effect.

The constructed action is checked against the graph G, ensuring that it contributes only edges that exist in G. If not, the action is discarded. Additionally, if the constructed action does not add any new causal edges and does not achieve new facts, we randomly decide whether to keep it.

The generic approach to action construction described above can be adapted to enforce particular properties. We discuss three such cases in detail.

(A) The first case is enforcing the action to achieve at least one new fact. For that, one can ensure in steps (I) and (II) that the precondition includes facts for some variables $v \in \mathcal{V}$ that are not fully covered by the fact layer F_i (that is $F_v \setminus F_i \neq \emptyset$), and in step (III) to choose one of these facts $F_v \setminus F_i$. Note that this can be done without adding any edges to the causal graph, if a single fact is chosen in step (I).

(B) The second case is enforcing the preconditions to include atoms from $F_i \setminus F_{i-1}$, enforced in step (I).

(C) The third case is enforcing adding a particular edge $\langle v, v' \rangle$ to the causal graph. This can be done by ensuring that $\langle v, \vartheta \rangle$ and $\langle v', \vartheta' \rangle$ are chosen in step (I), and in step (II) at most one of these facts is chosen for the prevail condition.

We randomly independently decide whether to enforce the options (A)-(C) and whether to add edges to the causal graph. Note that not all combinations are always possible. In such cases, an action is not constructed in that iteration. Each layer is constructed until a sufficient number of new facts $F_{i+1} \setminus F_i$ is added. The construction is stopped when all facts were achieved and all edges from G are reflected in the causal graph of the constructed planning task. The latter is enforced in the last layer. The goal is then randomly chosen from the last layer according to step (I), ensuring that at least one of the facts is not achieved before the last layer, analogously to how a precondition of an action is chosen when enforcing the option (B). Algorithm 1 describes the construction of a planning task from the given graph G, where the function CREATEACTION creates a single action,

[1] While SAS$^+$ representation does not require to specify the precondition when the effect is specified, in order to ensure maintaining variables as mutexes of facts, we restrict ourselves here to always specifying the precondition in such cases.

randomly choosing among the options described above.

Theorem 1 *Given G and n, Algorithm 1 terminates in time polynomial in $|G|$ and n and returns a planning task with the causal graph G.*

Proof: The proof follows from the fact that in line 18 of Algorithm 1, for some of the options for action creation must eventually hold $E_a \subseteq E$ and $E_a \setminus E_{A \cup A'} \neq \emptyset$. Therefore, CREATELAYER terminates and returns a layer with m new facts or, if $m = 0$, with A' such that $E_{A \cup A'} = E$. As there are only a constant number of options, a new fact is achieved or a new causal graph edge is covered in time $O(1)$ and therefore CREATELAYER terminates in time $O(m + |E|)$. Since at least one new fact is added in each layer, Algorithm 1 terminates in time $O(n|E|)$. Since the while loop in line 7 terminates only when $E \setminus E_A = \emptyset$ and actions a are added to A only if $E_a \subseteq E$, when the algorithm terminates we have $E_A = E$, and therefore the causal graph of the returned task Π is exactly G. □

In order to create a PDDL task, the SAS$^+$ task is then translated to the STRIPS fragment of PDDL, ignoring the facts that correspond to the last value of the variables representing the at-most-one case. PDDL preconditions are taken from SAS$^+$ preconditions, add effects are taken from SAS$^+$ effects, and delete effects are taken from non-prevail preconditions. Note that if the tasks are translated from STRIPS back to SAS$^+$, there is nothing that enforces that the same mutex groups will be detected, as different planners implement different translation procedures. Thus, the causal graph structure is not necessarily preserved by translating to STRIPS and back to SAS$^+$.

Experimental Evaluation

We start by constructing the benchmark set, as described in the previous section. Our benchmark set was generated as follows. For each of the causal graph structures mentioned above, and a value in $[0.1, 0.25, 0.5, 0.75]$ for edge probability (if needed), we create a collection of tasks. This results in 27 collections in our case, with 7 causal graph structures that do not consider edge probabilities and 5 causal graph structures that do. For each such collection, we generate 512 instances by uniformly choosing the number of atoms (4 variants), variables (4 variants), goal variables (4 variants), maximum prevail size (2 variants), maximum effect size (2 variants), and the upper bound on the minimum number of atoms per layer (2 variants). Thus, our constructed benchmark set consists of 13824 generated planning tasks. The benchmark set is available at https://github.com/IBM/structural-benchmarks-PDDL. To give a general impression of typical generation time, Figure 2 shows the mean generation time and 95% confidence intervals for each collection. It is worth mentioning that while in most collections task generation is typically quick, in some collections, such as *complete* and *random*, it can be quite time consuming. We note that these causal structures are somewhat less interesting. Nonetheless, we have decided to include these collections in our generated set.

Collection	Comp	PDBs	Scorp	LM-cut	SYMBA*
bipartite	105	73	**182**	180	59
bd-bipartite	96	78	**137**	107	72
chain 0.1	358	321	**395**	372	282
chain 0.5	349	327	**389**	359	288
chain 0.25	409	345	**444**	428	309
chain 0.75	320	304	**391**	333	261
complete	123	117	**153**	129	110
dag 0.1	113	94	**179**	166	77
dag 0.5	65	45	108	**121**	25
dag 0.25	66	50	**134**	108	15
dag 0.75	73	62	**113**	110	32
d-chain	382	341	**416**	391	296
fork	350	321	**421**	395	308
inverted fork	**436**	393	356	386	357
polytree 0.1	325	253	**386**	343	224
polytree 0.5	323	269	**373**	338	254
polytree 0.25	367	268	**405**	379	245
polytree 0.75	365	276	**421**	408	260
random 0.1	54	38	**167**	95	32
random 0.5	57	52	**117**	88	34
random 0.25	97	89	**153**	129	83
random 0.75	56	39	**81**	50	35
star 0.1	226	130	**272**	224	125
star 0.5	257	144	**333**	272	139
star 0.25	219	114	**290**	248	115
star 0.75	172	132	**263**	199	134
tree	364	277	**433**	402	245
Sum (13824)	6127	4952	**7512**	6760	4416

Table 1: Per-collection coverage of state-of-the-art planning systems: Complementary (Comp), planning-PDBs (PDBs), Scorpion (Sc), as well as A^* with LM-cut heuristic and SYMBA* planner. Bolded results indicate the best coverage in a collection and overall.

Our evaluation of the constructed set aims at understanding whether the set is sufficiently challenging for modern cost-optimal planners. Therefore, we have selected the top-performing cost-optimal planners from the most recent International Planning Competition (IPC) 2018: Complementary (Franco et al. 2018), Planning-PDBs (Moraru et al. 2018), and Scorpion (Seipp 2018). We excluded the portfolio planner Delfi (Katz et al. 2018), and included instead its top performing components: the symbolic planner SYMBA* (Torralba et al. 2014) and explicit heuristic search with LM-cut heuristic (Helmert and Domshlak 2009),[2] both with h^2 mutex detection (Alcázar and Torralba 2015). The planners

[2]While the components of Delfi also use symmetry based pruning (Domshlak, Katz, and Shleyfman 2012) and partial order reduction (Wehrle and Helmert 2014), here we do not use these pruning techniques.

Collection	Comp	PDBs	Scorp	LM-cut	SYMBA*
bd-bipartite	66	66	66	66	61
chain 0.1	6	6	6	6	5
chain 0.5	8	8	8	8	7
chain 0.25	19	19	19	19	20
chain 0.75	30	30	30	30	21
complete	108	108	108	108	104
fork	25	25	25	25	27
inverted fork	1	1	1	1	8
random 0.5	40	40	40	40	33
random 0.25	50	50	50	50	46
star 0.5	14	14	14	14	16
star 0.25	0	0	0	0	1
star 0.75	14	14	16	14	14
Sum other	154	154	154	154	154
Sum all	535	535	537	535	517

Table 2: Per-collection number of instances that proved to be unsolvable.

are run on the entire constructed benchmark set, with the timeout of 30 minutes and memory limit of 7.6GB allocated to each run. The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines.

Table 1 shows per-collection aggregated coverage comparison of the selected planners. Each task in a collection contributes a value of 1 to the coverage if it was either solved by the planner or the planner was able to prove the task to be unsolvable. Otherwise, the task contributes 0. Separately, Table 2 depicts the number of tasks in each collection that were proved to be unsolvable. Note that the tested planning systems perform very similarly in terms of unsolvability detection. In contrast, when looking at tasks solved, the tested planning systems perform very differently.

Going beyond aggregated coverage results, and focusing on two planners with the lowest total time on average – LM-cut and SYMBA*, Figure 3 shows the per-task total time for these planners. For the tasks solved by both approaches within the time bound, there is no clear advantage to any of the planners.Looking at the timeouts, there are 2963 cases where SYMBA* times outs but LM-cut does not, and 103 cases where LM-cut times outs but SYMBA* does not.

We observe that for each of the tested planners, in each of the collections, there still remains a significant number of tasks not solved. Further, while some causal graph structures correspond to seemingly easier planning tasks, at least for the tested planners, there is a significant number of tasks in each collection that were not solved by any of the tested planners: from 54 in *chain 0.25* to 427 in *random 0.75*, with the average of 219 tasks in a collection, 5917 tasks overall. Clearly, the generated tasks are challenging for the state of the art in cost-optimal classical planning.

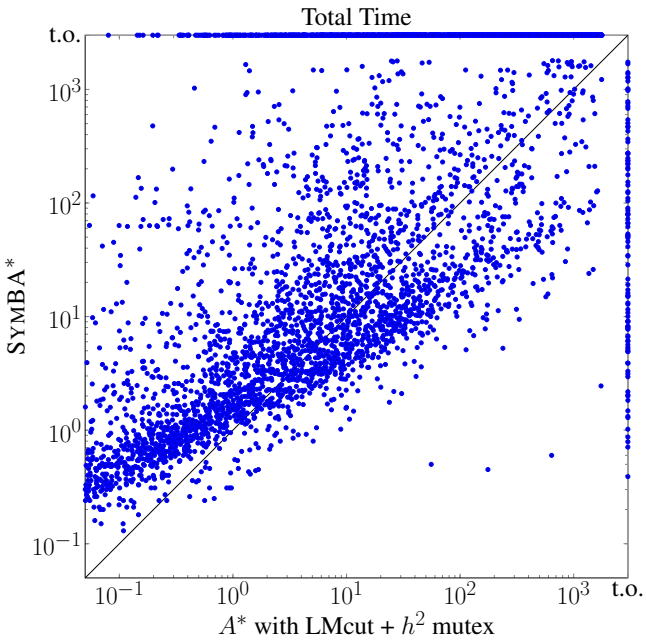


Figure 3: Total time comparison of A^* with LM-cut heuristic to SYMBA* planner.

Related Work

The idea of generating domain models as well as specific planning tasks has been explored in planning community, with a major focus on learning domain models from traces, for classical planning (e.g., (Yang, Wu, and Jiang 2007; Zhuo et al. 2010; Tian, Zhuo, and Kambhampati 2016)) and HTN planning (e.g., (Hogg, Muñoz-Avila, and Kuter 2008; Hogg, Kuter, and Muñoz-Avila 2010; Hogg, Muñoz-Avila, and Kuter 2016)). The work on learning domain models often assumes an existence of a complete model where the plan traces or plan examples are generated from. Some aspects of these domain models are then learned or reconstructed from successful plan traces. Some examples include learning action preconditions (Zhuo et al. 2009), or refine incomplete action descriptions (Zhuo, Nguyen, and Kambhampati 2013).

Probably a more related to our current work is the work on generating problem instances for CSP/SAT problems (e.g., (Achlioptas et al. 2000; Xu et al. 2005)). There are also several online tools/services such as the "Tough SAT Project" or "SATLIB" that generate CNF formulas encoding "difficult" problems (e.g., (Yuen and Bebel 2017; Hoos and Stützle 2000)). Producing hard satisfiable instances has several advantages one of which is to advance the research field in SAT/CSP by providing a suite of problems that can be used for evaluation of solvers. Further, these instances can be polynomial-time reduced to STRIPS in theory, but also in practice (Porco, Machado, and Bonet 2011). The authors provide a tool to translate multiple NP-complete computational problem instances (including SAT, CLIQUE, DirectedHamiltonianPath, etc.) into an NP-Complete fragment of STRIPS that they call STRIPS-1. In that fragment, the actions

are either delete-free or can be applied at most once. While the fragment is somewhat limited, the approach can be used for creating additional benchmark sets for planning. Unfortunately, the work has not yet received the attention it deserves, and the instances or the tool are not currently widely used. It is worth mentioning that our suggested approach to generating random PDDL instances is a somewhat different task than generating random CNF formula, and then translating to STRIPS. Our focus is on being able to control the causal structure of the generated problem, which is not possible with the aforementioned methods.

Another highly related work is the work on random planning tasks generation for the purpose of analyzing the phase transition in classical planning (Bylander 1996; Rintanen 2004). The authors propose a variety of models for sampling the space of STRIPS planning problem instances, exploring the possibility of phase transition at some constant ratio of the number of actions to the number of state variables. These models correspond to a constrained set of problem instances, restricting the sizes of preconditions and effects, and reducing the chances of generating trivially unsolvable tasks. Unfortunately, the proposed methods for generating tasks do not yield tasks of a desired structure and it is not clear what additional restrictions can be imposed in order to obtain such tasks.

Summary

In this work, we have presented an approach that allows to generate planning tasks with the causal graph of a specific given structure. Further, we cast these tasks into a STRIPS fragment of PDDL, allowing using as an input to any PDDL planner that supports the STRIPS fragment, as most mature planning systems do. We have generated a benchmark set of 27 task collections characterized by the causal graph structure, with 512 tasks in each collection, summing up to 13824 ground PDDL tasks in total. Our experimental evaluation clearly shows that the generated benchmark set is challenging for both the heuristic search based and the symbolic search based planners. In the hope to facilitate further research and enable better comparison of planning tools, we make our tool publicly available to the planning community.

For future work, we intend to explore additional structural restrictions of planning tasks, such as, e.g., k-dependence (Katz and Domshlak 2007), as well as possibly additional causal graph structures. Further, we would like to investigate the phase transition in planning according to structural characterization of planning tasks. We conjecture that phase transition might appear at different number of actions to state variables ratios for different causal graph structures. As an additional future work, we would like to explore the usage of generating planning tasks for the purpose of learning a planner selection strategy. Finally, we would like to explore the possibility of generating lifted PDDL tasks of a meaningful causal structure. For that, we would need to understand how to characterize the ground concept of causal graph structure on a lifted level.

References

Achlioptas, D.; Gomes, C. P.; Kautz, H. A.; and Selman, B. 2000. Generating satisfiable problem instances. In *Proc. AAAI 2000*, 256–261.

Aghighi, M.; Jonsson, P.; and Ståhlberg, S. 2015. Tractable cost-optimal planning over restricted polytree causal graphs. In *Proc. AAAI 2015*, 3225—3231.

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. ICAPS 2015*, 2–6.

Bäckström, C., and Jonsson, P. 2013. A refined view of causal graphs and component sizes: SP-Closed graph classes and beyond. *JAIR* 47:575–611.

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

Bäckström, C.; Jonsson, P.; and Ordyniak, S. 2019. A refined understanding of cost-optimal planning with polytree causal graphs. In *Proc. IJCAI 2019*, 6126–6130.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS 2009*, 34–41.

Bylander, T. 1996. A probabilistic analysis of propositional STRIPS planning. *AIJ* 81(1–2):241–271.

Domshlak, C., and Brafman, R. I. 2002. Structure and complexity in planning with unary operators. In *Proc. AIPS 2002*, 34–43.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Proc. ICAPS 2012*, 343–347.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 84–90.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ* 2:189–208.

Franco, S.; Lelis, L. H. S.; Barley, M.; Edelkamp, S.; Martines, M.; and Moraru, I. 2018. The Complementary1 planner in the IPC 2018. In *IPC-9 planner abstracts*, 28–31.

Giménez, O., and Jonsson, A. 2008. The complexity of planning problems with simple causal graphs. *JAIR* 31:319–351.

Giménez, O., and Jonsson, A. 2009. Planning over chain causal graphs for variables with domains of size 5 is NP-hard. *JAIR* 34:675–706.

Giménez, O., and Jonsson, A. 2012. The influence of k-dependence on the complexity of planning. *AIJ* 177:25–45.

Grastien, A., and Scala, E. 2018. Sampling strategies for conformant planning. In *Proc. ICAPS 2018*, 97–105.

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.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS 2007*, 176–183.

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.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2010. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In *Proc. AAAI 2010*, 1530–1535.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proc. AAAI 2008*, 950–956.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning hierarchical task models from input traces. 32(1):3–48.

Hoos, H., and Stützle, T. 2000. SATLIB: An online resource for research on SAT. 283–292.

Katz, M., and Domshlak, C. 2007. Structural patterns of tractable sequentially-optimal planning. In *Proc. ICAPS 2007*, 200–207.

Katz, M., and Domshlak, C. 2008. Structural patterns heuristics via fork decomposition. In *Proc. ICAPS 2008*, 182–189.

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *JAIR* 39:51–126.

Katz, M., and Keyder, E. 2012. Structural patterns beyond forks: Extending the complexity boundaries of classical planning. In *Proc. AAAI 2012*, 1779–1785.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. In *IPC-9 planner abstracts*, 57–64.

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.

McDermott, D. 2000. The 1998 AI Planning Systems competition. *AI Magazine* 21(2):35–55.

Moraru, I.; Edelkamp, S.; Martinez, M.; and Franco, S. 2018. Planning-PDBs planner. In *IPC-9 planner abstracts*, 69–73.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *JAIR* 35:623–675.

Porco, A.; Machado, A.; and Bonet, B. 2011. Automatic polytime reductions of NP problems into a fragment of STRIPS. In *Proc. ICAPS 2011*, 178–185.

Rintanen, J. 2004. Phase transitions in classical planning: an experimental study. In *Proc. ICAPS 2004*, 101–110.

Seipp, J. 2018. Fast Downward Scorpion. In *IPC-9 planner abstracts*, 77–79.

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.

Sohrabi, S.; Riabov, A. V.; Katz, M.; and Udrea, O. 2018. An AI planning solution to scenario generation for enterprise risk management. In *Proc. AAAI 2018*, 160–167.

Tian, X.; Zhuo, H. H.; and Kambhampati, S. 2016. Discovering underlying plans based on distributed representations of actions. In *Proc. AAMAS 2016*, 1135–1143.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A symbolic bidirectional A* planner. In *IPC-8 planner abstracts*, 105–109.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proc. ICAPS 2014*, 323–331.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2005. A simple model to generate hard satisfiable instances. In *Proc. ICAPS 2005*, 337–342.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *AIJ* 171:107–143.

Yuen, H., and Bebel, J. 2017. Tough SAT project.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Muñoz-Avila, H. 2009. Learning HTN method preconditions and action models from partial observations. In *Proc. IJCAI 2009*, 1804–1810.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *AIJ* 174(18):1540–1569.

Zhuo, H. H.; Nguyen, T.; and Kambhampati, S. 2013. Refining incomplete planning domain models through plan traces. In *Proc. IJCAI 2013*, 2451–2457.

Bounding Quality in Diverse Planning

Michael Katz and **Shirin Sohrabi** and **Octavian Udrea**
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
michael.katz1@ibm.com, {ssohrab,udrea}@us.ibm.com

Abstract

Diverse planning is an important problem in automated planning with various real world applications. Recently, diverse planning has seen renewed interest, with work that defines a taxonomy of computational problems with respect to both plan quality and solution diversity. However, despite the recent advances in diverse planning the variety of approaches and the number of available tools for these problems are still quite limited, even nonexistent for several computational problems. In this work, we aim to extend the portfolio of approaches and tools for various computational problems in diverse planning. To that end, we introduce a novel approach to finding solutions for three computational problems within diverse planning and present planners for these three problems. For one of these problems, our approach is the first one that is able to provide solutions to the problem. For another, we show that top-k and top quality planners can provide, albeit naive, solutions to the problem and we extend these planners to improve the diversity of the obtained solution. Finally, for the third problem, we show that some existing diverse planners already provide solutions to the problem. Further, we suggest another approach and empirically show that our suggested approach compares favorably with these existing planners.

Introduction

Diverse planning is an important problem in AI Planning with many practical applications that require generating multiple plans rather than one. Example applications include automated machine learning (Mohr, Wever, and Hüllermeier 2018), risk management (Sohrabi et al. 2018), automated analysis of streaming data (Riabov et al. 2015), and malware detection (Boddy et al. 2005). Diverse planning is also important in the context of re-planning and plan monitoring (Fox et al. 2006), under-specified user preferences (Myers and Lee 1999; Nguyen et al. 2012), as well as plan recognition and its related applications (Sohrabi, Riabov, and Udrea 2016). In all these applications it is important to generate multiple diverse plans, and it is of equal importance to be able to control solution quality.

Most diverse planners developed over the last decade are focused on addressing a particular diversity metric. For example, while the diverse planner DLAMA focuses on finding a set of plans by considering a landmark-based diversity measure (Bryce 2014), other diverse planners such as LPG-d, DIV, DFAA/DFAM, and A*AA/A*AM focus on finding a set of plans with a particular minimum action distance (Nguyen et al. 2012; Coman and Muñoz-Avila 2011; Vadlamudi and Kambhampati 2016). Goldman and Kuter (2015) propose a diversity metric based on information retrieval literature. Roberts, Howe, and Ray (2014) suggest another diversity metric, introducing several planners, such as itA^* and MQA, which, in addition to the diversity metrics, consider plan quality. While all these planners implement the chosen diversity metric and switching to another metric is not trivial, the planners DFAA/DFAM and A*AA/A*AM work in two phases: finding a set of plans and choosing a proper subset from the found set. That is, selection of a set of plans is independent of the diversity metric. Recently, planners *FI*-diverse were introduced (Katz and Sohrabi 2020). These planners also separate the phase of finding candidate plans from choosing a diverse subset of these plans. Further, the authors provide a tool for selecting a subset of plans for a variety of metrics and computational problems (Katz and Sohrabi 2019).

Another important recent contribution introduced a taxonomy of computational problems and classified existing planners according to the problems they tackle (Katz and Sohrabi 2020). Most existing planners, according to that taxonomy, solve *Satisficing Diverse Planning* (sat-k), where any sufficiently large set of plans is a solution, and the aim is to improve solution diversity. The planners LPG-d (Nguyen et al. 2012) and bFI (Katz and Sohrabi 2020) tackle *Bounded Diversity Diverse Planning* (bD-k), where a set of plans is a solution only if its diversity is above a certain specified bound. Top-k planners (Katz et al. 2018b; Speck, Mattmüller, and Nebel 2020) and top-quality planners (Katz, Sohrabi, and Udrea 2020), while usually are not considered as diverse planners, according to the aforementioned taxonomy return, albeit naive, solutions to the *Bounded Quality Diverse Planning* (bQ-k) problem, where plan set quality is required to be above a specified bound. The planners DFAA/DFAM and A*AA/A*AM (Vadlamudi and Kambhampati 2016) tackle *Bounded Quality and Diversity Diverse Planning* (bQbD-k), where both the quality and the diversity of plan sets is bounded from above.

Despite these recent advances in diverse planning, the pool of existing tools is still quite limited. The planners DFAA/DFAM and A*AA/A*AM by Vadlamudi and Kambhampati (2016) are the only existing planners for bQbD-k. Top-quality planners (Katz, Sohrabi, and Udrea 2020), although technically solving bQ-k, do not aim at improving the diversity of the solution. No planners exist for other computational problems, such as *Optimal Diversity Bounded Quality Diverse Planning* (bQoptD-k), where a solution corresponds to a set of plans of best diversity among the sets of bounded quality.

In this work, expand the pool of available planners for diverse planning. We introduce novel planners for the three aforementioned computational problems, bQ-k, bQbD-k, and bQoptD-k, exploiting the recently introduced top-quality planners. To that end, we introduce a novel quality metric that reflects bounded plan costs, making a connection between the costs of plans in a set and quality of a set of plans. Focusing on the most popular diversity metric (Nguyen et al. 2012; Coman and Muñoz-Avila 2011; Vadlamudi and Kambhampati 2016), for all three planners we generate a subset of all plans of bounded quality as a first step. In the second step, we select a subset of plans found in the first step that constitutes a solution to the respective computational problem. For bQ-k, as any sufficiently large subset of plans is, albeit naive, a solutions to the problem, we extend these planners by using a previously suggested greedy algorithm to choose a subset of plans of higher diversity (Katz and Sohrabi 2020). For bQbD-k, we show that the decision problem that corresponds to the second step is NP-complete and suggest using previously proposed integer linear programming formulation. As the formulation was not previously formally described, we describe it formally and prove that it provides us with a solution to bQbD-k. For bQoptD-k, as the optimization problem in second step is NP-hard, we propose using a novel mixed integer linear program to solve it. We formally describe the program and prove that a solution to the program can be used for solving bQoptD-k. Our approach is the first one that is able to provide solutions to the problem.

Finally, we perform an empirical evaluation of our proposed planners. For bQbD-k, we show to favorably compare to the existing planners. For bQ-k, as no previous planners exist, we test the quality of our solution by comparing it to the quality of the optimal solution, obtained by our proposed planner for bQoptD-k, where no previous planners exist either. We show that the greedy algorithm works very well on tested domains, producing results close to the optimum. Our novel contributions, thus, include (i) the introduction of the new quality metric that allows us to connect between the cost of plans and quality of a set of plans, (ii) a concrete algorithmic scheme that uses top quality planners for the first step, finding a set of plans of bounded cost, (iii) computational complexity investigation of the second step, choosing a proper subset from the found set, for various computational problems, and (iv) introduction of the new mixed integer linear program for the resulting optimization problem.

Preliminaries and Related Work

In this work we follow the notation of Katz and Sohrabi (2020). A SAS$^+$ *planning task* (Bäckström and Nebel 1995) is given by a tuple $\langle \mathcal{V}, \mathcal{A}, s_0, s_* \rangle$, with $\mathcal{V}$ being a set of *state variables* and $\mathcal{A}$ being a finite set of *actions*. Each state variable $v \in \mathcal{V}$ has a finite domain $dom(v)$ of values. A pair $\langle v, \vartheta \rangle$ with $v \in \mathcal{V}$ and $\vartheta \in dom(v)$ is called a *fact*. A (partial) assignment to $\mathcal{V}$ is called a *(partial) state*. Often it is convenient to view partial state p as a set of facts with $\langle v, \vartheta \rangle \in p$ if and only if $p[v] = \vartheta$. Partial state p is *consistent* with state s if $p \subseteq s$. We denote the set of states of a planning task by $\mathcal{S}$. s_0 is the *initial state*, and the partial state s_* is the *goal*. Each *action* a is a pair $\langle pre(a), \mathit{eff}(a) \rangle$ of partial states called *preconditions* and *effects*. An *action cost* is a mapping $C : \mathcal{A} \to \mathbb{R}^{0+}$. An action a is applicable in a state $s \in \mathcal{S}$ if and only if $pre(a)$ is consistent with s. Applying a changes the value of v to $\mathit{eff}(a)[v]$, if defined. The resulting state is denoted by $s[\![a]\!]$. An action sequence $\pi = \langle a_1, \ldots, a_k \rangle$ is applicable in s if there exist states $s_0, \cdots, s_k$ such that (i) $s_0 = s$, and (ii) for each $1 \leq i \leq k$, a_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![a_i]\!]$. We denote the state s_k by $s[\![\pi]\!]$. π is a plan iff π is applicable in s_0 and s_* is consistent with $s_0[\![\pi]\!]$. We denote by $\mathcal{P}(\Pi)$ (or just $\mathcal{P}$ when the task is clear from the context) the set of all plans of Π. The cost of a plan π, denoted by $C(\pi)$ is the summed cost of the actions in the plan.

In regard to reasoning about sets of plans rather than individual plans, there are two main measures defined on sets of plans, *quality* and *diversity*. Previous work has introduced one definition of quality, mirroring the International Planning Competition (IPC) quality metric for individual plans (Katz and Sohrabi 2020).

Definition 1 *Let P be the set of known plans of Π and let $P' \subseteq P$ be a subset of plans. The relative quality of P' with respect to P is defined as*

$$Q_P(P') := \frac{1}{|P'|} \times \sum_{i=1}^{|P'|} \frac{C(\pi_i)}{C(\pi'_i)},$$

where $\pi_1, \ldots, \pi_{|P'|}$ and $\pi'_1, \ldots, \pi'_{|P'|}$ are plans in P and P', respectively, sorted in ascending order of their costs.

The relative quality of a set of plans is always between 0 and 1, being 1 if and only if there is no plan in $P \setminus P'$ that is cheaper than any plan in P'.

Switching now our attention to diversity metrics, pairwise plan distance is defined by $\delta(\pi, \pi') = 1 - \mathrm{sim}(\pi, \pi')$, where sim is a similarity measure, a value between 0 (unrelated) and 1 (equivalent). The diversity of a set of plans, $D(P)$, $P \subseteq \mathcal{P}$ is then defined as some aggregation (e.g., min or average) of the pairwise distance within the set P. In this work, we focus on one of the most popular similarity measures, *stability* (Fox et al. 2006; Coman and Muñoz-Avila 2011). Stability similarity measures the ratio of the number of actions that appear on both plans to the total number of actions on these plans, referring to plans as action multi-sets (sets with repetitions). Given two plans π, π', it is defined as $\mathrm{sim}_{\text{stability}}(\pi, \pi') = |A(\pi) \cap A(\pi')| / |A(\pi) \cup A(\pi')|$, where $A(\pi)$ is the multi-set of actions in π. In what follows,

by D_{ma} we denote the diversity metric computed as minimum over the pairwise plan distance under stability similarity, the diversity metric implemented by multiple existing diverse planners (Nguyen et al. 2012; Coman and Muñoz-Avila 2011; Vadlamudi and Kambhampati 2016).

There is a variety of computational problems that fall under the umbrella of diverse planning. In our work, focusing on bounded quality problems, we follow the recently introduced taxonomy (Katz and Sohrabi 2020).

Definition 2 (Diverse planning solution) *Let Π be a planning task and $\mathcal{P}$ be the set of all plans for Π. Given a natural number k, $P \subseteq \mathcal{P}$ is a k-diverse planning solution if $|P| = k$ or $P = \mathcal{P}$ if $|\mathcal{P}| < k$.*

Definition 3 (Quality-bounded solution) *Let Π be a planning task, Q be some quality metric, c be some bound, and $\mathcal{P}$ be the set of all Π's plans. Given a natural number k, $P \subseteq \mathcal{P}$ is a c-quality-bounded k-diverse planning solution if it is a k-diverse planning solution and $Q(P) \geq c$.*

Bounded Quality Diverse Planning computational problem is defined as follows.

bQ-k : Given k and c, find a c-quality-bounded k-diverse planning solution.

Definition 4 (Diversity-bounded solution) *Let Π be a planning task, D be some diversity metric, b be some bound, and $\mathcal{P}$ be the set of all Π's plans. Given a natural number k, $P \subseteq \mathcal{P}$ is a b-diversity-bounded k-diverse planning solution if it is a k-diverse planning solution and $D(P) \geq b$.*

Bounded Quality and Diversity Diverse Planning computational problem is defined as follows.

bQbD-k : Given $k, b,$ and c, find a c-quality-bounded and b-diversity-bounded k-diverse planning solution.

Note that the definition above generalizes the previously suggested search problem described in Equation 1 below and implemented for the diversity metric D_{ma} by Vadlamudi and Kambhampati (2016).

$$\text{cCOSTdDISTANTkSET : find } P \text{ with } P \subseteq \mathcal{P}, \quad |P| = k, \min_{\pi,\pi' \in P} \delta(\pi, \pi') \geq d, C(\pi) \leq c \ \forall \pi \in P. \tag{1}$$

Finally, Optimal Diversity Bounded Quality Diverse Planning optimization problem is defined as follows.

bQoptD-k : Given k and c, find a diversity-optimal among c-quality-bounded k-diverse planning solutions.

Bounded Quality in Diverse Planning

As stated above, in this work, we focus on the three computational problems in diverse planning taxonomy of Katz and Sohrabi (2020) that deal with bounded quality, bQ-k, bQbD-k, and bQoptD-k. Our proposed solutions to these three problems all have in common the first step - finding a set of plans of bounded quality. While existing planners for bounded quality diverse planning took the same approach (Vadlamudi and Kambhampati 2016), they used planners for the top-k planning problem (Riabov, Sohrabi, and Udrea 2014). Specifically, A*AA/A*AM apply the m-A^* algorithm (Flerova, Marinescu, and Dechter 2016), while DFAA/DFAM apply the well-known branch-and-bound algorithm. We suggest using a different approach, generating plans with a planner for a recently proposed unordered top-quality problem (Katz, Sohrabi, and Udrea 2020). Switching to top-quality allows to ensure that *all* plans of bounded cost are found. Unordered top-quality allows to disregard plans that are reorderings of the found plans. For some diversity metrics, this is highly beneficial, with pairwise diversity between a plan and its reordering might be very low. Further, ignoring plan orders reduces the computational effort required for finding all plans of bounded cost.

The second step, after finding a set of plans of bounded quality, is different for the different computational problems that we consider in this work. For bQ-k, although any set of k plans is a solution, we strive to obtain solutions of higher diversity. Therefore, we apply a greedy algorithm that iteratively increases the set of plans by adding at each step the candidate plan that increases the overall diversity score the most, the same algorithm that was used for satisficing diverse planning (Katz and Sohrabi 2020). For bQbD-k and bQoptD-k, we cast the problem of finding the subset of plans as (mixed) integer linear programs. We describe these programs in detail in what follows. We start with the discussion of the quality metric that we consider in this work.

Quality Metric

While we consider the planners DFAA/DFAM and A*AA/A*AM to be solving the Bounded Quality and Diversity Diverse Planning problem as defined by Katz and Sohrabi (2020), the quality metric they maximize is not obvious. These planners consider the set P of plans of cost smaller or equal than a given absolute bound value c, or $\max_{\pi \in P} C(\pi) \leq c$. Alternatively, the criterion can be expressed via $Q_a(P) \geq c'$ for the quality measure

$$Q_a(P) = \frac{c^*}{\max_{\pi \in P} C(\pi)} \tag{2}$$

where c^* is the task's optimal plan cost and $c' = \frac{c^*}{c} \in [0, 1]$. Thus, the planners above solve the bQbD-k problem for the quality metric Q_a as in Eq. 2. Note that this quality measure is different from the measure described in Definition 1, as introduced by Katz and Sohrabi (2020), where the quality of the set of plans is affected by the costs of all plans, not only most expensive ones. The above proposed quality metric makes it possible to connect the quality metric to the cost bound.

Bounded Quality Planning

Let us consider now the (unordered) top quality planners (Katz, Sohrabi, and Udrea 2020), that, given a multiplier $q_m \geq 1$, return the set of all plans P such that $\forall \pi \in P$ we have $C(\pi) \leq q_m \times c^*$, or a subset thereof with a single representative for plans that differ only in the order of their actions for the unordered case. For such sets, we have

$Q_a(P) \geq \frac{1}{q_m}$, and therefore these planners can be used to derive solutions of bounded quality according to the quality metric Q_a. Furthermore, top quality planners produce a set of plans that is a super-set of the sets of plans that constitute solutions to all three computational problems of interest, bQ-k, bQbD-k, and bQoptD-k. Therefore, in what follows, we will focus on finding subsets of plans out of a given set of plans, according to the relevant solution definition for the corresponding computational problem.

Focusing first on bQ-k, while any subset of required size of the set of plans returned by top quality planners is a solution to bQ-k, different subsets can vary significantly in their diversity measure score. Since bQ-k does not pose any restrictions on these subsets beyond the desired size, one possible way of coming up with subsets of high diversity is to employ the same greedy selection algorithm that was used for Satisficing Diverse Planning (Katz and Sohrabi 2020). The algorithm iteratively constructs a set of plans by greedily adding a plan that will contribute the most to already added plans.

Bounding Diversity

Switching now our attention to bQbD-k, first, note that for a set of plans and a number k, the decision problem of whether there exists a subset of bounded diversity of size k is NP-complete. The membership in NP is trivial. We show the hardness by a polynomial reduction from the clique problem (Garey and Johnson 1979).

For a graph $G = (V, E)$, let $P_G = \{\pi_v \mid v \in V\}$ be a collection of plans. For a pair of plans $\pi_u, \pi_v \in P$

$$d(\pi_u, \pi_v) = \begin{cases} d, & (u, v) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 1 *Given a number k and diversity bound $d > 0$ for diversity metrics minimal pairwise diversity, if there is a subset of plans P_G of bounded by d diversity of size at least k, then there is a clique in G of the same size.*

Proof: Let $P \subseteq P_G$ be a subset of plans such that $|P| \geq k$ and $D(P) \geq d$. Then for all $\pi_u, \pi_v \in P$ we have $d(\pi_u, \pi_v) \geq d$ and therefore $d(\pi_u, \pi_v) > 0$. Thus, it must be the case that $(u, v) \in E$ for all $\pi_u, \pi_v \in P$ and thus the set $V' = \{v \mid \pi_v \in P\}$ is a clique, of size $|V'| = |P| \geq k$. ■

Next, we describe the mixed integer linear program that is used for finding a subset of plans of bounded diversity. While the program is not novel,[1] its description was not presented in the literature. Here, we describe the program in detail. Given a set of plans P and a bound on the diversity d, the variables are as follows.

- A binary variable v_π per plan $\pi \in P$, describing whether the plan is selected for the subset.

The constraints are as follows.

(i) $\forall \pi, \pi' \in P$, s.t. $d(\pi, \pi') < d : v_\pi + v_{\pi'} \leq 1$, stating that if the pairwise diversity of π and π' is below d, then at most one of these plans can be selected for the subset, and

(ii) $\sum_{\pi \in P} v_\pi \geq k$, forcing the size of the subset be at least k.

The objective of the program is to minimize $\sum_{\pi \in P} v_\pi$. In words, the program encodes a subset selection and restrict the selected subset to not have pairs of plans with diversity outside of the provided bound. In what follows, we prove that the program can be used for devising solutions for bQbD-k.

Theorem 2 *For a planning task Π with a set of all plans of bounded quality P such that $|P| \geq k$, and a bound d, the binary program finds a subset of size k with the bounded by d diversity score, for diversity metrics maximizing minimal pairwise diversity, if such subset exists. Otherwise, the program is infeasible.*

Proof: We first show that a solution to bQbD-k corresponds to a feasible assignment. Let $P' \subset P$ be a solution to bQbD-k for the bound d, with $|P'| = k$. Then, let $\overline{v}$ assign 1 to plans $\pi \in P'$ and 0 otherwise. Since $|P'| = k$, constraint (ii) holds. For $\pi, \pi' \in P$, if either of the plans is not in P', then $\overline{v_\pi} + \overline{v_{\pi'}} \leq 1$. If both plans are in P', then $d(\pi, \pi') \geq d$. Thus, constraint set (i) holds for $\overline{v}$ and the program is feasible.

Now, let $\overline{v}$ be some feasible solution and let $P' = \{\pi \in P \mid \overline{v_\pi} = 1\}$ be the corresponding subset of P. Then, (a) from constraint (ii) we have $|P'| \geq k$, and (b) for all $\pi, \pi' \in P'$, since $\overline{v_\pi} + \overline{v_{\pi'}} = 2$, we know that the corresponding constraint is not in the constraint set (i), and therefore $d(\pi, \pi') \geq d$. Therefore, P' (or any of its subset of size k) is a solution to bQbD-k. ■

We now switch our attention to the next computational problem, bQoptD-k.

Optimizing Diversity

Due to the NP-completeness of the decision problem of selecting a subset of plans of bounded diversity, the corresponding optimization problem is NP-hard. To solve it efficiently, we encode it in mixed integer linear programming. We present a novel mixed integer linear program that we use for finding a subset of size k that optimizes the diversity metric. Given a set of plans P, we define the variables as follows.

- A binary variable v_π per plan $\pi \in P$, describing whether the plan is selected for the subset, and
- a single continuous variable d for bounding the pairwise diversity.

The constraints are as follows.

(i) $\sum_{\pi \in P} v_\pi = k$, stating that the size of the subset is exactly k, and

(ii) $\forall \pi, \pi' \in P : d + v_\pi + v_{\pi'} \leq d(\pi, \pi') + 2$, stating that d is bounded by the diversity of each chosen pair, if the pair is chosen.

[1] The mixed integer linear program was previously used for bounded diversity diverse planning (Katz and Sohrabi 2020)

	$q_m = 1.00$				$q_m = 1.05$				$q_m = 1.10$				$q_m = 1.20$			
Coverage	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym
airport (28)	0	7	**18**	7	0	7	**18**	7	0	7	**17**	7	0	7	**17**	7
barman11 (8)	0	0	4	**8**	0	0	**5**	4	0	0	**5**	4	0	0	**5**	4
barman14 (4)	0	0	3	**4**	0	0	**3**	**3**	0	0	**3**	**3**	0	0	2	**4**
blocks (30)	12	21	18	**28**	11	20	19	**29**	18	20	18	**28**	20	20	17	**22**
childsnack14 (6)	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**
data-ntwrk18 (13)	9	7	9	**11**	**10**	7	**10**	**10**	9	7	**10**	**10**	9	7	**10**	**10**
depot (12)	0	1	**3**	**3**	0	1	**3**	**3**	2	1	**3**	**3**	2	1	**3**	**3**
driverlog (14)	3	6	**10**	9	2	6	**10**	9	6	2	**10**	9	8	2	**10**	9
elevators08 (24)	4	2	**7**	0	4	2	**7**	0	4	2	**5**	0	2	2	**6**	0
elevators11 (18)	3	1	**5**	0	2	1	**5**	0	2	1	**5**	0	2	1	**5**	0
floortile11 (14)	2	0	2	**5**	2	0	2	**5**	2	0	2	**5**	2	0	2	**5**
floortile14 (20)	0	0	0	**8**	0	0	0	**8**	0	0	0	**7**	0	0	0	**8**
freecell (22)	6	6	7	**21**	6	6	7	**21**	6	6	7	**21**	8	6	7	**18**
ged14 (19)	12	13	12	**15**	12	13	12	**15**	12	13	12	**15**	12	13	12	**15**
grid (2)	**2**	1	1	**2**	**2**	1	1	**2**	**2**	1	1	**2**	**2**	1	1	**2**
gripper (20)	3	2	6	**15**	3	2	6	**15**	4	2	6	**15**	8	2	6	**15**
hiking14 (19)	3	2	**7**	**7**	3	2	**8**	6	2	1	**6**	**6**	3	1	**8**	5
logistics00 (20)	0	3	**15**	5	0	3	**12**	4	0	3	**10**	3	1	0	**6**	3
logistics98 (6)	1	0	**5**	2	1	0	**5**	2	3	0	**5**	2	3	0	**5**	2
miconic (143)	**121**	70	53	79	**121**	67	54	70	**122**	68	53	63	**118**	67	53	57
movie (30)	**30**	**30**	**30**	2	**30**	**30**	**30**	2	**30**	**30**	**30**	2	**30**	**30**	**30**	0
mprime (24)	6	**21**	20	17	6	**21**	20	17	6	**21**	20	17	15	**21**	19	16
mystery (17)	1	**16**	**16**	11	1	**16**	**16**	12	1	**16**	15	12	10	**16**	15	11
nomystery11 (17)	4	11	11	**13**	7	7	10	**13**	10	6	9	**12**	11	4	9	**13**
openstacks08 (30)	**26**	4	17	21	**26**	4	17	21	**26**	4	17	21	**26**	4	17	21
openstacks11 (20)	**18**	1	12	16	**18**	1	12	15	**18**	1	12	15	**18**	1	12	16
openstacks14 (18)	**5**	0	2	1	**5**	0	2	2	**5**	0	2	1	**6**	0	2	1
openstacks (17)	**5**	0	**5**	0	**5**	0	**5**	0	**5**	0	**5**	0	0	0	**5**	0
organic-s18 (7)	6	**7**	**7**	**7**	6	**7**	**7**	**7**	6	**7**	**7**	**7**	6	**7**	**7**	**7**
organic-ssp18 (15)	11	**14**	**14**	12	11	**14**	**14**	13	11	14	**15**	12	11	14	**15**	11
parcprinter08 (30)	0	6	**15**	6	1	4	**13**	6	1	4	**13**	6	1	4	**13**	6
parcprinter11 (20)	0	3	**11**	3	1	1	**9**	3	1	1	**9**	3	1	1	**8**	3
parking11 (3)	**2**	1	0	0	**1**	**1**	**1**	0	**1**	**1**	**1**	0	**2**	1	1	0
parking14 (4)	**2**	**2**	0	0	**2**	**2**	0	0	**1**	**1**	0	0	**2**	1	0	0
pegsol08 (30)	9	21	28	**29**	9	21	27	**29**	9	21	28	**29**	20	20	28	**29**
pegsol11 (20)	9	9	18	**19**	9	9	17	**19**	11	8	18	**19**	16	6	17	**19**
pipes-notank (22)	**14**	11	**14**	12	**14**	11	**14**	12	**14**	11	**14**	12	**16**	11	14	10
pipes-tank (18)	7	6	**13**	8	7	6	**13**	9	5	6	**14**	7	7	6	**13**	6
psr-small (50)	4	31	**46**	44	4	30	42	**45**	5	29	40	**43**	13	26	38	**42**
rovers (12)	3	4	**6**	**6**	2	4	**6**	**6**	4	4	5	**6**	**6**	4	5	**6**
satellite (15)	7	5	7	**11**	6	5	7	**11**	6	5	7	**9**	6	5	**7**	**7**
scanalyzer08 (19)	12	**13**	12	12	7	**13**	**13**	12	6	**13**	**13**	10	8	**13**	**13**	10
scanalyzer11 (15)	**11**	10	10	9	6	**10**	**10**	9	5	**10**	**10**	7	7	**10**	**10**	7
snake18 (11)	**5**	4	3	3	**5**	4	3	3	**6**	4	3	3	**7**	4	3	3
sokoban08 (30)	5	**6**	0	0	5	**6**	0	0	3	**6**	0	0	3	**6**	0	0
sokoban11 (20)	**3**	**3**	0	0	**3**	**3**	0	0	1	**3**	0	0	0	**3**	0	0
spider18 (11)	5	**7**	6	2	6	**7**	4	3	**8**	7	5	3	**8**	7	5	3
storage (18)	8	11	**16**	14	8	11	**16**	14	7	11	**16**	14	11	11	**16**	13
termes18 (16)	0	0	0	**10**	0	0	0	**9**	0	0	0	**10**	0	0	0	**10**
tetris14 (9)	1	1	3	**5**	1	1	3	**5**	2	1	3	**5**	4	1	3	**5**
tidybot11 (16)	1	5	6	**10**	4	3	4	**10**	**12**	2	3	9	**12**	1	3	9
tidybot14 (10)	**4**	2	0	3	**6**	2	0	2	**6**	2	0	3	**7**	2	0	3
tpp (11)	0	3	**6**	4	0	3	**6**	4	0	3	**6**	4	0	3	**6**	4
transport08 (12)	4	5	8	**11**	5	5	**8**	4	6	5	**8**	4	6	5	**8**	4
transport11 (8)	2	1	3	**7**	2	1	**3**	**3**	2	1	**3**	**3**	2	1	**3**	**3**
transport14 (7)	1	0	2	**5**	0	0	2	**3**	0	0	**2**	**2**	1	0	**2**	**2**
trucks (12)	6	3	**7**	5	5	3	**7**	6	6	3	**7**	5	**7**	3	**7**	5
visitall11 (12)	7	9	9	**11**	8	9	9	**10**	**9**	**9**	**9**	7	**9**	**9**	**9**	5
visitall14 (6)	**6**	3	3	5	**6**	3	3	2	**6**	3	3	0	**6**	3	3	0
woodwork08 (28)	8	7	10	**22**	10	6	10	**22**	14	6	10	**22**	12	6	10	**22**
woodwork11 (20)	7	2	5	**16**	8	2	4	**16**	9	2	5	**16**	7	2	5	**16**
zenotravel (13)	7	**9**	8	**9**	7	9	8	**10**	6	9	8	**10**	8	9	8	**10**
Sum other(27)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum (1192)	453	449	594	**631**	452	433	582	**603**	484	424	573	**574**	548	411	**564**	548

Table 1: Domain-wise coverage comparison of FI-bQbD and Sym-bQbD to DFAM and A*AM, for $k = 5$, diversity bound 0.15, and four quality bounds.

	$q_m = 1.00$				$q_m = 1.05$				$q_m = 1.10$				$q_m = 1.20$			
k	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym	DFA	A*A	FI	Sym
10	390	373	530	**594**	389	354	511	**559**	426	338	494	**535**	**505**	318	480	499
100	189	238	433	**491**	194	216	403	**448**	229	185	365	**396**	291	123	302	**352**
1000	79	238	380	**435**	82	216	345	**386**	105	185	287	**321**	124	123	203	**257**

Table 2: The overall coverage comparison of FI-bQbD and Sym-bQbD to DFAM and A*AM, for diversity bound 0.15, four quality bounds, and various k values.

The objective of the program is then to maximize d. In words, as in the previous case, the program encodes a subset selection, but in this case all subsets of size k correspond to valid assignments. We additionally have a continuous variable d that is bounded by the diversity score of the selected subset. In the case of diversity metrics that correspond to minimal pairwise diversity, this would mean to require the variable d to be bounded by the diversity of each selected pair. In other words, if a pair of plans is selected, then d should be no greater than their diversity score. If a pair is not selected, there is no such restriction, but since there is a natural upper bound of 1 on the overall diversity, d can be required to be upper bounded by any value that is larger or equal to 1. If at least one of v_π, $v_{\pi'}$ gets 0 assigned to it, the constraint $d + v_\pi + v_{\pi'} \leq d(\pi, \pi') + 2$ is then satisfied. Therefore, the constraint is valid whether the variables v_π and $v_{\pi'}$ are assigned 0 or 1.

In what follows, we prove that the program can be used for devising solutions for bQbD-k.

Theorem 3 *For a planning task Π with a set of all plans of bounded quality P such that $|P| \geq k$, the mixed integer program finds a subset of size k with the optimal diversity score, for diversity metrics maximizing minimal pairwise diversity.*

Proof: Let $\overline{v}, \overline{d}$ be a feasible assignment to the variables of the mixed integer program and let $P' = \{\pi \in P \mid \overline{v_\pi} = 1\}$ be the corresponding subset of P. Then, from the constraint set (i) we have $|P'| = k$ and from constraint set (ii) we have $d \leq d(\pi, \pi')$ for all $\pi, \pi' \in P'$. Further, for a plan $\pi \in P \setminus P'$ and a plan $\pi' \in P'$ we have $d \leq 1 + d(\pi, \pi')$, which does not pose additional constraint on the values of d since all pairwise distances $d(\pi, \pi')$ are upper-bounded by 1. Similarly, for $\pi, \pi' \in P \setminus P'$, we have $d \leq 2 + d(\pi, \pi')$, which also does not pose additional constraint on the values of d. Therefore we have $\overline{d} \leq d(\pi, \pi')$ for all $\pi, \pi' \in P'$ and maximizing d without changing $\overline{v}$ would lead to $\overline{d} = \min_{\pi, \pi' \in P'} d(\pi, \pi')$. Thus, the linear program finds a subset of size k with maximum minimal pairwise diversity. ∎

Experimental Evaluation

To empirically evaluate the feasibility of our suggested approach, we have implemented our diverse planners on top of the Diversity Score Computation component (Katz and Sohrabi 2019), using CPLEX v12.8.0 for solving the mixed integer linear programs. The code is available at https://github.com/IBM/diversescore. The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines, with the time and memory limit of 30min and 2GB, respectively. The benchmark set consists of all STRIPS benchmarks from optimal tracks of International Planning Competitions (IPC) 1998-2018, a total of 1797 tasks in 64 domains. For Bounded Quality and Diversity Diverse Planning (bQbD-k), we compare to the existing planners for that computational problem DFAM and A*AM (Vadlamudi and Kambhampati 2016). Since these planners are implemented for the diversity metric D_{ma}, we focus our experimental evaluation on D_{ma}, although our approach works with any metric. Further, since these planners require an absolute bound on the solution cost to be provided as a parameter, we further restrict the benchmark set to tasks where optimal costs could be found with a state-of-the-art cost-optimal planner. For that, we used the 17 single planners from the portfolio of Delfi1 (Katz et al. 2018a). As a result, for the bQbD-k computational problem, the benchmark set consists of 1192 tasks.

As a first step, we generate a set of plans of bounded quality. Focusing on D_{ma} allows us to use unordered top-quality planners (Katz, Sohrabi, and Udrea 2020) to derive all plans (modulo reorderings) of bounded cost. This is due to the fact that two plans that differ only in the order of their actions would produce pairwise diversity of 0 and thus any set of plans P that includes two such plans would get $D_{ma}(P) = 0$. For other diversity metrics we might need to produce the set of all plans of bounded cost (Katz, Sohrabi, and Udrea 2020). Note that some top-k planners, such as K^*-based (Katz et al. 2018b) and symbolic search based (Speck, Mattmüller, and Nebel 2020) can be easily adapted to produce solutions for top-quality planning. Further, these two planners can be rather naively adapted to produce unordered toq-quality solutions, by performing a duplicate check and skipping plans for which a reordering was previously found. In our experiments, we have performed the first step with each of these three planners, namely *FI*, K^*, and *Sym*. We run these planners with a 29min time bound, to allow at least one minute for the second step. In all cases, the overall time bound for both steps is 30min. Further, to avoid generating a larger amount of plans, the overall bound on the number of generated plans for the first step is set to 10000. As a second step, we select a subset of plans according to the computational problem of interest. For bQ-k, we use the greedy approach suggested by Katz and Sohrabi (2020). For the bQbD-k and bQoptD-k computational problems, we solve a mixed-integer linear program, as described in the previous section. This results in three configurations for each computational problem of interest. For space reasons, in what follows, we focus on the two best performing ones, *FI* and *Sym*.

Table 1 presents a domain-wise comparison of our planners, FI-bQbD and Sym-bQbD to the existing planners DFAM and A*AM (Vadlamudi and Kambhampati 2016), for

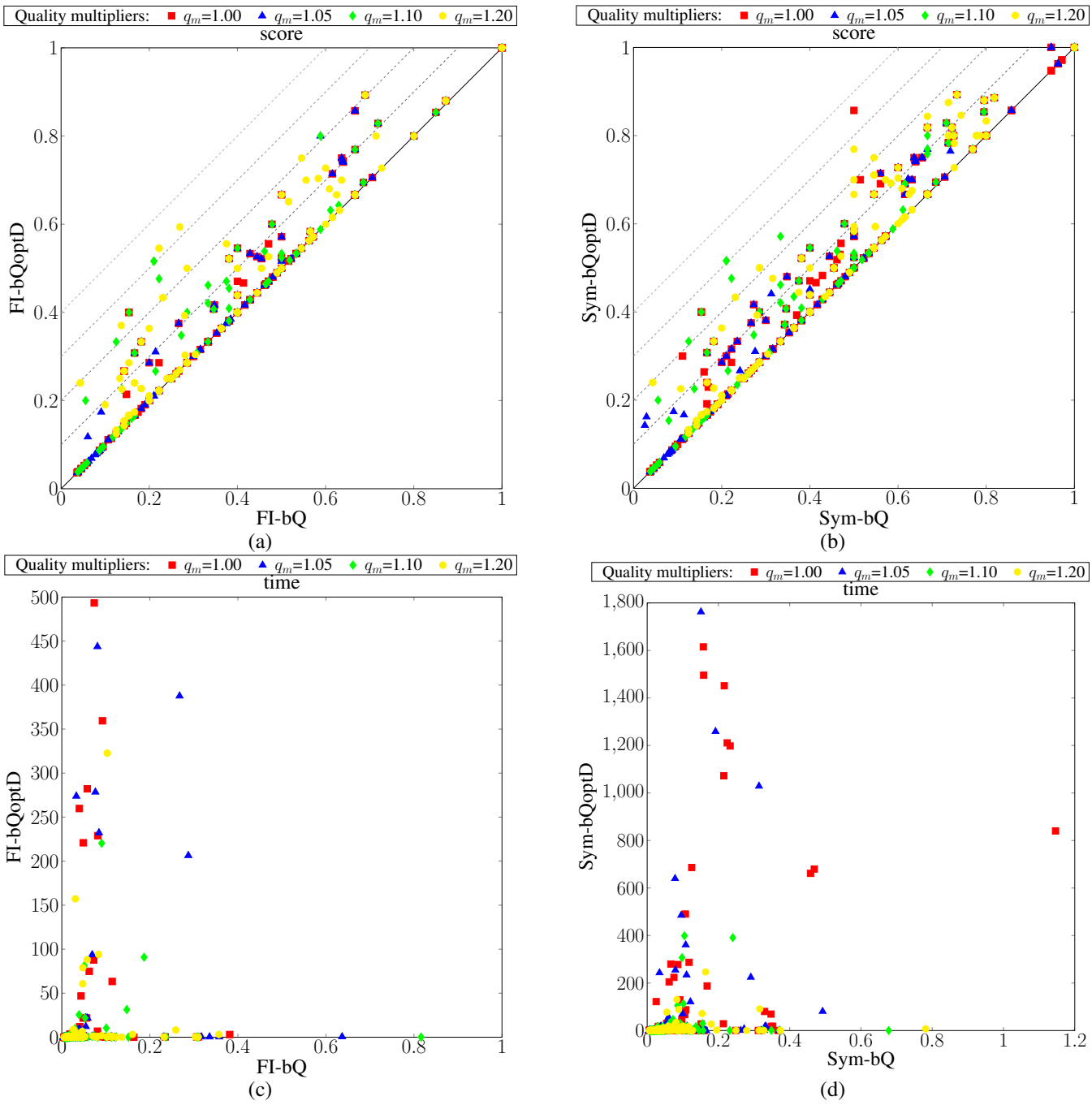


Figure 1: Comparison of the greedy and the optimal approaches to subset selection for $k = 5$. FI-bQ vs. FI-bQoptD: (a) diversity score and (c) score computation time. Sym-bQ vs. Sym-bQoptD: (b) diversity score and (d) score computation time.

$k = 5$. We use the diversity bound of 0.15 and experiment with four quality bounds, defined by multipliers of the optimal plan cost, from $q_m = 1.0$ (optimal plans only), to $q_m = 1.05$, $q_m = 1.10$, and to $q_m = 1.20$ (up to 120% of the optimal plan cost). The results for these four quality bound multipliers are depicted in the four parts of the table. Each part presents the coverage value for the four planners. A planner gets a coverage of 1 on a planning task if it was able to either find a solution of size k or prove that no such solution exists. Otherwise, the planner gets coverage 0. The coverage of a domain is a sum over coverages of all tasks in the domain. Best results are highlighted in bold. While both FI-bQbD and Sym-bQbD outperform the existing approaches in terms of overall coverage, it is worth

mentioning that for each of the approaches there are multiple domains where that approach exhibit superior behavior. To show how these planners scale with larger values of k, Table 2 presents aggregated overall coverage for the values of $k = 10, 100, 1000$. Going deeper into the coverage results, note that DFAM does not prove unsolvability. A*AM, on the other hand, for large values of $k = 100$ and $k = 1000$, did not find any solutions, and all the instances reported in the table for A*AM and these values of k correspond to unsolvable cases. For our suggested approach, both FI-bQbD and Sym-bQbD are able to cope with both, for any value of k. It is worth mentioning that the performance of DFAM often improves, sometimes significantly, with larger quality bounds. We conjecture that this increase is due to the nature of the branch-and-bound algorithm, that does not necessarily produce plans in the order of their costs.

Switching to bQ-k and bQoptD-k, in order to evaluate the quality of the solution obtained using the greedy algorithm, we compare the diversity metric score of the subset chosen by FI-bQ (respectively, Sym-bQ) to the best possible score, obtained with the FI-bQoptD (respectively, Sym-bQoptD) planner. Figure 1(a,b) depict the comparison for $k = 5$, for all four quality multipliers 1.0, 1.05, 1.1, and 1.2, for tasks where both planners were able to find a solution, and the first step produced at least $k + 1$ plans. Note that the greedy approach works surprisingly well. On these tasks, in most cases the greedy algorithm has reached the optimum (nodes on the diagonal): 80 out of 121, 90 out of 126, 71 out of 105, and 50 out of 105 for FI-bQ and 106 out of 176, 99 out of 162, 67 out of 118, and 63 out of 129 for Sym-bQ (for the four quality multipliers, respectively). When it hasn't reached the optimum, the scores are still mostly below the $y = x + 0.1$ line. There are only 21, 18, 17, and 26 tasks for FI-bQ and 26, 23, 20, and 30 tasks for Sym-bQ for the four quality multipliers 1.0, 1.05, 1.1, and 1.2, respectively above the $y = x + 0.1$ line, and only 21 tasks for FI-bQ and 16 tasks for Sym-bQ in total for all quality multipliers above the $y = x + 0.2$ line.

While our experiments show that the greedy approach often produces solutions of diversity close to optimum, the question remains how these algorithms compare in their run time. Figure 1 (c) and (d) present such run time comparison between the greedy and the optimal approaches. The greedy algorithm always finished in under 1.2 seconds, while solving mixed integer linear program takes significantly longer on these tasks, up to 500 seconds for FI-bQoptD and 1760 seconds for Sym-bQoptD.

Finally, note that an inherent limitation of our approach to solving bQoptD-k is that the first step must produce a solution to the (unordered) top quality planning problem. There is no such limitation when solving bQ-k. As a result, FI-bQ successfully solves bQ-k in 617, 617, 615, and 613 tasks for the four quality multipliers, while FI-bQoptD solves bQoptD-k in only 369, 333, 273, and 191 tasks. Similarly, Sym-bQ successfully solves bQ-k in 702, 675, 648, and 624 tasks for the four quality multipliers, while Sym-bQoptD solves bQoptD-k in only 379, 338, 262, and 196 tasks.

Discussion and Future Work

In this work, we extend the portfolio of existing tools for various computational problems in diverse planning by introducing three new such tools. We follow the recently introduced taxonomy and, focusing on bQbD-k, map existing planners DFAA/DFAM and A*AA/A*AM to that problem. For that, we introduce a novel quality metric under which these planners can be considered to solve bQbD-k. The metric also allows us to use top quality planners as a basis for our proposed planners, for bQbD-k as well as for other computational problems, choosing a subset of plans from the solution for the top quality problem. We show that it is NP-complete to find a solution to bQbD-k, given the set of all plans of bounded cost and suggest using a previously proposed integer linear programming based approach, which is experimentally shown to favorably compete with existing planners. As the integer linear program was not previously detailed in the literature, we present it in detail and formally prove that it can be used for solving bQbD-k. Switching from bounding to optimizing diversity, we suggest a novel mixed integer linear program and formally prove that this program solves bQoptD-k. For another computational problem, bQ-k, we use an existing greedy approach of selecting a subset of plans, and empirically show that such a simple approach is able to often achieve the optimum in practice.

Our suggested approach is similar to the one of Vadlamudi and Kambhampati (2016), in that it is also separated into two steps: (i) finding a set of plans of bounded cost, and (ii) choosing a proper subset from the found set. There are two major differences. The first one is the stopping criteria for step (i): while Vadlamudi and Kambhampati (2016) iterate until enough plans are found or no more plans exist, and can stop before finding all plans of bounded cost, we are using an existing (unordered) top-quality planner as is, and therefore will produce the set of all plans of bounded cost in step (i). While it is possible to adapt the top quality planner that we used to terminate earlier, we decided not to do so, to allow for easily replacing the top quality planner with a different one. The second major difference is that instead of trying to construct a feasible solution during the execution of step (i), we perform step (ii) after the first step is finished, as a post-processing. Further, instead of implementing a dedicated algorithm, we cast the problem of choosing the proper subset as an integer linear program, allowing us to use existing solvers. Thus, our solution is highly modular, allowing us to easily replace the solvers when better ones become available.

While there has been significant progress in the field of diverse planning recently, there are still several interesting computational problems for which no planners currently exist. For example, in this work we show how to optimize diversity when the set of candidate plans is given. However, if the quality restriction is alleviated, it is not clear how to choose a set of maximal diversity. It is not even clear whether all plans must be considered while searching for such a set. Another possible problem of interest is finding a subset of optimal quality among the bounded diversity ones. Focusing on these planning problems is an interesting research direction.

References

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

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of action generation for cyber security using classical planning. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 12–21. AAAI Press.

Bryce, D. 2014. Landmark-based plan distance measures for diverse planning. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 56–64. AAAI Press.

Coman, A., and Muñoz-Avila, H. 2011. Generating diverse plans using quantitative and qualitative plan distance metrics. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 946–951. AAAI Press.

Flerova, N.; Marinescu, R.; and Dechter, R. 2016. Searching for the m best solutions in graphical models. *jair* 55:889–952.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 212–221. AAAI Press.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman.

Goldman, R. P., and Kuter, U. 2015. Measuring plan diversity: Pathologies in existing approaches and a new plan distance metric. In Bonet, B., and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3275–3282. AAAI Press.

Katz, M., and Sohrabi, S. 2019. Diversity score computation for diverse planning. https://doi.org/10.5281/zenodo.2691996.

Katz, M., and Sohrabi, S. 2020. Reshaping diverse planning. In Conitzer, V., and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018a. Delfi: Online planner selection for cost-optimal planning. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 57–64.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018b. A novel iterative approach to top-k planning. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press.

Katz, M.; Sohrabi, S.; and Udrea, O. 2020. Top-quality planning: Finding practically useful sets of best plans. In Conitzer, V., and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press.

Mohr, F.; Wever, M.; and Hüllermeier, E. 2018. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning* 107(8):1495–1515.

Myers, K. L., and Lee, T. J. 1999. Generating qualitatively different plans through metatheoretic biases. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, 570–576. AAAI Press.

Nguyen, T. A.; Do, M. B.; Gerevini, A.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. *Artificial Intelligence* 190:1–31.

Riabov, A. V.; Sohrabi, S.; Sow, D. M.; Turaga, D. S.; Udrea, O.; and Vu, L. H. 2015. Planning-based reasoning for automated large-scale data analysis. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 282–290. AAAI Press.

Riabov, A. V.; Sohrabi, S.; and Udrea, O. 2014. New algorithms for the top-k planning problem. In *ICAPS 2014 Scheduling and Planning Applications woRKshop*, 10–16.

Roberts, M.; Howe, A. E.; and Ray, I. 2014. Evaluating diversity in classical planning. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 253–261. AAAI Press.

Sohrabi, S.; Riabov, A. V.; Katz, M.; and Udrea, O. 2018. An AI planning solution to scenario generation for enterprise risk management. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 160–167. AAAI Press.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3258–3264. AAAI Press.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic top-k planning. In Conitzer, V., and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press.

Vadlamudi, S. G., and Kambhampati, S. 2016. A combinatorial search perspective on diverse solution generation. In Schuurmans, D., and Wellman, M., eds., *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, 776–783. AAAI Press.

Automatic Configuration of Benchmark Sets for Classical Planning

Álvaro Torralba,[1] Jendrik Seipp,[2] Silvan Sievers[2]
[1]Aalborg University, Denmark
[2]University of Basel, Switzerland
alto@cs.aau.dk, jendrik.seipp@unibas.ch, silvan.sievers@unibas.ch

Abstract

The benchmarks from previous International Planning Competitions are commonly used to evaluate new planning algorithms. Since this set has grown organically over the years, it has several flaws: it contains duplicate tasks, unsolvable tasks, trivially solvable domains, and domains with modelling errors. Also, diverse domain sizes complicate aggregating results. Most importantly, however, the range of task difficulty is very small in many domains. We propose an automated method for creating benchmarks that solves these issues. To find a good scaling in difficulty, we automatically configure the parameters of benchmark domains. We show that the resulting benchmark set improves empirical comparisons by allowing to differentiate between planners more easily.

1 Introduction

Domain-independent planning aims to develop general solvers that find solutions to arbitrary sequential decision-making problems. This makes the evaluation of planners an essential part of planning research. The International Planning Competition (IPC) has set some evaluation standards and triggered the development of tools that compare planners in terms of different metrics (Linares López, Celorrio, and Helmert 2013; Seipp et al. 2017; Vallati, Chrpa, and McCluskey 2018). The most popular metric is coverage, i.e., the number of solved benchmark instances within certain time and memory limits. Typically, there are two main goals for the evaluation: (1) analyze the impact of the novel algorithms by comparing their performance against a baseline, and (2) compare the performance against the state of the art to evaluate the progress in the area.

Evaluating planners on different benchmark sets may produce different results, leading to different conclusions from the evaluation. Not only is it important which domains we choose, but also how we model the domains (Riddle, Holte, and Barley 2011), and which instances of a domain we select. Therefore, having a standardized benchmark set is important to increase the comparability of results across different papers, and to avoid the use of benchmarks tailored for the proposed technique.

We focus on classical planning where the current standard benchmark set has grown across the nine editions of the IPC so far, from 1998 to 2018 (e.g., Hoffmann and Edelkamp 2005; Linares López, Celorrio, and Olaya 2015). Numerous researchers have contributed to this set by carefully designing new domains (e.g., Hoffmann et al. 2006), so it features a diverse set of domains that pose interesting challenges for planning algorithms. However, there are several issues with this benchmark set (Moraru and Edelkamp 2019). For example, it uses a different number of instances per domain, which reduces the value of statistics aggregated over different domains. Moreover, instances in the current benchmark set were scaled to be useful for the evaluation of planners at the respective IPC. Some of the domains are trivially solved by modern planners, making it impossible to show any coverage advantages over a baseline. On the other hand, early IPC editions did not have a specialized track for optimal planning, and some of their instances are too hard even for state-of-the-art optimal planners.

This paper deals with the question of how to generate instances of a domain to evaluate planning algorithms. Our goal is to improve the empirical evaluation of future planning papers by (1) providing an algorithm for automatically constructing interesting benchmark sets and by (2) using this algorithm to construct a new benchmark set where differences in performance are better reflected in coverage than under the current standard. We aim to generate a set of instances that range from very easy (solved by most planners) to very hard (out of reach for current state-of-the-art planners) allowing future approaches to show benefits with respect to the harder instances. This definition necessarily depends on the algorithms being evaluated.

We identify which properties are desirable for a benchmark set and propose an automatic method that generates a set of instances, given an *instance generator*, a *baseline* planner that represents the expected minimum performance of any planner, and a set of *state-of-the-art* planners. The instance sets generated by our method fulfill the desirable properties by design. To avoid overfitting to the sets of planners used and not introduce a bias in our benchmark set, our method does not select a set of instances directly, but rather performs a search on the space of possible parameters for the generator to obtain a set of instances of adequate difficulty. We use our tool to design two separate sets of benchmarks, for optimal and satisficing planning, and show their advantages over the current standard IPC benchmark set.

2 Background

Informally, a classical *planning task* is defined by an initial state, a set of actions and a goal description. Given a planning task, a *planner* finds a *plan*, that is, a sequence of actions that can be applied in the initial state to achieve the goal. A plan is optimal if it minimizes the summed-up cost of the actions among all plans. If the planner is guaranteed to find an optimal solution, it is an optimal planner, otherwise it is a satisficing planner. In both settings, we only consider solvable planning tasks.

Since its inception in 1998, the International Planning Competition (IPC) has set the standards for the evaluation of planners such as the planner input language PDDL (McDermott et al. 1998). The IPC also introduced numerous planning tasks from different problem settings, called *domains*.

A planning task is typically divided into a domain and an instance file. The domain file defines the types of objects, their properties, and the action schemas. Each instance file can have a different number of objects, initial state and goals. Most domains have an instance generator, a program that, given certain parameters and a random seed, will generate a new instance of the domain. Even though many instance generators are available,[1] most planning papers use the benchmarks introduced for the IPCs, since a standardized benchmark set makes research more reproducible.

As an example, consider the Nomystery domain, where a truck must deliver a set of packages to certain locations. To do that, there is a limited amount of fuel that is consumed by drive actions. Instances differ in the amount of fuel available, the number of locations and their connections, the number of packages, and their initial and final location. The instance generator for Nomystery accepts several parameters that allow the benchmark designer to control the difficulty of the generated instances: the number of locations, the number of packages, the number of edges between locations, the maximum fuel consumption between two locations, and the constrainedness $C \geq 1$, so that the amount of fuel in the initial state is set to C times the minimum fuel consumption required to solve the instance.

3 Benchmark Design Principles

The purpose of a benchmark set is to evaluate planners and compare their performance on a diverse class of problems. Ideally, one should select a diverse set of domains that are representative of real-world scenarios where different users apply planning to solve their problems. However, in addition to selecting interesting domains, one must select a set of concrete instances from each domain to evaluate the planners on. This selection of instances is an important step in the design of the benchmark set, since different instance sets of a domain may lead to different conclusions about which planner is better at solving instances of a given domain. Our goal is that, for any two planners A and B (possibly unknown at the time when the instance set is generated) if A is consistently faster than B on the instances of a domain, the probability that this is reflected on the coverage score should be as high as possible.

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

	IPC			New'14		
	L	D	O	L	D	O
Nomystery	11	**20**	12	25	**30**	24
Rovers	**40**	**40**	**40**	**22**	18	21
Woodworking	**50**	**50**	**50**	18	27	**30**
Total	101	**110**	102	65	**75**	**75**

Table 1: Coverage of LAMA (L), and two IPC 2018 agile planners Decstar (D) and OLCFF (O) on three domains.

For aggregated statistics to be meaningful, not only should all domains have the same number of instances, but their difficulty should also scale similarly. Otherwise, conclusions taken from the empirical evaluation may be biased. Table 1 shows an example comparing 3 planners in 3 domains when using the IPC instances and our New'14 benchmark set, as described in the evaluation section. A paper evaluating these planners with IPC instances would reach the conclusion that Decstar is clearly superior to the other two planners in these domains, both in total coverage and on a per-domain basis since it has better or equal coverage in all domains. However, this conclusion is biased because instances are not well scaled. Instances in Rovers and Woodworking are way too easy and therefore they do not show any differences between the planners. Using our New'14 instances leads to a different conclusion: all three planners are complementary. Of course, no strong conclusions can be taken out of only 3 domains. However, using more domains will help to alleviate this issue only if the instances are well scaled.

A good scaling must meet three conditions: (1) have easy instances that are solved by all planners, (2) have hard instances that are not solved by any current planner, and (3) the instance difficulty should grow smoothly.

Condition (1) is necessary for experiments to be informative at all: if some planners do not solve any instance, no conclusions can be obtained about their relative performance. This happens in some domains of the IPC benchmark set for optimal planning. E.g., Fišer, Torralba, and Shleyfman (2019) write that "In childsnack, [they] measured about twice as many expanded states per second. However, no planner solved any instance in this domain.".

Condition (2) is necessary for new algorithms to show that they can deal with instances that previous planners could not, as shown by our example in Table 1.

Condition (3) is necessary for differences between the planners' performance to be reflected in coverage. To see why, consider an idealized setting where a baseline planner A, whose runtime scales exponentially ($t(A, x) = x^C$ for some constant C), is compared to an improved planner version B which is always faster than A by at least a factor of $K > 1$, i.e., $t(B, x) \leq \frac{t(A,x)}{K}$. Given these assumptions, there is a guaranteed difference in coverage if and only if (1) some instances are solved by B, (2) not all instances are solved by A, and (3) $K \geq C$. Otherwise, there may be cases where both planners solve the same number of instances,

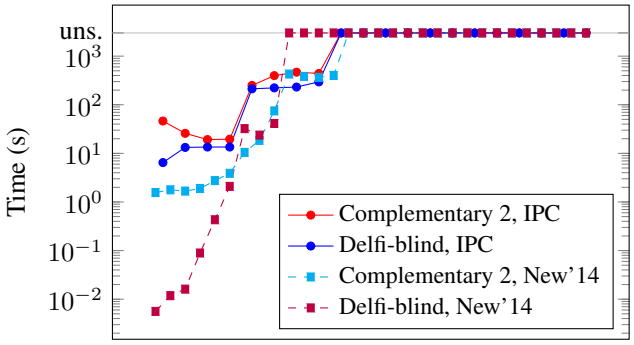


Figure 1: Runtime of two IPC 2018 optimal planners in the Barman domain using the IPC and New'14 instance sets.

and the difference in performance by a factor of K is missed by the coverage analysis. If these conditions do not hold, it is possible to choose runtimes for A and B that are compatible with these exponential scalings, so that their coverage is equal. For example, if $K = 2$ and $C = 3$, then (3) does not hold. So for any time limit (e.g., 300 seconds), if the runtime of the last instance solved by A is close enough to the time limit (e.g., 250 seconds), the next instance cannot be solved by B below the time limit (e.g., $\frac{250 \cdot 3}{2} > 300$).

Real distributions of planner runtimes over sets of instances differ from this idealized example in that they usually involve constant factors, the runtime scaling of different planners may be completely different, and even for a single planner it may be impossible to obtain instances that scale according to the desired runtimes in some domains. But ideally, all domains should consist of a collection of instances of increasing difficulty, ranging from very easy to very hard for current planners. Therefore, we aim for a collection where the easiest instance is quickly solved by most planners; all domains have instances that are not solved by current planners; and difficulty scales by approximately a factor of 1.5–2 between consecutive instances.

Figure 1 exemplifies why a smooth scaling is important in practice. The plot shows the runtimes of two optimal planners on the Barman domain from IPC 2011 and our New'14 benchmark set. In the IPC instances the difficulty does not grow smoothly. Instead, for each group of four instances the difficulty increases visibly and the runtime of all planners increases by about one order of magnitude. This is undesirable since we cannot observe differences in performance for some planners by inspecting their coverage. In contrast, the difficulty on the new benchmark set grows more smoothly, there are instances of more varied difficulty for all planners, and fewer jumps in their runtime. Accordingly, now we can observe that Complementary 2 is able to solve some instances that are not solved with Delfi-blind in this domain.

Given our definition of an ideal benchmark set as one that meets conditions (1), (2), and (3) described above, the instance selection necessarily depends on the planning algorithms being evaluated. As our objective is to generate a benchmark set to evaluate future planners that do not exist yet, one cannot directly select instances that are useful to compare planner's right now. However, selecting an instance set that scales well for current planners may generalize well for planners that are introduced in the next few years. Indeed, our New'14 instance set featured in our examples of Table 1 and Figure 1 was configured without using any planner after 2014, so it did not use any information regarding the IPC'18 planners mentioned in our examples.

One must be careful not to "overfit" the benchmark set to match the set of selected planners, so that the difficulty scales well for the selected planners but not for future planners. To avoid overfitting, we impose two restrictions on the benchmark configuration process. On the one hand, the optimization process does not consider concrete instances, but rather only decides which overall characteristics they should have (e.g. the number of objects in each instance). The final benchmark set is then generated with a random seed different from the one used during our optimization process. On the other hand, we do not consider the individual results of all planners available for the optimization. In each domain, we require a baseline planner that represents the expected minimum performance of any planner to ensure that some instances are solved by all planners. Also, to ensure that some instances remain unsolved, we estimate the performance of state-of-the-art planners by taking the minimum runtime of any of the available planners on each instance. This makes our instance selection as objective as possible since it does not depend on the concrete set of planners available to the benchmark designer, as long as the best planner for the given domain is considered.

4 Configuration of Planning Benchmarks

We consider domains that have an instance generator with several parameters to control the hardness of the generated instances.

4.1 Framework

We model the problem of generating the instances of a benchmark set as follows. Our tool takes as input a tuple $(spec, G, \mathcal{A}, \mathcal{B})$, where $spec$ is a domain specification; G is an instance generator; $\mathcal{A}$ is a set of state-of-the-art planners; and $\mathcal{B}$ is a set of baseline planners. The output will be a set of instances of that domain.

The domain specification describes the instance generator parameters and their constraints and it is discussed in detail in the next section. The instance generator G is a function that takes as input a tuple of parameter values $\rho = \langle \rho_1, \ldots, \rho_k \rangle$ and a random seed $seed \in \mathbb{N}^+$ and outputs a planning task. Let $p \in \mathcal{A} \cup \mathcal{B}$ be a planner, and Π a planning task produced by $G(\rho, seed)$ for some $seed \in \mathbb{N}^+$. We define $t(p, \Pi)$ as the runtime of planner p to solve task Π. For every instance we characterize the performance of a set of planners $\mathcal{A}$ on an instance Π as the minimum runtime of any planner, $t(\mathcal{A}, \Pi) = \min_{p \in \mathcal{A}} t(p, \Pi)$.

Given a parameter configuration ρ, we define $t(p, \rho)$ as the average runtime over all possible tasks that the generator may output for different random seeds. We estimate $t(p, \rho)$ by sampling k tasks $\Pi_1, \ldots, \Pi_k$ from the distribution and taking the average running time $\frac{\sum_{i \in [1,k]} t(p, \Pi_i)}{k}$. In practice,

a small k is sufficient for most domains. In our experiments we used $k = 1$, which was enough to produce robust results.

Our goal is to select a set of parameter configurations $\rho^1, \ldots, \rho^n$ such that the running time $t(\mathcal{A}, \rho^i)$ scales smoothly with i, as described in our general design principles. Note that our automatic tool is not allowed to hand-pick the random seed, but rather the final benchmark set is created by sampling these distributions of instances with new random seeds. This helps to avoid overfitting. An assumption is that the variance of the runtimes $t(\mathcal{A}, \Pi_i)$ for tasks generated with the same parameter configuration is not too high, since otherwise the parameters provided to the generator are irrelevant for obtaining a smooth difficulty scaling. This is a reasonable assumption in practice, following analyses made in the context of predicting planner runtimes (de la Rosa, Cenamor, and Fernández 2017). Out of 10 domains analyzed by de la Rosa, Cenamor, and Fernández, most of them had a very low variance. The one with highest variance was Barman, where 90% of the instances were still relatively close to the average runtime, especially for our purposes.

4.2 Domain Specification

In order to use our benchmark configuration tool, the benchmark designer must specify how to call the instance generator, what parameters are available, as well as which values are appropiate for each parameter.

We distinguish between two types of parameters. *Linear parameters* can be assigned arbitrary non-negative numeric values, where larger values usually result in harder instances. They are typically used to specify the number of objects of a given type. Each generator should have at least one linear parameter that helps to control the difficulty of the generated instances. In contrast, *enumerated parameters* have a finite set of values, and we do not make any assumption about their impact on instance hardness. All other parameters are fixed to a predefined constant value.

We define the instances of a domain as a set of *sequences* of instances. A sequence consists of a list of planning tasks $\Pi_1, \Pi_2, \ldots$ of increasing difficulty. To ensure that difficulty increases, all instances in the sequence have a fixed value for all enumerated parameters, whereas the value of linear parameters increases linearly across the sequence. We specify this via the base value b that the linear parameter takes for Π_1 and the slope m. For example, suppose that a domain has two linear parameters that define the number of packages ($b = 2, m = 1$), and trucks ($b = 1, m = 0.5$). Then, the sequence will generate instances with the following numbers of packages and trucks: $(2, 1), (3, 1), (4, 2), (5, 2), (6, 3)$, etc.

Considering sequences of instances allows us to choose the parameters that generate instances which current planners fail to solve within reasonable time. A limitation of this approach is that not all combinations of parameters are possible. In the example above, a single sequence cannot contain both $(3, 2)$ and $(2, 3)$ because that would require to decrease one of the parameters, which is not allowed by our linear scaling. In most cases, this is not a problem because we can use multiple sequences of instances for a single domain. A notable exception are parameters that define the width and height of a grid, because they have a strong interaction, i.e., the number of cells is the product of both parameters. In that case, we consider them a single parameter so that the number of tiles in the grid scales linearly.

```
generator_command = "nomystery -l {locations}
  -p {packages} -n {edgefactor} -m {edgeweight}
  -c {constrainedness} -s {seed} -e 0"
domain_attributes = [
  LinearAttr("locations", lower_b=3, upper_b=5,
                          lower_m=0.1, upper_m=1),
  LinearAttr("packages", lower_b=2, upper_b=10),
  ConstantAttr("edgefactor", "1.5"),
  ConstantAttr("edgeweight", "25"),
  EnumAttr("constrainedness", [1.1, 1.5, 2.0])]
```

Figure 2: Example of a domain specification with the generator command and the specification of the corresponding parameters.

The snippet from Figure 2 shows the domain specification for the nomystery domain. For each linear parameter, lower and upper bounds for the base and slope values should be provided. This allows the domain modeller to specify preferences on which parameters to scale (e.g., by restricting the slope m for the number of locations to be between 0.1 and 1, they indicate their preference to increase difficulty by scaling the number of packages). Note that this is important, since a property of a good benchmark set is that instances reflect problems that are "interesting in practice", and this is a subjective matter that the configuration tool cannot decide on its own. If the benchmark designer has no such preference, all parameters can be left with a default interval.

Often, instance generators impose constraints on the range of parameter values or their combination. Those constraints must be enforced by adding a postprocessing function that updates the value of the parameters passed to the generator. This is an arbitrary function provided by the benchmark designer which receives the parameters that were automatically chosen and outputs the final parameters that will be provided to the generator. For example, if the number of packages has to be greater than the number of locations, instead of directly selecting the number of packages, our linear scaling will consider the number of locations and the number of additional packages. All of these adjustments must be done on a per-domain basis, since they depend on the specific characteristics of the domain and generator.

Given this framework, our automatic tool decides which sequences of instances are suitable for each domain. This is done in two phases: the first phase designs a set of candidate sequences (Sequence Optimization), and the second phase performs a final selection that adheres to our design principles as much as possible (Sequence Selection).

4.3 Sequence Optimization

The first phase generates sequences of 30 instances by optimizing sequence parameters. To guide the search towards sequences that scale the instance difficulty as smoothly as possible, we compute a penalty score for each sequence and

search for the sequence that minimizes this score.

Sequences are evaluated by running hand-picked state-of-the-art ($\mathcal{A}$) and baseline ($\mathcal{B}$) planners on their instances, using a time limit of 180 seconds per instance. We ignore instances that are solved under 10 seconds, considering that differences of ± 5 seconds are not meaningful enough. Since the sequences are generated with increasing values of the linear parameters, we assume that the runtimes will always increase, so we can stop our evaluation as soon as one instance is not solved under the time limit. In cases where this does not hold, we enforce it by sorting the runtimes of the instances. Our assumption is that these anomalies stem from using different random seeds for the instance generator and the results will be different with different random seeds.[2] The runtime of a set of planners is the minimum of the runtimes of the individual planners. For evaluating a sequence we consider the first five instances with a minimum runtime above 10 seconds. We ignore harder instances because they will usually incur runtimes above the 180 seconds time limit. Let $t(X,1),\ldots,t(X,5)$ be the runtimes of the set of planners X on the first five instances with a runtime above 10 seconds. The penalty score is defined as $\sum_{i\in[2,5]} S(\mathcal{B},i) + S(\mathcal{A},i)$ where $S(X,i) =$

$$\begin{cases} 3 - \dfrac{2t(X,i)}{t(X,i-1)} & \text{if } 1 \leq \dfrac{t(X,i)}{t(X,i-1)} \leq 1.5 \\ 0 & \text{if } 1.5 < \dfrac{t(X,i)}{t(X,i-1)} \leq 2 \\ 1 - \dfrac{2t(X,i-1)}{t(X,i)} & \text{if } 2t(X,i-1) \leq t(X,i) \leq 180 \\ 2 & \text{if } t(X,i) > 180 \end{cases}$$

This penalty is lower for sequences whose runtime scales smoothly, assigning a minimum score of 0 to any sequence where the runtimes of both the baseline and state-of-the-art planners scale exponentially with a factor between 1.5 and 2, e.g., $\langle 10, 15, 23, 35, 52, \ldots \rangle$, or $\langle 10, 20, 40, 80, 160, \ldots \rangle$. If not enough instances are solved in the $[10, 180]$ second interval, the sequence gets a penalty of 2, and otherwise we assign it a penalty between 0 and 1. To avoid generating sequences where all instances are solved by the state-of-the-art planners, we also add a penalty of 1 for each instance solved by them beyond 20 instances. To guarantee that all valid sequences contain some instances solvable within the time limit and to speed up the evaluation we require the first three instances to be solved within 10, 60, and 180 seconds, respectively. Otherwise, we discard the sequence, unless all linear parameters are at their minimum value.

The concrete choice of penalty values is arbitrary. What matters is that sequences that minimize this score adhere more to the design principles introduced in Section 3 than those that do not, thereby guiding the parameter optimization towards good sequences.

[2]Note that any parameter that has an unpredictable influence on the runtime of a planner should be considered an enumerated parameter and remain constant for a given sequence.

4.4 Sequence Selection

After performing one or more optimization runs for a domain (using different random seeds) as described above, we collect all sequences seen during the optimization process. Since this set can be very large, we only keep the 100 sequences with the lowest penalty score per value of the enumerated parameters. For each group of sequences where the planners solves the same instances, we only keep one member of the group. This filtering ensures that we keep a set of diverse sequences with a good penalty score.

For each sequence, we collect the runtimes of all instances solvable in 180 seconds from the sequence optimization phase. For the rest of the instances, we estimate the runtime by assuming that runtimes will increase according to the average increasing factor $t(\mathcal{A},i)/t(\mathcal{A},i-1)$ observed on the instances solved between 5 and 180 seconds. This is a very rough estimate but it is accurate enough for the purposes of choosing up to when a sequence should be continued (see below).

We model the problem of selecting a suitable set of subsequences as a Mixed-Integer Programming (MIP) problem, where constraints directly aim to model the design principles of Section 3. The decision variables model the start and end points of each sub-sequence of instances. The selection must satisfy the following *hard constraints* that model properties desirable for a good set of instances:

(H1) The number of selected instances must be exactly 30.

(H2) There must be at least one instance solvable by the baseline under 30 seconds.

(H3) All sequences must start with an instance that is solvable by a state-of-the-art planner and end with an instance whose estimated runtime is higher than 2000 seconds.

(H4) Each parameter configuration must be used (with different random seeds) at most twice, and only once for domains whose generators do not admit a random seed.

The objective is to minimize the summed-up penalty score of all sequences used, plus the penalty incurred for violating any of the following *soft constraints*:

(S1) The number of instances solved by the baseline under 30 seconds must be between 2 and 6 (with a penalty of $2x^2$ where x is the deviation with respect to the constraint).

(S2) The number of instances solved under 180 seconds must be between 8 and 15 (with a penalty of $2x^2$ where x is the deviation with respect to the constraint).

(S3) All sequences must end with an instance whose estimated runtime is between 18 000 and 180 000 seconds (that is, 1–2 orders of magnitude larger than the typical time limit of 30 minutes). Larger times t incur a penalty of $100t/180000$ and smaller times incur a penalty of $100(18000/t)$.

(S4) If a parameter configuration is used more than once, there is a penalty of 100.

	Optimal	Satisficing
Configuration New'14	blind search (baseline), all four components of the FDSS 1 portfolio from IPC 2011 (Helmert et al. 2011) and SymBA$^*_1$ from IPC 2014 (Torralba et al. 2014)	greedy best-first search with FF heuristic (baseline, Hoffmann and Nebel 2001), LAMA (Richter and Westphal 2010) and Madagascar (Rintanen 2012)
Configuration New'20	union of *Configuration New'14* and *Evaluation*	union of *Configuration New'14* and *Evaluation*
Evaluation	five components of Delfi1 portfolio from IPC 2018 using symmetry pruning and partial order reduction (blind search, iPDB, LM-Cut and two M&S variants, see Katz et al. 2018) and three vanilla IPC 2018 planners: Complementary2 (Franco, Lelis, and Barley 2018), DecStar (Gnad, Shleyfman, and Hoffmann 2018), Scorpion (Seipp 2018b)	eight vanilla IPC 2018 planners: Cerberus (Katz 2018), BFWS-PREF, DUAL-BFWS and POLY-BFWS (Francès et al. 2018), DecStar (Gnad, Shleyfman, and Hoffmann 2018), OLCFF (Fickert and Hoffmann 2018), Fast Downward Remix (Seipp 2018a) and Saarplan (Fickert et al. 2018)

Table 2: Choice of planners for benchmark generation and evaluation.

For domains where all instances in the sequence are solved by state of the art planners under 180 seconds (because the domain is solvable in polynomial time and it is impossible to fulfill our criteria with state-of-the-art planners), we consider the runtimes of the baseline instead of those of the state of the art planners in our constraints described above.

Constraints (H2), (S1) and (S2) ensure that the instance set contains some easy instances, so that any future planning algorithms are expected to solve at least some instances, allowing researchers to analyze the behaviour of their algorithms in the domain. Constraints (H3) and (S3) ensure that, whenever possible, at least some of the instances are expected to be out of reach for state-of-the-art planners. Together with minimizing the penalty score of the selected sequences, they aim to obtain a smooth scaling, since sequences must interpolate between easy and hard instances and sequences with smoother scaling are preferred. Finally, constraints (H4) and (S4) are needed to avoid duplicate instances and instances that are very similar to each other.

The penalties are set arbitrarily, but they scale quadratically with respect to the deviation because it is better to not fulfill several soft constraints entirely than to completely ignore one of the constraints.

5 Experiments

We implemented the first phase, i.e., sequence optimization, using the automatic configurator SMAC (Hutter, Hoos, and Leyton-Brown 2011). We test our approach by running two completely separated optimizations for optimal and satisificing planners. As baseline planners, we use blind search for optimal planning and greedy best-first search with the FF heuristic (Hoffmann and Nebel 2001) for satisficing planning, both implemented in Fast Downward (Helmert 2006a).

Both for optimal and satisficing planning, we generate two separate benchmark sets, New'14 and New'20, that differ in the set of state-of-the-art planners available for optimization. The New'14 set consists of a heterogeneous set of planning algorithms from IPCs 2011 and 2014. The New'20 set is trained using the same planners plus the ones used in the evaluation. Therefore, for New'14, the configuration and evaluation sets are disjoint, while for New'20, the sets overlap. Table 2 gives an overview of the planners used.

Since we limit each planner run during the optimization to 3 minutes, we adapt the planners for the configuration phase by breaking portfolios into components and by adapting preprocessing time limits. For optimization in each domain, we hand-pick 1–3 planners that perform best in that domain, which is sufficient to approximate the minimum time of any planner in each instance. We run SMAC 10 times using different random seeds. Each run is limited to 10 hours. After the first phase finishes, we consider all sequences encountered during optimization for the second phase, i.e., sequence selection. We filter the instances as described in Section 4.4 and solve the MIP for sequence selection using CPLEX 12.10, which finishes in under 30 seconds for each domain.

We evaluate the new benchmark sets using the aforementioned planners, limiting each run to 30 minutes and 3.5 GiB. In Table 3 we compare the IPC benchmarks to the new benchmark sets for optimal and satisficing planning. We compare benchmark sets according to two metrics: the range of coverage scores per domain, which allows us to see how many instances are solved by all planners and how many remain unsolved by any of the planners; and the number of pairwise comparisons in which a planner had higher coverage than another, which quantifies how many differences in the performance of planners are reflected by the coverage score.

In optimal planning the difference between the benchmark sets is rather subtle because difficulty typically scales very fast with increasing instance size. Therefore, the IPC set has some interesting instances in all domains. Also, it can be very hard to generate instance sequences whose difficulty scales smoothly, since often increasing one of the parameters of a generator by a unit has a big impact on runtime.

The results are more pronounced for satisficing planners, where the IPC set scales very poorly for some domains. Only in the Elevators domain the IPC set is superior in terms of comparisons detected by the coverage score compared to both new benchmark sets. In contrast, with the new benchmark sets, we observe differences in performance in domains like Blocksworld, Driverlog or Zenotravel, where all planners solve all instances in the IPC set. Overall, New'14 uncovers more differences in coverage between pairs of planners than the IPC set in 21 out of 26 domains, while

		coverage range			comparisons		
Optimal	#IPC	IPC	'14	'20	IPC	'14	'20
barman	34	4–11	9–13	9–12	12	**21**	19
blocksworld	35	18–30	5–12	5–12	18	**24**	**24**
childsnack	20	0–6	9–20	6–21	12	18	**22**
data-network	20	6–14	5–12	5–16	**27**	25	**27**
depot	22	5–14	9–25	8–16	**26**	**26**	24
driverlog	20	7–15	6–30	5–18	22	**26**	25
elevators	50	28–44	7–14	10–18	**26**	**26**	23
floortile	40	16–34	9–18	8–17	21	21	**22**
grid	5	1–3	6–26	4–21	19	**28**	27
gripper	20	8–20	11–30	11–30	**7**	**7**	**7**
hiking	20	12–18	7–9	5–16	23	15	**25**
logistics	63	13–34	5–17	5–14	**27**	**27**	25
miconic	150	56–142	4–28	3–30	25	27	**28**
nomystery	20	8–20	3–27	5–21	18	**28**	27
openstacks	130	42–71	4–11	3–7	**24**	18	7
parking	40	0–15	11–18	12–21	**28**	24	23
rovers	40	6–13	4–26	6–19	**25**	22	7
satellite	36	7–14	8–30	4–27	22	25	**26**
scanalyzer	50	21–33	6–16	7–15	**27**	24	24
snake	20	7–14	5–20	7–19	22	**24**	21
storage	30	15–18	9–25	2–19	21	**27**	26
tpp	30	7–20	7–30	2–7	**24**	**24**	21
transport	70	24–35	5–30	8–19	21	18	**22**
visitall	40	12–30	6–21	5–20	**27**	**27**	**27**
woodworking	50	38–50	16–25	10–14	22	**26**	24
zenotravel	20	8–13	6–30	3–13	23	26	**28**

		coverage range			comparisons		
Satisficing	#IPC	IPC	'14	'20	IPC	'14	'20
barman	40	39–40	7–25	9–30	7	24	**27**
blocksworld	35	35–35	7–24	4–22	0	27	**28**
childsnack	20	1–20	14–30	2–19	27	25	**28**
data-network	20	9–19	10–30	13–30	24	**27**	25
depot	22	21–22	12–20	11–26	7	**27**	22
driverlog	20	20–20	29–30	9–19	0	12	**24**
elevators	50	49–50	30–30	30–30	**7**	0	0
floortile	40	4–40	1–12	1–11	17	**25**	24
grid	5	5–5	4–20	9–21	0	**26**	24
gripper	20	20–20	26–30	26–30	0	**7**	**7**
hiking	20	10–20	2–22	3–26	24	**28**	27
logistics	63	51–63	5–30	5–26	17	**27**	26
miconic	150	150–150	30–30	30–30	**0**	**0**	**0**
nomystery	20	12–20	19–30	2–30	23	18	**26**
openstacks	160	99–160	12–21	14–23	21	**27**	25
parking	40	36–40	14–20	13–16	7	**24**	21
rovers	40	38–40	10–22	6–30	7	26	**27**
satellite	36	26–36	5–30	6–14	**23**	17	**23**
scanalyzer	50	48–50	9–16	13–14	12	**21**	12
snake	20	3–17	6–30	5–14	27	**28**	26
storage	30	21–30	6–26	7–17	26	**27**	26
tpp	30	29–30	10–26	6–21	15	**27**	**27**
transport	70	65–70	22–30	15–23	7	24	**26**
visitall	40	36–40	4–30	4–29	7	24	**26**
woodworking	50	28–50	6–30	5–30	13	**27**	**27**
zenotravel	20	20–20	6–29	5–17	0	23	**25**

Table 3: Comparison of the IPC and new benchmark sets for optimal and satisficing planning. The #IPC column shows the number of tasks per domain in the IPC benchmark set, which is always 30 for the new sets. The coverage range shows the minimum and maximum coverage of any planner. In the "comparisons" columns we list how many pairs of planners yield different coverage scores for each benchmark set.

the opposite is the case in only 4 domains.

The comparison between New'14 and New'20 reveals that our technique is not very sensitive to the set of state-of-the-art planners. The reason is that the state of the art has not advanced enough in the last six years to make a set of instances trained with our method in 2014 outdated.

6 Discussion

Our paper deals with the problem of generating instances that are adequate to evaluate planning algorithms. The goal is to select instances that scale well, so that differences in algorithm performance are reflected in the number of problems solved within a certain time limit. It must be remarked that no benchmark set can replace a careful analysis of the results. Aggregating results from different domains without further analysis may be misleading and an empirical analysis only based on total coverage should be discouraged. Nevertheless, coverage is a useful metric to summarize experimental data and it is used by most planning papers. As shown by our experiments, the coverage metric is more meaningful for the benchmark sets generated with our approach than with the previous standard. Our main result is a new benchmark set, as well as a set of generators and tools that can be used in the future to automatically generate new instances.

In other communities like SAT, there has been a lot of research on how to construct random instances (Selman, Mitchell, and Levesque 1996; Achlioptas et al. 2000; Giráldez-Cru and Levy 2015; Xu et al. 2005) around the phase transition (Cheeseman, Kanefsky, and Taylor 1991). Our approach is orthogonal to any approach that can generate new instances, e.g., around the phase transition of planning problems (Rintanen 2004; Rieffel et al. 2014), or with suitable initial states and goals for Sokoban (Bento, Pereira, and Lelis 2019). Those approaches provide an instance generator that adjusts the instance difficulty for a given problem size, but to generate an instance set still requires to select the value of certain parameters. Our approach is complementary, since it can be used to select suitable values that are useful to evaluate a given set of solvers. Our tool can also be adapted to generate benchmarks with different characteristics, e.g., with smaller instances that are solved in a few seconds (Ruml 2010). Future work could also consider relations among different domains theoretically (Helmert 2003, 2006b) or empirically (Cenamor and Pozanco 2019).

Acknowledgments

We thank Florian Pommerening for helping us set up the experiments and we thank the anonymous reviewers for their helpful comments. We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639). Álvaro Torralba was employed by Saarland University and the CISPA Helmholtz Center for Information Security during part of the development of this paper.

References

Achlioptas, D.; Gomes, C. P.; Kautz, H. A.; and Selman, B. 2000. Generating Satisfiable Problem Instances. In Kautz, H.; and Porter, B., eds., *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, 256–261. AAAI Press.

Bento, D. S.; Pereira, A. G.; and Lelis, L. H. S. 2019. Procedural Generation of Initial States of Sokoban. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 4651–4657. IJCAI.

Cenamor, I.; and Pozanco, A. 2019. Insights from the 2018 IPC Benchmarks. In *ICAPS 2019 Workshop on the International Planning Competition (WIPC)*, 8–14.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the Really Hard Problems Are. In Mylopoulos, J.; and Reiter, R., eds., *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI 1991)*, 331–337. Morgan Kaufmann.

de la Rosa, T.; Cenamor, I.; and Fernández, F. 2017. Performance Modelling of Planners from Homogeneous Problem Sets. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 425–433. AAAI Press.

Fickert, M.; Gnad, D.; Speicher, P.; and Hoffmann, J. 2018. SaarPlan: Combining Saarlands Greatest Planning Techniques. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 11–16.

Fickert, M.; and Hoffmann, J. 2018. OLCFF: Online-Learning h^{CFF}. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 17–19.

Fišer, D.; Torralba, Á.; and Shleyfman, A. 2019. Operator Mutexes and Symmetries for Simplifying Planning Tasks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7586–7593. AAAI Press.

Francès, G.; Geffner, H.; Lipovetzky, N.; and Ramiréz, M. 2018. Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 23–27.

Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The Complementary2 Planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 32–36.

Giráldez-Cru, J.; and Levy, J. 2015. A modularity-based random SAT instances generator. In Yang, Q.; and Wooldridge, M., eds., *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 1952–1958. AAAI Press.

Gnad, D.; Shleyfman, A.; and Hoffmann, J. 2018. DecStar – STAR-topology DECoupled Search at its best. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 42–46.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence* 143(2): 219–262.

Helmert, M. 2006a. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Helmert, M. 2006b. New Complexity Results for Classical Planning Benchmarks. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 52–61. AAAI Press.

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.

Hoffmann, J.; and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research* 24: 519–579.

Hoffmann, J.; Edelkamp, S.; Thiébaux, S.; Englert, R.; dos Santos Liporace, F.; and Trüg, S. 2006. Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4. *Journal of Artificial Intelligence Research* 26: 453–541.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In Coello, C. A. C., ed., *Proceedings of the Fifth Conference on Learning and Intelligent OptimizatioN (LION 2011)*, 507–523. Springer.

Katz, M. 2018. Cerberus: Red-Black Heuristic for Planning Tasks with Conditional Effects Meets Novelty Heuristic and Enchanced Mutex Detection. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 47–51.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 57–64.

Linares López, C.; Celorrio, S. J.; and Helmert, M. 2013. Automating the evaluation of planning systems. *AI Communications* 26(4): 331–354.

Linares López, C.; Celorrio, S. J.; and Olaya, A. G. 2015. The deterministic part of the seventh International Planning Competition. *Artificial Intelligence* 223: 82–119.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version

1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Moraru, I.; and Edelkamp, S. 2019. Benchmarks Old and New: How to compare domain independence for cost-optimal classical planning? In *ICAPS 2019 Workshop on the International Planning Competition (WIPC)*, 36–39.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39: 127–177.

Riddle, P. J.; Holte, R. C.; and Barley, M. W. 2011. Does Representation Matter in the Planning Competition? In *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation (SARA 2011)*. AAAI Press.

Rieffel, E. G.; Venturelli, D.; Do, M.; Hen, I.; and Frank, J. 2014. Parametrized Families of Hard Planning Problems from Phase Transitions. In Brodley, C. E.; and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2337–2343. AAAI Press.

Rintanen, J. 2004. Phase Transitions in Classical Planning: an Experimental Study. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 101–110. AAAI Press.

Rintanen, J. 2012. Planning as Satisfiability: Heuristics. *Artificial Intelligence* 193: 45–86.

Ruml, W. 2010. The Logic of Benchmarking: A Case Against State-of-the-Art Performance. In Felner, A.; and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 142–143. AAAI Press.

Seipp, J. 2018a. Fast Downward Remix. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 74–76.

Seipp, J. 2018b. Fast Downward Scorpion. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 77–79.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461. doi:10.5281/zenodo.790461. URL https://doi.org/10.5281/zenodo.790461.

Selman, B.; Mitchell, D. G.; and Levesque, H. J. 1996. Generating Hard Satisfiability Problems. *Artificial Intelligence* 81(1–2): 17–29.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A Symbolic Bidirectional A* Planner. In *Eighth International Planning Competition (IPC-8): planner abstracts*, 105–109.

Vallati, M.; Chrpa, L.; and McCluskey, T. L. 2018. What you always wanted to know about the deterministic part of the International Planning Competition (IPC) 2014 (but were too afraid to ask). *The Knowledge Engineering Review* 33.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2005. A Simple Model to Generate Hard Satisfiable Instances. In Kaelbling, L. P.; and Saffiotti, A., eds., *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 337–342. Professional Book Center.

Revisiting Dominance Pruning in Decoupled Search

Daniel Gnad
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
gnad@cs.uni-saarland.de

Abstract

In classical planning as heuristic search, duplicate state pruning is a standard method to avoid unnecessarily handling the same state multiple times. In decoupled search, similar to symbolic search approaches, search nodes, called *decoupled states*, do not correspond to individual states, but to entire *sets* of states. As a result, duplicate state pruning cannot be applied in a straightforward manner. Instead, *dominance pruning* is employed, taking into account the state sets. We observe that the time required for dominance checking dominates the overall runtime, and propose two ways of tackling this issue. Our main contribution (1) is a stronger variant of dominance checking for optimal planning, where efficiency and pruning power are most crucial. The new variant greatly improves the latter, without incurring a computational overhead. Furthermore, (2) we develop and implement three methods that make the dominance check more efficient: exact duplicate checking, which, albeit resulting in weaker pruning, can pay off due to the use of hashing; avoiding the dominance check when leaf state spaces are invertible; and exploiting the transitivity of the dominance relation to only check against the relevant subset of visited decoupled states. We show empirically that all our improvements are indeed beneficial across many standard benchmark domains.

Introduction

In classical planning, the most popular approach to solve planning tasks is heuristic search in the explicit state space (Bonet and Geffner 1999). Heuristic search, however, suffers from the state explosion problem that arises from the fact that the size of the state space of a task is exponential in the size of its description. Many methods have been introduced to tackle this explosion, such as partial-order reduction (Valmari 1989; Godefroid and Wolper 1991; Edelkamp, Leue, and Lluch-Lafuente 2004; Alkhazraji et al. 2012; Wehrle et al. 2013; Wehrle and Helmert 2014), symmetry breaking (Starke 1991; Pochter, Zohar, and Rosenschein 2011; Domshlak, Katz, and Shleyfman 2012), dominance pruning (Hall et al. 2013; Torralba and Hoffmann 2015; Torralba 2017), or symbolic representations (Bryant 1986; Edelkamp and Helmert 1999; Torralba et al. 2017). In this work, we look into a recent addition to this set of techniques, namely star-topology decoupled state space search, or decoupled search for short (Gnad and Hoffmann 2018).

Decoupled search is a form of factored planning (Amir and Engelhardt 2003; Brafman and Domshlak 2006, 2008; Fabre et al. 2010) that partitions the variables of a planning task into components such that the causal dependencies between the components form a star topology. The *center* C of this topology can interact arbitrarily with the other components, the *leaves* $\mathcal{L} = \{L_1, \dots, L_n\}$, while any interaction between leaves has to involve the center, too. A *decoupled state* $s^{\mathcal{F}}$ corresponds to a single *center state*, an assignment to C, and a non-empty set of *leaf states* (assignments to an L_i) for each L_i. The *member states* of $s^{\mathcal{F}}$, i. e., the set of explicit states it represents, result from all combinations of leaf states across leaf factors, sharing the same center state. Thereby, a decoupled state represents exponentially many explicit states, leading to a reduction in search effort. Prior work has shown that there exist scalable families of planning tasks where this reduction is exponentially larger for decoupled search than it is for other state-space-reduction methods like partial-order reduction (Gnad, Hoffmann, and Wehrle 2019), symmetry breaking (Gnad et al. 2017), symbolic representations (Gnad and Hoffmann 2018), and Petri-net unfolding (Gnad and Hoffmann 2019).

Since decoupled states $s^{\mathcal{F}}$, correspond to *sets of states*, namely the member states of $s^{\mathcal{F}}$, the standard concept of duplicate elimination, ignoring a state that has already been visited (on a cheaper path) to avoid repeated work, cannot be applied so easily. More importantly, it is not as effective as in explicit state search, because two decoupled states are only equal if the entire sets of member states they represent are equal. Therefore, prior work on decoupled search only considered dominance pruning (Torralba et al. 2016; Gnad and Hoffmann 2018), where a decoupled state $s_1^{\mathcal{F}}$ that contains the member states S_1 is *dominated by* a decoupled state $s_2^{\mathcal{F}}$ that represents the set of states S_2 if $S_1 \subseteq S_2$, respecting the so-called *pricing function* of $s_1^{\mathcal{F}}$ and $s_2^{\mathcal{F}}$ in case of optimal planning. Then, if each member state of $s_2^{\mathcal{F}}$ is reached in $s_1^{\mathcal{F}}$ with at most the price it has in $s_2^{\mathcal{F}}$, we can safely prune $s_1^{\mathcal{F}}$, like duplicate states in explicit state search.

Initiating this work was the observation that on average around 60% of the overall runtime of decoupled search for optimal planning is spent on dominance checking (on the benchmarks from our evaluation that are solved in $\geq 0.1s$). Thus, we take a closer look at (1) algorithmic improvements that lead to an increased pruning power for optimal planning,

and (2) ways to make the dominance check more efficient in general. Regarding (1), we introduce two new extensions to the dominance check. First, we take into account not only the pricing function, but incorporate the g-value of A* in the check. Second, we propose a decoupled-state transformation that moves cost from the pricing function into the g-value. Both make the dominance check more informed, without introducing a computational overhead. For (2), we experiment with an implementation of exact duplicate checking, which, albeit resulting in weaker pruning, can be beneficial runtime-wise due to the use of hashing; we identify cases where leaves can be skipped in the check, namely if their *leaf state space* is invertible; and, exploiting the transitivity of the dominance relation, we only check against the non-dominated subset of visited decoupled states.

In our experimental evaluation, we see that the improvements as of (2) indeed have a (mostly mild) positive impact on the runtime. The stronger pruning variants from (1) lead to a substantial reduction in search effort, which translates to a strong runtime advantage.

Background

We consider a classical planning framework with finite-domain state variables (Bäckström and Nebel 1995; Helmert 2006). In this framework a *planning task* is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, I, G \rangle$, where $\mathcal{V}$ is a finite set of *variables*, each variable $v \in \mathcal{V}$ is associated with a finite domain $\mathcal{D}(v)$. $\mathcal{A}$ is a finite set of *actions*, each $a \in \mathcal{A}$ being a triple $\langle \mathsf{pre}(a), \mathsf{eff}(a), \mathsf{cost}(a) \rangle$ of *precondition*, *effect*, and *cost*. The preconditions $\mathsf{pre}(a)$ and effects $\mathsf{eff}(a)$ are partial assignments to $\mathcal{V}$, and the cost is a non-negative real number $\mathsf{cost}(a) \in \mathbb{R}^{0+}$. A *state* is a complete assignment to $\mathcal{V}$, I is the *initial state*, and the *goal* G is a partial assignment to $\mathcal{V}$. For a partial assignment p, we denote by $vars(p) \subseteq \mathcal{V}$ the subset of variables on which p is defined. For $V' \subseteq \mathcal{V}$, by $p[V']$ we denote the restriction of p onto $V' \cap vars(p)$, i. e., the assignment to V' made by p. We identify (partial) variable assignments with sets of variable/value pairs.

An action a is *applicable* in state s if $\mathsf{pre}(a) \subseteq s$. Applying a in a (partial) state s changes the value of all $v \in vars(\mathsf{eff}(a)) \cap vars(s)$ to $\mathsf{eff}(a)[v]$, and leaves s unchanged elsewhere. The outcome state is denoted $s[\![a]\!]$. A *plan* for Π is an action sequence π iteratively applicable in I, and resulting in a state s_G where $G \subseteq s_G$. A plan π is *optimal* if the summed-up cost of its actions, denoted $\mathsf{cost}(\pi)$, is minimal among all plans for Π.

During an A* search, we denote by $g(s)$ the minimum cost of a path on which a state s was reached from I. Note that the g-value of a state can get reduced during the search, in case a cheaper path from I to s is generated.

Decoupled Search

Decoupled search is a technique developed to avoid the combinatorial explosion of having to enumerate all possible variable assignments of causally independent parts of a planning task. It does so by partitioning the state variables into a *factoring* $\mathcal{F}$, whose elements are called *factors*. By imposing a structural requirement on the interaction between these factors, namely a *star topology*, decoupled search can efficiently handle cross-factor dependencies. A *star factoring* is one that has a *center* $C \in \mathcal{F}$ that interacts arbitrarily with the other factors $L \in \mathcal{L} := \mathcal{F} \setminus C$, called *leaves*, but where the only interaction between leaves is via the center.

Actions affecting C, i. e., with an effect on a variable in C, are called *center actions*, denoted $\mathcal{A}^C$, and those affecting a leaf are called *leaf actions*, denoted $\mathcal{A}^{\mathcal{L}}$. The actions that affect a particular leaf $L \in \mathcal{L}$ are denoted $\mathcal{A}^L$.[1] A sequence of center actions applicable in I in the projection onto C is a *center path*, a sequence of leaf actions affecting L, applicable in I in the projection onto L, is a *leaf path*. A complete assignment to C, respectively an $L \in \mathcal{L}$, is called a *center state*, respectively *leaf state*. The set of all leaf states is denoted $S^{\mathcal{L}}$, and that of a particular leaf L is denoted S^L.

A *decoupled state* $s^{\mathcal{F}}$ is a pair $\langle \mathsf{center}(s^{\mathcal{F}}), \mathsf{prices}(s^{\mathcal{F}}) \rangle$, where $\mathsf{center}(s^{\mathcal{F}})$ is a center state, and $\mathsf{prices}(s^{\mathcal{F}}) : S^{\mathcal{L}} \mapsto \mathbb{R}^{0+} \cup \{\infty\}$ is the *pricing function*, that assigns every leaf state a non-negative price. The pricing function is maintained during decoupled search in a way so that the price of a leaf state s^L is the cost of a cheapest leaf path that ends in s^L and that is *compliant*, i. e., that can be scheduled alongside the center path executed up to $s^{\mathcal{F}}$. By $S^{\mathcal{F}}$ we denote the set of all decoupled states. We say that a decoupled state $s^{\mathcal{F}}$ *satisfies* a condition p, denoted $s^{\mathcal{F}} \models p$, iff (i) $p[C] \subseteq \mathsf{center}(s^{\mathcal{F}})$ and (ii) for every $L \in \mathcal{L}$ there exists an $s^L \in S^L$ s.t. $p[L] \subseteq s^L$ and $\mathsf{prices}(s^{\mathcal{F}})[s^L] < \infty$. We define the set of leaf actions *enabled* by a center state s^C as $\mathcal{A}^{\mathcal{L}}|_{s^C} := \{a^L \mid a^L \in \mathcal{A}^{\mathcal{L}} \wedge \mathsf{pre}(a^L)[C] \subseteq s^C\}$.

The *initial decoupled state* $s^{\mathcal{F}}_0$ is defined as $s^{\mathcal{F}}_0 := \langle \mathsf{center}(s^{\mathcal{F}}_0), \mathsf{prices}(s^{\mathcal{F}}_0) \rangle$, where $\mathsf{center}(s^{\mathcal{F}}_0) = I[C]$. Its pricing function is given, for each $L \in \mathcal{L}$, as $\mathsf{prices}(s^{\mathcal{F}}_0)[s^L_0] = 0$, where $s^L_0 = I[L]$; and elsewhere as $\mathsf{prices}(s^{\mathcal{F}}_0)[s^L] = c_{s^{\mathcal{F}}_0}(s^L_0, s^L)$, where $c_{s^{\mathcal{F}}_0}(s^L_0, s^L)$ is the cost of a cheapest path of $\mathcal{A}^L|_{\mathsf{center}(s^{\mathcal{F}}_0)} \setminus \mathcal{A}^C$ actions from s^L_0 to s^L. If no such path exists $c_{s^{\mathcal{F}}_0}(s^L_0, s^L) = \infty$. The set of *decoupled goal states* is $S^{\mathcal{F}}_G := \{s^{\mathcal{F}}_G \mid s^{\mathcal{F}}_G \models G\}$.

Decoupled-state transitions are induced only by center actions, where a center action a^C is *applicable* in a decoupled state $s^{\mathcal{F}}$ if $s^{\mathcal{F}} \models \mathsf{pre}(a^C)$. By $S^L_{a^C}(s^{\mathcal{F}})$ we define the set of leaf states of L in $s^{\mathcal{F}}$ that *comply* with the leaf precondition of a^C, i. e., $S^L_{a^C}(s^{\mathcal{F}}) := \{s^L \mid \mathsf{pre}(a^C)[L] \subseteq s^L \wedge \mathsf{prices}(s^{\mathcal{F}})[s^L] < \infty\}$. Applying a^C to $s^{\mathcal{F}}$ results in the decoupled state $t^{\mathcal{F}} = s^{\mathcal{F}}[\![a^C]\!]$ as follows: $\mathsf{center}(t^{\mathcal{F}}) = \mathsf{center}(s^{\mathcal{F}})[\![a^C]\!]$, and $\mathsf{prices}(t^{\mathcal{F}})[t^L] = \min_{s^L \in S^L_{a^C}(s^{\mathcal{F}})}(\mathsf{prices}(s^{\mathcal{F}})[s^L] + c_{t^{\mathcal{F}}}(u^L, t^L))$, where $s^L[\![a^C]\!] = u^L$.

By $\pi^C(s^{\mathcal{F}})$ we denote the center path that starts in $s^{\mathcal{F}}_0$ and ends in $s^{\mathcal{F}}$. Accordingly, we define the g-value of $s^{\mathcal{F}}$ as $g(s^{\mathcal{F}}) = \mathsf{cost}(\pi^C(s^{\mathcal{F}}))$, the cost of its center path.

A decoupled state $s^{\mathcal{F}}$ represents a set of explicit states, which takes the form of a *hypercube* whose dimensions are the leaf factors $\mathcal{L}$. Hypercubes are defined as follows:

[1] We remark that actions can be center and leaf action at the same time, so possibly $\mathcal{A}^{\mathcal{L}} \cap \mathcal{A}^C \neq \emptyset$.

Definition 1 (Hypercube) *Let Π be a planning task, and $\mathcal{F}$ a star factoring. Then a state s of Π is a* member state *of a decoupled state $s^{\mathcal{F}}$, if $s[C] = \mathsf{center}(s^{\mathcal{F}})$ and, for all leaves $L \in \mathcal{L}$, $\mathsf{prices}(s^{\mathcal{F}})[s[L]] < \infty$. We say that s has* price $\mathsf{price}(s^{\mathcal{F}}, s)$ *in $s^{\mathcal{F}}$, where $\mathsf{price}(s^{\mathcal{F}}, s) := \sum_{L \in \mathcal{L}} \mathsf{prices}(s^{\mathcal{F}})[s[L]]$. The* hypercube *of $s^{\mathcal{F}}$, denoted $[s^{\mathcal{F}}]$, is the set of all member states of $s^{\mathcal{F}}$.*

The hypercube of $s^{\mathcal{F}}$ captures both, the reachability and the prices of all member states s of $s^{\mathcal{F}}$. For every member state s of a decoupled state $s^{\mathcal{F}}$, we can construct the *global plan*, i. e., the sequence of actions that starts in I and ends in s by augmenting $\pi^C(s^{\mathcal{F}})$ with cheapest-compliant leaf paths, i. e., leaf action sequences that lead to the pricing function of $s^{\mathcal{F}}$. The cost of member states in a hypercube only takes into account the cost of the leaf actions, since center action costs are not included in the pricing function. The cost of a plan reaching a member state s of $s^{\mathcal{F}}$ from I can be computed as follows: $\mathsf{cost}(s^{\mathcal{F}}, s) = g(s^{\mathcal{F}}) + \mathsf{price}(s^{\mathcal{F}}, s)$.

Dominance Pruning for Decoupled Search

Prior work on decoupled search has only considered dominance pruning instead of exact duplicate checking (Torralba et al. 2016; Gnad and Hoffmann 2018). With dominance pruning, instead of duplicate states, the search prunes decoupled states that are *dominated* by an already visited decoupled state (with lower g-value). We formally define the dominance relation $\preceq_B \subseteq S^{\mathcal{F}} \times S^{\mathcal{F}}$ over decoupled states as $(t^{\mathcal{F}}, s^{\mathcal{F}}) \in \preceq_B$ iff (1) $[t^{\mathcal{F}}] \subseteq [s^{\mathcal{F}}]$ and (2) for all $s^L \in S^{\mathcal{L}}$: $\mathsf{prices}(s^{\mathcal{F}})[s^L] \leq \mathsf{prices}(t^{\mathcal{F}})[s^L]$. Instead of $(t^{\mathcal{F}}, s^{\mathcal{F}}) \in \preceq_B$, we often write $t^{\mathcal{F}} \preceq_B s^{\mathcal{F}}$ to denote that $s^{\mathcal{F}}$ *dominates* $t^{\mathcal{F}}$. Note that (2) is only required for optimal planning. In satisficing planning we can simply set the price of all reached leaf states to 0, ignoring the leaf action costs completely. In practice these checks are performed by first comparing the center states $\mathsf{center}(s^{\mathcal{F}}) = \mathsf{center}(t^{\mathcal{F}})$ via hashing, followed by a component-wise comparison of the prices of reached leaf states.

Exact Duplicate Checking

In explicit state search, duplicate checking is performed to avoid unnecessary repeated handling of the same state. This can be implemented efficiently by means of hashing functions: if a state is re-visited during search – and, in case of optimal planning using A*, the path on which it is reached is not cheaper than its current g-value – the new state can be pruned safely. In this section, we will look into exact duplicate checking for decoupled search, showing how an efficient hashing can be implemented.

Formally, we define the *duplicate state relation* over decoupled states $\preceq_D \subseteq S^{\mathcal{F}} \times S^{\mathcal{F}}$ as the identity relation where $(t^{\mathcal{F}}, s^{\mathcal{F}}) \in \preceq_D$ iff $s^{\mathcal{F}} = t^{\mathcal{F}}$. Like in explicit state search, a decoupled state $t^{\mathcal{F}}$ can safely be pruned if there exists an already visited state $s^{\mathcal{F}}$ where $g(s^{\mathcal{F}}) \leq g(t^{\mathcal{F}})$ and $t^{\mathcal{F}} \preceq_D s^{\mathcal{F}}$.

In decoupled search, a search node, i. e., a decoupled state $s^{\mathcal{F}}$, does not represent a single state, but a set of states, namely its hypercube $[s^{\mathcal{F}}]$. As a consequence, duplicate checking is less effective. This is because the chances of finding a decoupled state with the exact same hypercube (including leaf state prices) are smaller than finding a duplicate in explicit state search. Importantly, care must be taken when hashing decoupled states, to properly take into account both reachability *and* prices of leaf states.

In order to hash decoupled states, we need a canonical form that provides a unique representation of a decoupled state. We achieve this by, prior to the search, constructing *all* reachable leaf states s^L for each leaf L, over-approximating reachability by projecting the task onto L. This ignores all interaction between the center and the leaf, assuming that action preconditions on $\mathcal{V} \setminus L$ are always achieved. The resulting transition systems are called the *leaf state spaces* for every leaf $L \in \mathcal{L}$. Given the leaf state spaces, we assign a unique ID to every leaf state, starting with 0, up to $|S^L_R| - 1$, where S^L_R is the set of leaf states of L that can be reached from $I[L]$ in the leaf state space of L.

With the leaf state IDs, we can efficiently store the pricing function of each leaf $L \in \mathcal{L}$ of a decoupled state $s^{\mathcal{F}}$ as an array A of numbers, where $A[i]$ contains the price of the leaf state with ID i. To get a canonical representation of $s^{\mathcal{F}}$, and to keep the memory footprint of its pricing function as small as possible, we decide to limit the size of the array to just fit the highest ID of a leaf state with finite price. Implicitly, all leaf states with a higher ID are not reached in $s^{\mathcal{F}}$ and have cost ∞. This does incur a memory overhead, in the extreme case wasting $|S^L_R|-1$ entries in the array, if only the leaf state with ID $|S^L_R| - 1$ is reached, so the entries for all other leaf states are ∞. However, leaf state spaces are mostly "well-behaved" in the sense that such pathologic behaviour does not usually occur.

In non-optimal planning, where, as previously noted, we do not require the actual leaf state prices, but only reachability information, we only keep a bitvector A for each leaf $L \in \mathcal{L}$, indicating whether the leaf state with ID i is reached if $A[i] = \top$.

The unique IDs and the maintenance of the pricing function as standard arrays allow the use of hashing functions, where two decoupled states can only be equal if the hashes of their center state, and for each leaf factor, the hashes of the representation of the pricing functions match.

Improved Dominance for Optimal Planning

In this section, we introduce two improvements over the basic dominance relation $\preceq_B$ for optimal planning. The first one incorporates the g-value of decoupled states into the dominance check when comparing the leaf-state prices. This increases the potential for pruning, allowing to prune states that have a lower g-value. The second technique is a decoupled-state transformation that moves part of the leaf-state prices into the g-value of a decoupled state, enhancing search guidance by fully accounting for costs that have to be spent to reach the cheapest member state.

Incorporating the g-value in Dominance Checking

In optimal planning, a decoupled state $t^{\mathcal{F}}$ can only be pruned with $\preceq_B$ if there exists an already visited state $s^{\mathcal{F}}$ with *lower g-value* that dominates it. Doing the dominance check in

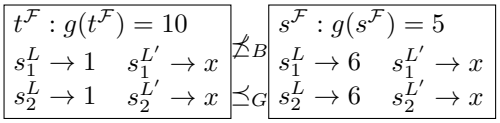


Figure 1: Illustrating example where $s^{\mathcal{F}}$ can only be pruned when using the new dominance relation $\preceq_G$.

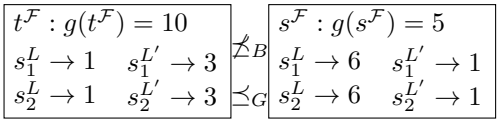


Figure 2: Example where $t^{\mathcal{F}}$ has lower prices in leaf L, higher prices in L', and $\preceq_G$ detects that $s^{\mathcal{F}}$ dominates $t^{\mathcal{F}}$.

this way, however, separately considers g-values and pricing function, where these in fact can be combined. We next show that the g-value difference of two decoupled states can be traded against differences in the pricing function. To see this, recall the definition of the cost of a member state s of a decoupled state $s^{\mathcal{F}}$:

$$\begin{aligned}\mathsf{cost}(s^{\mathcal{F}}, s) &= g(s^{\mathcal{F}}) + \mathsf{price}(s^{\mathcal{F}}, s) \\ &= g(s^{\mathcal{F}}) + \sum_{L \in \mathcal{L}} \mathsf{prices}(s^{\mathcal{F}})[s[L]]\end{aligned}$$

Instead of only comparing the pricing function to visited states with lower g-value, we can directly compare the costs of the member states to all visited states, independent of their g-values. Then, a new decoupled state $t^{\mathcal{F}}$ can be pruned if there exists a visited state $s^{\mathcal{F}}$ s.t. all member states s of $t^{\mathcal{F}}$ have lower cost in $s^{\mathcal{F}}$: $\forall s \in [t^{\mathcal{F}}] : \mathsf{cost}(s^{\mathcal{F}}, s) \leq \mathsf{cost}(t^{\mathcal{F}}, s)$. If all member states s of a decoupled state $t^{\mathcal{F}}$ are contained with lower cost in an already visited decoupled state $s^{\mathcal{F}}$, then analogously to pruning duplicate states with higher g-value in explicit search, we can safely prune $t^{\mathcal{F}}$.

Consider the example in Figure 1. Each box represents a decoupled state, and an arrow $s_i^L \to 6$ indicates that in a state $s^{\mathcal{F}}$ we have $\mathsf{prices}(s^{\mathcal{F}})[s_i^L] = 6$. Say $s^{\mathcal{F}}$ is visited and $t^{\mathcal{F}}$ is a new state, where $g(t^{\mathcal{F}}) = 10$ and $g(s^{\mathcal{F}}) = 5$. Further, the prices in all but one leaf factor L of both states are identical. In leaf L, we have $\mathsf{prices}(s^{\mathcal{F}})[s^L] = \mathsf{prices}(t^{\mathcal{F}})[s^L] + 5$, so all leaf states of L in $t^{\mathcal{F}}$ are cheaper by a cost of 5, but $s^{\mathcal{F}}$ has a g-value that is by 5 lower than that of $t^{\mathcal{F}}$. With the dominance relation $\preceq_B$ from prior work, $t^{\mathcal{F}}$ cannot be pruned, because its prices are lower than the ones of $s^{\mathcal{F}}$. However, the cost of all its member states is equal to the cost of the states in $s^{\mathcal{F}}$, so it is actually safe to prune $t^{\mathcal{F}}$.

An important question is how to compute this efficiently, i. e., without explicitly enumerating the costs of all member states. We next show that, similar to $\preceq_B$, dominance can be checked component-wise by only considering, for each leaf L, the leaf state with the highest price difference.

Formally, we define the *g-value aware dominance relation* $\preceq_G \subseteq S^{\mathcal{F}} \times S^{\mathcal{F}}$ as follows:

$$\begin{aligned}&(t^{\mathcal{F}}, s^{\mathcal{F}}) \in \preceq_G \Leftrightarrow\ g(t^{\mathcal{F}}) - g(s^{\mathcal{F}}) \geq \\ &\sum_{L \in \mathcal{L}} max_{s^L \in S_R^L}(\mathsf{prices}(s^{\mathcal{F}})[s^L] - \mathsf{prices}(t^{\mathcal{F}})[s^L]), \\ &\text{where } S_R^L = \{s^L \in S^L \mid \mathsf{prices}(t^{\mathcal{F}})[s^L] < \infty\}\end{aligned}$$

If $t^{\mathcal{F}}$ has a higher g-value than $s^{\mathcal{F}}$, but has leaf states with a lower price, then the disadvantage in g-value can be traded against the advantage in leaf state prices. More concretely, it suffices to sum-up only the maximal price-difference of any leaf state over the leaves. Thereby, we essentially compare only the member state $s \in [t^{\mathcal{F}}]$ for which the price-advantage is maximal. This can be done component-wise, so is efficient to compute. Indeed, $\preceq_G$ detects that $t^{\mathcal{F}}$ in the above example is dominated and can be pruned.

Theorem 1 *Let $s^{\mathcal{F}}$ and $t^{\mathcal{F}}$ be two decoupled states. Then $t^{\mathcal{F}} \preceq_G s^{\mathcal{F}}$ iff for all $s \in [t^{\mathcal{F}}] : \mathsf{cost}(t^{\mathcal{F}}, s) \geq \mathsf{cost}(s^{\mathcal{F}}, s)$.*

Proof Sketch: Let s be the member state of $t^{\mathcal{F}}$ where $\mathsf{prices}(s^{\mathcal{F}})[s[L]] - \mathsf{prices}(t^{\mathcal{F}})[s[L]]$ is maximal for all $L \in \mathcal{L}$. If $\mathsf{prices}(s^{\mathcal{F}}, s) - \mathsf{prices}(t^{\mathcal{F}}, s) \leq g(t^{\mathcal{F}}) - g(s^{\mathcal{F}})$, then this also holds for all other $s' \in [t^{\mathcal{F}}]$. With $\mathsf{cost}(t^{\mathcal{F}}, s') = g(t^{\mathcal{F}}) + \mathsf{prices}(t^{\mathcal{F}}, s')$ the claim follows. □

The new relation $\preceq_G$ also tackles more subtle cases, where prices differ in several leaf factors. We then need to *distribute* the difference in g-values *across* the leaf factors, i. e., we cannot use the full difference for each factor. However, we can even trade lower prices in one leaf by higher prices in another, incorporating these different prices in the g-difference. Consider the example in Figure 2, which extends the previous example by a leaf factor L' where $\mathsf{prices}(t^{\mathcal{F}})[s^{L'}] = \mathsf{prices}(s^{\mathcal{F}})[s^{L'}] + 2$ for all $s^{L'} \in S^{L'}$. We can then combine the price advantage of $+2$ in L' for $s^{\mathcal{F}}$ with its g-advantage $+5$ to make up for a total price disadvantage of 7 in other leaves, where $t^{\mathcal{F}}$ might have lower prices:

$$\begin{aligned}&g(t^{\mathcal{F}}) - g(s^{\mathcal{F}}) = 5 \\ &\geq \sum_{L \in \mathcal{L}} max_{s^L \in S_R^L}(\mathsf{prices}(s^{\mathcal{F}})[s^L] - \mathsf{prices}(t^{\mathcal{F}})[s^L]) \\ &= (6-1) + (1-3) = 3 \implies t^{\mathcal{F}} \preceq_G s^{\mathcal{F}}\end{aligned}$$

In this case, given that $s^{\mathcal{F}}$ is visited before $t^{\mathcal{F}}$ during search, we can prune $t^{\mathcal{F}}$, although the prices of its leaf states are neither lower-equal, nor higher-equal than the prices of $t^{\mathcal{F}}$. There is even a difference of cost 2 left that could be used to trade higher prices of $s^{\mathcal{F}}$ in another leaf factor.

g-Value Adaptation

We next introduce a canonical form which moves as much of the leaf-state prices into the g-value of a decoupled state as possible. Assume that, in a decoupled state $s^{\mathcal{F}}$, there exists a leaf L such that all leaf states s^L have a minimum non-zero price p, so $\forall s^L \in S^L : \mathsf{prices}(s^{\mathcal{F}})[s^L] \geq p$. Then we can reduce the prices of all these leaf states by p and increase $g(s^{\mathcal{F}})$ by p without affecting the cost $\mathsf{cost}(s^{\mathcal{F}}, s)$

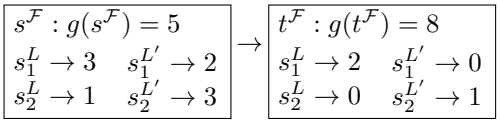


Figure 3: A decoupled state $s^{\mathcal{F}}$ and its g-adapted variant $t^{\mathcal{F}}$.

of the member states s of $s^{\mathcal{F}}$. Intuitively, the transformation moves the price that has to be spent to reach the cheapest member state of $s^{\mathcal{F}}$ into its g-value, reducing the price of all leaf states accordingly, so that in every leaf L there exists at least one leaf state with price 0. See Figure 3 for an example of a decoupled state $s^{\mathcal{F}}$ and its canonical representative $t^{\mathcal{F}}$.

The main advantage of adapting the g-value of a decoupled state occurs when executing decoupled search using the A* algorithm. Here, on a cost layer f the search usually prioritizes states with lower heuristic value. By moving cost into the g-value we achieve that the heuristic of a decoupled state (which takes into account the pricing function (Gnad and Hoffmann 2018)) can only get lower, aiding A* to focus on more promising states. A second important effect is that the part of the prices moved into the g-value will always be considered entirely by the search, whereas heuristics (in the extreme case blind search) might not be able to capture all the cost encoded in the pricing function.

Note that the g-value adaptation is independent of the new dominance relation $\preceq_G$. It can have a positive impact on the number of state expansions of $\preceq_G$, the base dominance check $\preceq_B$, and exact duplicate checking $\preceq_D$.

Efficient Implementation

In this section, we propose two optimizations that aim at making the dominance check more efficient. First, we show that with *invertible* leaf state spaces the comparison of leaf reachability can be entirely avoided. Second, we show how to exploit the *transitivity* of the dominance relation to focus the checking on the relevant subset of decoupled states. Both optimizations do not affect the pruning behavior.

Invertible Leaf State Spaces

Given the precomputed leaf state spaces as described in the previous section, it is straightforward to compute the connectivity of these graphs. In particular, we can efficiently check if a leaf state space is strongly connected when only considering transitions of leaf actions that do not affect, nor are preconditioned by, the center factor. Formally, we define the set of *no-center* actions of a leaf L as $\mathcal{A}^L_{\neg C} := \{a^L \in \mathcal{A}^L \mid vars(\mathsf{pre}(a)) \cap C = \emptyset \wedge vars(\mathsf{eff}(a)) \cap C = \emptyset\}$.

Let S^L_R be the set of L-states that is reachable from $I[L]$ in the projection onto L using all actions $\mathcal{A}$. Let further $S^L_R|_{\mathcal{A}^L_{\neg C}}$ be the corresponding set using only the no-center actions $\mathcal{A}^L_{\neg C}$ of L. We say that L is *leaf-invertible*, if $S^L_R = S^L_R|_{\mathcal{A}^L_{\neg C}}$, i. e., any L-state reachable from $I[L]$ can be reached using no-center actions, and the part of the leaf state space induced by S^L_R and $\mathcal{A}^L_{\neg C}$ is strongly connected.

Proposition 1 *Let L be leaf-invertible and S^L_R the set of L-states reachable from $I[L]$, then in every decoupled state $s^{\mathcal{F}}$ reachable from $s^{\mathcal{F}}_0$, the set of reached L-states in $s^{\mathcal{F}}$ is S^L_R.*

Proof: In $s^{\mathcal{F}}_0$, the claim trivially holds. Let $s^{\mathcal{F}}$ be a (not necessarily direct) successor of $s^{\mathcal{F}}_0$. The center action that generates $s^{\mathcal{F}}$ can possibly restrict the set of compliant leaf states S^L_a, and affect the remaining ones, resulting in a set of leaf states that is a subset of S^L_R. Since S^L_R is strongly connected by $\mathcal{A}^L_{\neg C}$, all L-states of S^L_R have a finite price in $s^{\mathcal{F}}$. □

All decoupled states reached during search can only differ in the leaf-state prices for leaf-invertible factors, but will always have the same set of leaf states reached. Thus, at least for satisficing planning, these leaves do not need to be compared in the dominance check at all. For optimal planning, we still need to compare the prices, since these might differ.

Another minor optimization that can be performed with the leaf-invertibility information is successor generation during search. When computing the center actions that are applicable in a decoupled state, we usually need to check leaf preconditions by looking for a reached leaf state that enables an action. For leaf-invertible leaf factors, however, this check is no longer needed (even for optimal planning), because the set of reached leaf states remains constant. We precompute the set of applicable center actions, and skip the check for leaf preconditions on leaf-invertible factors.

Transitivity of the Dominance Relation

In explicit state search, a duplicate state can be pruned if it has already been visited (with a lower g-value). This can be efficiently implemented using a hash table. In decoupled search with dominance pruning, the corresponding check needs to iterate over *all* previously visited states (with a lower g-value) that have the same center state, and compare the pricing function.

Instead of iterating over *all* visited decoupled states, though, we can exploit the transitivity of our dominance relations to focus on the relevant visited states, namely those that are not themselves dominated by another visited state.

Proposition 2 *Let V be the set of decoupled state already visited during search and let $t^{\mathcal{F}}$ a newly generated decoupled state. If there exist $s^{\mathcal{F}}_1, s^{\mathcal{F}}_2 \in V$ such that $s^{\mathcal{F}}_1 \preceq s^{\mathcal{F}}_2$, where $\preceq$ is a transitive relation over decoupled states, then $t^{\mathcal{F}} \not\preceq s^{\mathcal{F}}_2$ implies $t^{\mathcal{F}} \not\preceq s^{\mathcal{F}}_1$.*

Clearly, we do not need to check dominance of $t^{\mathcal{F}}$ against $s^{\mathcal{F}}_1$, but only need to compare $s^{\mathcal{F}}_2$ and $t^{\mathcal{F}}$ to see if $t^{\mathcal{F}}$ can be pruned. During search, we incrementally compute the set of "dominated visited states" as a side product of the dominance check. If a new state $t^{\mathcal{F}}$ dominates an existing state $s^{\mathcal{F}}_1$, then either there exists another visited state $s^{\mathcal{F}}_3$ that dominates $t^{\mathcal{F}}$, so it will be pruned, or there is no state yet that dominates $t^{\mathcal{F}}$. In both cases, $s^{\mathcal{F}}_1$ can be skipped in every future dominance check because there exists another visited state, either $s^{\mathcal{F}}_3$ or $t^{\mathcal{F}}$, that is visited and that dominates it.

Experimental Evaluation

We implemented all proposed methods in the decoupled search planner by Gnad & Hoffmann (2018), which itself builds on the Fast Downward planning system (Helmert

		Blind Search										A* with $h^{\text{LM-cut}}$									
		Dominance Pruning						Duplicate Checking				Dominance Pruning						Duplicate Checking			
Domain	#	$\preceq_B$	$\preceq_B^{IT}$	$\preceq_B^{gIT}$	$\preceq_G^{IT}$	$\preceq_G^{g}$	$\preceq_G^{gIT}$	$\preceq_D$	$\preceq_D^{I}$	$\preceq_D^{g}$	$\preceq_D^{gI}$	$\preceq_B$	$\preceq_B^{IT}$	$\preceq_B^{gIT}$	$\preceq_G^{IT}$	$\preceq_G^{g}$	$\preceq_G^{gIT}$	$\preceq_D$	$\preceq_D^{I}$	$\preceq_D^{g}$	$\preceq_D^{gI}$
DataNet	20	**9**	**9**	5	**9**	**9**	**9**	5	5	5	5	**14**	**14**	12	**14**	**14**	**14**	12	12	12	12
Depots	22	3	3	**4**	**4**	**4**	**4**	2	2	**4**	**4**	**7**	**7**	**7**	**7**	**7**	**7**	5	5	**7**	**7**
Driverlog	20	**11**	**11**	**11**	**11**	**11**	**11**	9	9	10	10	13	13	13	13	13	13	13	13	13	13
Elevators	30	6	6	9	12	**16**	**16**	0	0	10	10	10	11	22	13	**23**	**23**	0	0	22	22
Floortile	40	**2**	**2**	**2**	**2**	**2**	**2**	0	0	0	0	**10**	**10**	**10**	**10**	**10**	**10**	5	5	5	5
Freecell	42	0	0	0	0	0	0	0	0	**2**	**2**	1	1	**2**	**2**	**2**	**2**	1	1	**2**	**2**
GED	20	13	13	**15**	**15**	**15**	**15**	7	7	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	13	13	**15**	**15**
Grid	5	1	1	1	1	1	1	1	1	1	1	**2**	**2**	**2**	**2**	**2**	**2**	1	1	**2**	**2**
Logistics	63	24	25	25	**26**	25	**26**	22	22	24	24	34	34	35	34	**36**	**36**	30	30	34	34
Miconic	145	46	**47**	**47**	**47**	45	**47**	42	42	42	42	135	135	135	135	135	135	135	135	135	135
NoMystery	20	**20**	**20**	**20**	**20**	19	**20**	16	16	16	16	**20**	**20**	**20**	**20**	**20**	**20**	19	19	19	19
OpenSt14	20	1	**2**	1	**2**	**2**	1	**2**	**2**	**2**	**2**	**2**	**2**	1	**2**	**2**	1	**2**	**2**	**2**	**2**
PSR	48	**48**	**48**	46	**48**	**48**	**48**	42	42	46	46	**48**	**48**	47	**48**	**48**	**48**	45	45	47	47
Rovers	40	**7**	**7**	**7**	**7**	**7**	**7**	6	6	**7**	**7**	8	8	8	8	8	8	8	8	8	8
Satellite	36	5	5	5	5	5	5	5	5	5	5	7	7	**9**	7	**9**	**9**	7	7	**9**	**9**
Tidybot14	10	3	3	3	3	3	3	3	3	3	3	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	5	**6**
Transport	28	10	11	13	14	**15**	**15**	0	0	13	13	12	12	**14**	13	**14**	**14**	6	6	**14**	**14**
Woodwork	26	7	7	7	7	7	7	7	7	7	7	16	16	16	**17**	**17**	**17**	16	16	16	16
Zenotravel	20	8	**9**	8	**9**	**9**	**9**	6	6	6	7	12	12	12	**13**	**13**	**13**	8	8	11	11
Others	239	67	67	67	67	67	67	67	67	67	67	87	87	87	87	87	87	87	87	87	87
$\sum$	894	291	296	296	309	310	**313**	242	242	285	286	459	460	473	466	**481**	480	419	419	465	466

Table 1: Coverage data for optimal planning with blind search and with A* using $h^{\text{LM-cut}}$. All configurations use the incident arcs factoring. Domains without difference in coverage are summarized in "Other". Best coverage is highlighted in **bold face**.

2006). We conducted our experiments using the lab python package (Seipp et al. 2017) on all benchmark domains of the International Planning Competition (IPC) from 1998-2018 in both the optimal and satisficing tracks. We also run decoupled search to prove planning tasks unsolvable, using the benchmarks of UIPC'16 and Hoffmann, Kissmann, and Torralba (2014). In all benchmark sets, we eliminated duplicate instances that appeared in several iterations. For optimal planning, we run blind search and A* with $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009); in satisficing planning, we use greedy best-first search (GBFS) with the h^{FF} heuristic without preferred operator pruning (Hoffmann and Nebel 2001); to prove unsolvability, we run A* with the $h^{\max}$ heuristic (Bonet and Geffner 2001). We use the common runtime/memory limits of 30min/4GiB. The code and experimental data of our evaluation are publicly available (https://doi.org/10.5281/zenodo.4061825).

Decoupled search needs a method that provides a factoring, i. e., that detects a star topology in the causal structure of the input planning task. We use two basic factoring methods, inverted-fork factorings (IF) – only for satisficing planning – as well as the incident arcs factoring (IA) as described in Gnad, Poser, and Hoffmann (2017). We expect IF factorings to nicely show the advantage of the more efficient handling of invertible leaf state spaces, since there are several domains that have such state spaces in this case, but not using IA. IA is the canonical choice since it is fast to compute and finds good decompositions in many domains.

We use the following naming convention for search configurations: we distinguish the three dominance relations $\preceq_B$, $\preceq_D$, and $\preceq_G$. We indicate the g-value adaptation, and the invertibility and transitivity optimizations by adding a superscript g, respectively I and T to the relation symbol, e. g. $\preceq_B^{gT}$ for a configuration that uses $\preceq_B$ and has the g-value adaptation and the transitivity optimization enabled.

Unsolvability				Satisficing			
Domain	#	$\preceq_B^{IT}$	$\preceq_D^{I}$	Domain	#	$\preceq_B^{IT}$	$\preceq_D^{I}$
Cavediving	21	**4**	2				
Diagnosis	20	**14**	13	Floortile	40	**5**	2
OverNoMy	24	**13**	12	NoMyst	20	**19**	16
OverTPP	30	**15**	14	Rovers	40	**21**	20
Other	182	88	88	Other	884	657	657
$\sum$	277	**134**	129	$\sum$	984	**702**	695

Table 2: Coverage data, like Table 1, for proving unsolvability and satisficing planning with preferred operators.

Tables 1 and 2 show coverage data (number of instances solved) for the benchmarks where the factoring methods are able to detect a star factoring. For optimal planning, Table 1, we see that both $\preceq_G$ and the g-adaptation individually lead to an increase in coverage across several domains for both blind search and A* with $h^{\text{LM-cut}}$. There even seems to be a positive correlation, shown by the fact that the combination $\preceq_G^{g}$ outperforms both its components. We do not separately evaluate the invertibility optimization, since the only change in the successor generation does not influence coverage a lot. Compared to $\preceq_B$, $\preceq_B^{I}$ only differs in coverage by $+1$ in Openstacks for blind search. The main advantage of the $\preceq_x^{IT}$ configurations stems from the transitivity optimization.

The duplicate checking configurations without g-adaptation show a significant drop in coverage compared to $\preceq_B$, so although the checking is computationally more

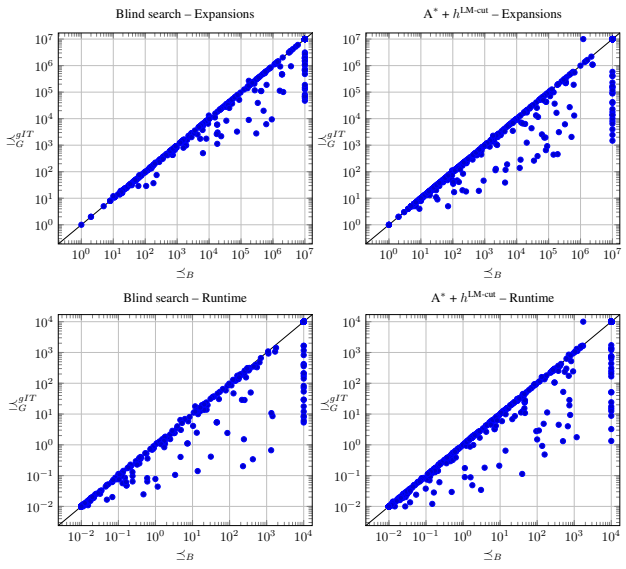


Figure 4: Scatter plots comparing runtime and number of state expansions for optimal planning.

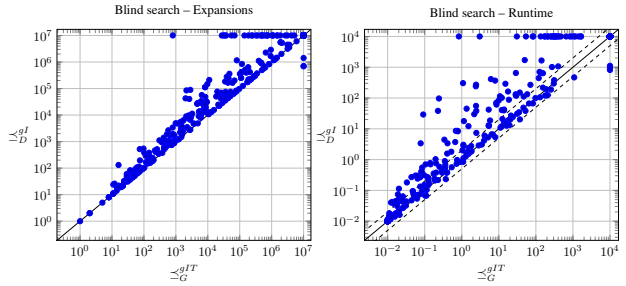


Figure 5: Like Figure 4, comparing dominance pruning to duplicate checking.

efficient, this does not even out the weaker pruning power. When enabling the g-adaptation, total coverage gets almost back to the level of $\preceq_B$ for blind search, and increases by 7 instances for $h^{\text{LM-cut}}$. Compared to $\preceq_B^{gIT}$, though, in both search variants $\preceq_D^{gI}$ drops in coverage. Surprisingly, duplicate checking can solve two Freecell instances that the dominance pruning cannot solve with blind search.

Table 2 has coverage (number of instances solved, resp. proved unsolvable) results for proving unsolvability and satisficing planning. We focus on the difference between dominance pruning and duplicate checking, as the invertibility and transitivity optimizations do not have an impact on coverage. Duplicate checking performs worse in several domains, does not affect coverage across many other domains, but never increases coverage. So even outside optimal planning, with strong pruning being less crucial, its use does in general not pay off.

The scatter plots in Figure 4 and 5 shed further light on the per-instance runtime and search space size comparison between some optimal planning configurations. Figure 4 shows the number of expanded states in the top row and the

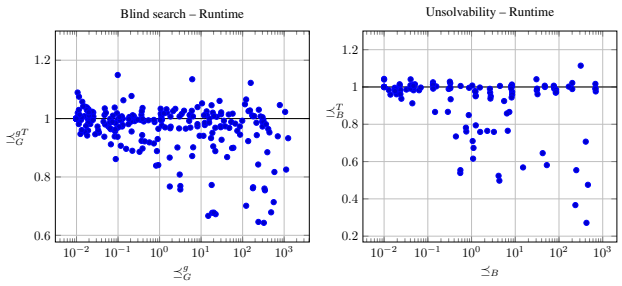


Figure 6: Improvement factors of $\preceq_y^{xT}$ over $\preceq_y^x$ showing the impact of the transitivity optimization in optimal planning and proving unsolvability.

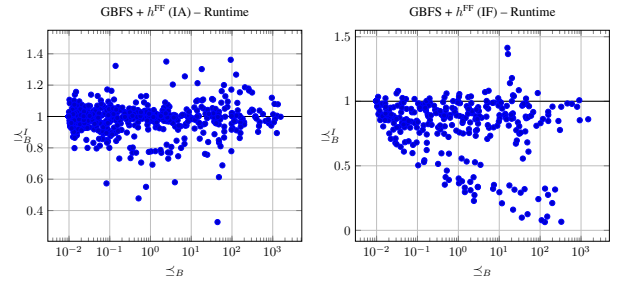


Figure 7: Like Figure 6, showing the impact of the invertibility optimization in satisficing planning with IA vs. IF.

runtime in the bottom row. All configurations use the IA factoring and compare $\preceq_B$ to $\preceq_G^{gIT}$, with blind search in the left column and $h^{\text{LM-cut}}$ in the right column. The advantage of the more clever dominance check, the g-adaptation, and our runtime optimizations is obvious, saving up to several orders of magnitude for state expansions and runtime.

Figure 5 illustrates the effect of exact duplicate checking. The left plot shows the expected increase in search space size, due to the reduced pruning power. The right plot indicates that where the increase in search space size is small the more efficient computation indeed pays off runtime-wise. The two dashed lines highlight a difference of a factor of 2, which can often be gained with duplicate checking.

Figures 6 and 7 show the impact of the transitivity and invertibility optimizations. The plots show per-instance runtime improvement factors of configuration Y on the y-axis over configuration X on the x-axis, where a y-value of a indicates that the runtime of Y is a times the runtime of X (values below 1 are a speed-up). The transitivity optimization clearly has a positive impact on runtime, reducing it up to 40% in optimal planning and up to 70% for proving unsolvability. The invertibility optimization (Figure 7) does not show such a clear picture when using the IA factoring (left plot). With IF, though, it indeed nicely accelerates the dominance check, as the optimization is applicable more often.

Conclusion

We have taken a closer look at the behavior and implementation details of dominance pruning in decoupled search. We

introduced exact duplicate checking, which, in spite of its weaker pruning, can improve search performance in practice due to higher computational efficiency under certain conditions. Furthermore, we developed two optimizations that make the dominance check more efficient to compute.

Our main contribution are two extensions of dominance pruning for optimal planning, that take the g-value of decoupled states into account. Both methods are highly beneficial and their combination significantly outperforms the baseline, improving the performance of decoupled search across many benchmark domains.

For future work, we think it is worthwhile to further investigate dominance pruning for decoupled search. A combination with the quantitative dominance pruning of Torralba (2017) could for example be interesting.

References

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A Stubborn Set Algorithm for Optimal Planning. In Raedt, L. D., ed., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, 891–892. Montpellier, France: IOS Press.

Amir, E.; and Engelhardt, B. 2003. Factored Planning. In Gottlob, G., ed., *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, 929–935. Acapulco, Mexico: Morgan Kaufmann.

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

Bonet, B.; and Geffner, H. 1999. Planning as Heuristic Search: New Results. In Biundo, S.; and Fox, M., eds., *Proceedings of the 5th European Conference on Planning (ECP'99)*, 60–72. Springer-Verlag.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1–2): 5–33.

Brafman, R.; and Domshlak, C. 2006. Factored Planning: How, When, and When Not. In Gil, Y.; and Mooney, R. J., eds., *Proceedings of the 21st National Conference of the American Association for Artificial Intelligence (AAAI'06)*, 809–814. Boston, Massachusetts, USA: AAAI Press.

Brafman, R. I.; and Domshlak, C. 2008. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, 28–35. AAAI Press.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8): 677–691.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

Edelkamp, S.; and Helmert, M. 1999. Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length. In Biundo, S.; and Fox, M., eds., *Proceedings of the 5th European Conference on Planning (ECP'99)*, 135–147. Springer-Verlag.

Edelkamp, S.; Leue, S.; and Lluch-Lafuente, A. 2004. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer* 6(4): 277–301.

Fabre, E.; Jezequel, L.; Haslum, P.; and Thiébaux, S. 2010. Cost-Optimal Factored Planning: Promises and Pitfalls. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, 65–72. AAAI Press.

Gnad, D.; and Hoffmann, J. 2018. Star-Topology Decoupled State Space Search. *Artificial Intelligence* 257: 24 – 60.

Gnad, D.; and Hoffmann, J. 2019. On the Relation between Star-Topology Decoupling and Petri Net Unfolding. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'19)*. AAAI Press.

Gnad, D.; Hoffmann, J.; and Wehrle, M. 2019. Strong Stubborn Set Pruning for Star-Topology Decoupled State Space Search. *Journal of Artificial Intelligence Research* 65: 343–392. doi:10.1613/jair.1.11576.

Gnad, D.; Poser, V.; and Hoffmann, J. 2017. Beyond Forks: Finding and Ranking Star Factorings for Decoupled Search. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. AAAI Press/IJCAI.

Gnad, D.; Torralba, Á.; Shleyfman, A.; and Hoffmann, J. 2017. Symmetry Breaking in Star-Topology Decoupled Search. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*. AAAI Press.

Godefroid, P.; and Wolper, P. 1991. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV'91)*, 332–342.

Hall, D.; Cohen, A.; Burkett, D.; and Klein, D. 2013. Faster Optimal Planning with Partial-Order Pruning. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*. Rome, Italy: AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability. In Schaub, T., ed., *Proceedings*

of the 21st European Conference on Artificial Intelligence (ECAI'14). Prague, Czech Republic: IOS Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting Problem Symmetries in State-Based Planners. In Burgard, W.; and Roth, D., eds., *Proceedings of the 25th National Conference of the American Association for Artificial Intelligence (AAAI'11)*. San Francisco, CA, USA: AAAI Press.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461. doi:10.5281/zenodo.790461. URL https://doi.org/10.5281/zenodo.790461.

Starke, P. 1991. Reachability analysis of Petri nets using symmetries. *Journal of Mathematical Modelling and Simulation in Systems Analysis* 8(4/5): 293–304.

Torralba, Á. 2017. From Qualitative to Quantitative Dominance Pruning for Optimal Planning. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4426–4432. AAAI Press/IJCAI.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence* 242: 52–79.

Torralba, Á.; Gnad, D.; Dubbert, P.; and Hoffmann, J. 2016. On State-Dominance Criteria in Fork-Decoupled Search. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press/IJCAI.

Torralba, Á.; and Hoffmann, J. 2015. Simulation-Based Admissible Dominance Pruning. In Yang, Q., ed., *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, 1689–1695. AAAI Press/IJCAI.

Valmari, A. 1989. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, 491–515.

Wehrle, M.; and Helmert, M. 2014. Efficient Stubborn Sets: Generalized Algorithms and Selection Strategies. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.

Wehrle, M.; Helmert, M.; Alkhazraji, Y.; and Mattmüller, R. 2013. The Relative Pruning Power of Strong Stubborn Sets and Expansion Core. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*. Rome, Italy: AAAI Press.

On the Optimal Efficiency of A* with Dominance Pruning

Álvaro Torralba
Department of Computer Science, Aalborg University, Denmark
alto@cs.aau.dk

Abstract

A well known result is that, given a consistent heuristic and no other source of information, A* does expand a minimal number of nodes up to tie-breaking. We extend this analysis for A* with dominance pruning, which exploits a dominance relation to eliminate some nodes during the search. We show that the expansion order of A* is not necessarily optimally efficient when considering dominance pruning with arbitrary dominance relations, but it remains optimally efficient under certain restrictions for the heuristic and dominance relation.

Introduction

Heuristic best-first search algorithms are a fundamental tool for problem solving whenever the problem can be modeled as finding paths in graphs. Heuristic functions guide the search towards the goal by estimating the distance from any given state to the goal. Whenever an optimal solution of minimum cost is required, A* search is often the algorithm of choice (Hart, Nilsson, and Raphael 1968). This is well supported by the well known result that, given a consistent heuristic h and no other source of information, A* does expand a minimal number of nodes up to tie-breaking among all algorithms that guarantee finding the optimal solution (Dechter and Pearl 1985).

Dominance pruning is a technique to eliminate nodes during the search if they can be proven to be dominated by another state (Hall et al. 2013; Torralba and Hoffmann 2015). This exploits an additional source of information in the form of a dominance relation $\preceq$, which compares two states to determine whether one can be proven to be as close to the goal as the other. This type of dominance appears naturally on problems that have to deal with resources, (i.e., removing states that have strictly less resources than another), and can also be applied on other kinds of problems (e.g., in gridworlds being at a central square can sometimes be proven better than being at a corner if the set of reachable squares in one step is strictly larger). This can be exploited by any search algorithm to reduce the number of nodes explored while retaining any solution optimality guarantees. This has been mainly used in the context of cost-optimal planning, as an enhancement for the A* algorithm.

In this paper, we address the question of whether the expansion order of A* is good to minimize the number of expansions when dominance pruning is used. Prioritizing the expansion of states with lower f-value is not necessarily an obvious choice anymore, since states that are more promising according to the heuristic function are not necessarily better according to the dominance relation. Furthermore, previous results proving the optimal efficiency of A* are no longer valid due to having a new source of information.

Indeed, we show that there are cases where A* with dominance pruning is not optimally efficient, and that different expansion orderings, or even expanding some states that could be pruned lead to a globally higher number of expansions in some cases. However, these cases can be attributed to "inconsistencies" in the information provided by the heuristic function and the dominance relation. We extend the notion of consistent heuristics to consistent heuristic and dominance relation pairs, and prove that A* with dominance pruning is indeed optimally efficient, meaning that there is a tie-breaking for A* that is guaranteed to expand the lowest number of nodes among all algorithms with admissible heuristics and dominance pruning.

We also analyze which tie-breaking strategies remain optimally efficient up to the last f-layer, i.e., when we ignore the expansions of nodes with an f-value equal to the solution cost. This is relevant because when consistent heuristics are used, the choice of tie-breaking rule in A* is only relevant for the last layer, since all nodes with an f-value lower than the optimal solution cost must be expanded regardless of the expansion order. Therefore, most implementations of A* choose tie-breaking strategies in favor of nodes with lower h-value, which are expected to find a solution faster in the last f-layer. We show that with dominance pruning this is no longer the case, as tie-breaking strategies in favor of nodes with lower g-value are preferable up to the last layer.

Background

A *transition system* (TS) is a tuple $\Theta = \langle S, L, T, s^I, S^G \rangle$ where S is a finite set of *states*, L is a finite set of *labels* each associated with a *label cost* $c(l) \in \mathbb{R}_0^+$, $T \subseteq S \times L \times S$ is a set of *transitions*, $s^I \in S$ is the *start state*, and $S^G \subseteq S$ is the set of *goal states*. We write $s \xrightarrow{l} t$ as a shorthand for $(s, l, t) \in T$. A *plan* for a state s is a path from s to any $s_G \in S^G$. We use $h^*(s)$ ($g^*(s)$) to denote the cost of a cheapest plan for s (path from s^I to s). A plan for s is *optimal* iff its cost equals $h^*(s)$. The sum $f^*(s) = g^*(s) + h^*(s)$ is the cost

of an optimal plan from s^I passing through s. We denote F^* to the optimal solution cost for s^I, $F^* = f^*(s^I) = h^*(s^I)$.

To deal with tasks with 0-cost actions, we define a modified cost function c_ϵ so that all 0-cost actions are assigned a cost of ϵ, where ϵ is a tiny constant such that the sum of arbitrarily many ϵ will still be lower than any other action cost greater than 0. We define $g_\epsilon, h^*_\epsilon, f_\epsilon, \ldots$ as the functions above under this new cost function.

A *heuristic* $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ is a function that estimates goal distance. It is *admissible* if it never overestimates the real cost, i.e., $h(s) \leq h^*(s)$ for all $s \in S$, and *consistent* if for any $s \xrightarrow{l} t$ it holds that $h(s) \leq h(t) + c(l)$.

Best-first search algorithms maintain an open and a closed list with all nodes that have been seen so far. A search node n_s characterizes a path from the initial state to the final state of the path, s, where the g-value $g(n_s)$ is the cost of the path. We write $n_s \xrightarrow{l} n_t$ as a shorthand for $s \xrightarrow{l} t$ and $g(n_t) = g(n_s) + c(l)$. The open list is initialized with the initial state that has a g-value of 0. At each step, a node is selected from the open list for expansion. When a node is expanded, it is removed from open and all the successors are generated and inserted into the open list. The closed list keeps all nodes that have been expanded to avoid duplicates so that a node is not expanded if another node with the same state and a lower or equal g-value has already been expanded. A* always selects for expansion a node with minimum f-value where $f(n_s) = g(n_s) + h(s)$. Since the behavior of A* is not uniquely defined, we say that it is a family of algorithms, one per possible tie-breaking rule.

Optimal Efficiency of A*

The seminal work by Dechter and Pearl (1985) analyzes the optimal efficiency of A* in great depth, considering several degrees of optimal efficiency. They consider the heuristic as part of the input to the algorithm, so a problem instance is a tuple $\langle \Theta, h \rangle$. An instance is consistent if it has a consistent heuristic h. An algorithm is admissible if it is guaranteed to return an optimal plan for Θ, whenever h is admissible.

To prove optimal efficiency of an algorithm, some assumptions about the considered algorithms are needed. In their paper, Dechter and Pearl define a family of algorithms that use only a few primitive functions, such as expansion and heuristic functions. Eckerle et al. (2017) refine this by making explicit the assumption that all these functions are deterministic, and black box, defining the family of Deterministic, Expansion-based, Black Box (DXBB) algorithms. We also assume that the transition relation can only be accessed in a forward manner, as a function that given a state returns its successors. If backward search is possible, A* does not guarantee optimal efficiency (Chen et al. 2017).

Definition 1 (*UDXBB* Algorithm)**.** *A algorithm is Unidirectional, Deterministic, Expansion-based, Black Box (UDXBB) if it is deterministic and it has access to the state space Θ via exactly the following functions:*

- *Start: returns the initial state s^I.*
- *Is-goal: given a state s returns true iff s is a goal state.*
- *Expand: given a state s returns a set of successor states $expand(s) = \{t \mid s \xrightarrow{l} t\}$.*
- *Cost: given a state and a successor state returns the cost of reaching it ($cost(s,t) = \min_{c(l)} s \xrightarrow{l} t$).*

They define a hierarchy with several degrees of optimality, based on comparing the sets of nodes expanded by different families of algorithms over a set of instances. Let $N(A, I)$ be the set of expanded nodes by algorithm A on instance I. A family of algorithms $\mathcal{A}$ is X-optimally efficient over another $\mathcal{B}$ relative to an instance set $\mathcal{I}$ if:

- Type 0: $\forall I \in \mathcal{I}, \forall B \in \mathcal{B}, \forall A \in \mathcal{A}, N(A, I) \subseteq N(B, I)$.
- Type 1: $\forall I \in \mathcal{I}, \forall B \in \mathcal{B}, \exists A \in \mathcal{A}, N(A, I) \subseteq N(B, I)$.
- Type 2: $\forall I \in \mathcal{I}, \forall B \in \mathcal{B}, \forall A \in \mathcal{A}, N(B, I) \not\subset N(A, I)$.
- Type 3: $\forall B \in \mathcal{B}, \forall A \in \mathcal{A}, (\exists I_1 \in \mathcal{I}, N(A, I_1) \not\subseteq N(B, I_1)) \implies (\exists I_2 \in \mathcal{I}, N(B, I_2) \not\subseteq N(A, I_2))$

Among other results, Dechter and Pearl proved that, on consistent instances A* is 1-optimal, meaning that for any admissible *UDXBB* algorithm X, there exists a tie-breaking for A* that expands a subset of the nodes expanded by X. They also show that no family of algorithms can be 0-optimal, meaning that there is no way to set the tie-breaking strategy to guarantee a minimal number of node expansions.

Dominance Pruning

Dominance pruning is a technique that makes use of a dominance relation as an additional source of information. A relation $\preceq \ \subseteq S \times S$ is a dominance relation if, whenever $s \preceq t$, then $h^*_\epsilon(t) \leq h^*_\epsilon(s)$. We say that a node n_t prunes another n_s if $n_s \neq n_t$, $g(n_t) \leq g(n_s)$ and $s \preceq t$.

We define A* with dominance pruning (A$^*_{pr}$) as the vanilla A* algorithm with a simple modification. Anytime that a node n_s is selected for expansion, skip it if there exists another node n_t in open or closed such that n_t prunes n_s. Nodes pruned this way are removed from open but they are neither expanded nor inserted into the closed list.[1] Therefore, pruned nodes are "forgotten" and no node can be pruned due to being dominated by a previously pruned node. This is necessary to correctly handle the case where there are only two nodes that prune each other, since in that case any of the two nodes could be pruned, but at least one of them must be expanded to find a solution.

In this work, we assume that the dominance relation is provided as an instance-dependent function. In practice, it can also be automatically obtained from a model of the problem, even though in this work we assume that the model is not available to the search algorithm. A common way to define a dominance relation is based on identifying resources (Hall et al. 2013), i.e. variables for which there exists a total order for their values such that larger values enable more actions. Furthermore, there are other more advance methods that find pre-orders on arbitrary abstract state spaces (Torralba and Hoffmann 2015). In both cases, the dominance relations that have been used in the literature are:

[1] Nodes can also be pruned upon generation to avoid the overhead of computing h and open list insertion. But this does not affect the number of expanded states, which is what interests us.

- Pre-order relations: they are reflexive ($s \preceq s$ for all s) and transitive ($s \preceq t \wedge t \preceq u \implies s \preceq u$).
- Cost-simulation relations: whenever $s \preceq t$, for all $s \xrightarrow{l} s'$, either $s' \preceq t$ or $\exists t \xrightarrow{l'} t'$ s.t. $c(l') \leq c(l)$ and $s' \preceq t'$.

Even though one can define dominance relations that do not satisfy these properties, they are naturally obtained in most cases. In particular, the property of cost-simulation is related to the way automatic methods prove that the obtained relation is a dominance relation without having access to h^*.

Definition of Optimal Efficiency

Following Dechter and Pearl, we are interested in the optimal efficiency of algorithms in regards of node expansions on concrete families of instances. In this section, we generalize their framework by considering the additional information of a dominance relation. This requires defining what consistent instances are in this case, as well as defining the different notions of optimal efficiency, and the families of algorithms that we will consider.

Consistent Instances

A problem instance is a tuple $\langle \Theta, h, \preceq \rangle$, where Θ is a transition system, h is an admissible heuristic for Θ, and $\preceq$ is a dominance relation for Θ. We say that an instance is consistent if both the heuristic and dominance relation are consistent on their own, and they are consistent with each other, meaning that they fulfill the following properties.

Definition 2. *An instance $I = \langle \Theta, h, \preceq \rangle$ is consistent if:*

(i) *h is consistent.*

(ii) *$\preceq$ is a transitive cost-simulation.*

(iii) *$\preceq$ is consistent with h: $s \preceq t \implies h(t) \leq h(s)$.*

Condition (ii) ensures that the information provided by $\preceq$ is consistent in two different ways. First, $\preceq$ must be transitive, because if we do know that $s \preceq t$ and $t \preceq u$, then $h^*(u) \leq h^*(t) \leq h^*(s)$ so $s \preceq u$ can be deduced. Second, for a dominance relation to be consistent, we require it to be a cost-simulation relation so that whenever n_t prunes n_s, then if n_s or any of its successors would prune n_u, then n_t or some of its successors prune n_u as well.

Condition (iii) requires $\preceq$ and h to not contradict each other on their comparison for any two states s and t. Note that this does not render $\preceq$ uninformative, since comparing states based on their heuristic value is no substitute for dominance analysis. In particular, even if $\preceq$ always agrees with h, its role is to identify cases where the relative heuristic evaluation of both states is provably correct.

A question is whether these conditions are extremely rare or they can be expected to happen in practice. The first two conditions are indeed quite common: most heuristics that come from an optimal solution to a relaxation of the problem are indeed consistent; and typical approaches to compute dominance relations in planning are guaranteed to return transitive cost-simulation relations (Torralba and Hoffmann 2015; Torralba 2017).

An analysis of whether heuristics are consistent with respect to a dominance relation $\preceq$ is beyond the scope of this paper since that would require to consider concrete heuristic functions and dominance relations. In practice, it is reasonable to expect that most consistent heuristics will fulfill this property. For example, consider resource-based dominance relations that identify that states having more resources (fuel or money for example) are preferred. These are dominance relations because more resources can only enable more transitions in the state space; so heuristics that result from systematic (symmetric) relaxations of the problem will typically associate a lower heuristic value to states with more resources, everything else being equal. Indeed, for several families of heuristic functions in domain-independent planning, they have been shown to be consistent with symmetry equivalence relations (Shleyfman et al. 2015; Sievers et al. 2015), which are a special case of dominance relation. We conjecture that this holds as well for dominance relations based on comparing the values of sub-sets of variables, for heuristics that take into account the same subsets (e.g. we conjecture h^{max} and h^+ are consistent with dominance relations over single variables, and pattern databases are consistent with dominance relations over subsets of the pattern).

Types of Optimality

We extend the optimality criteria considered by Dechter and Pearl in several ways.

Definition 3 (#-optimally efficient)**.** *Let $N(A, I)$ be the set of expanded nodes by algorithm A on instance I. A family of algorithms $\mathcal{A}$ is #-optimally efficient over another $\mathcal{B}$ relative to an instance set $\mathcal{I}$ if for any algorithm $B \in \mathcal{B}$ and instance $I \in \mathcal{I}$, there exists $A \in \mathcal{A}$ such that $|N(A, I)| \leq |N(B, I)|$.*

This definition of #-optimality is a relaxed variant of the 1-optimality definition by Dechter and Pearl, which requires the number of expansions by A to be lower or equal than that of B, instead of requiring it to be a subset ($N(A, I) \subseteq N(B, I)$). Our criteria is slightly weaker since it only requires having an overall minimum number of expansions, which implicitly assumes that all expansions are equally time consuming. We say that 1-optimality is strict if $\mathcal{A}$ is 1-optimally efficient over $\mathcal{B}$, but $\mathcal{B}$ is not over $\mathcal{A}$. We say that #-optimality is strict if $\mathcal{A}$ is #-optimally efficient over $\mathcal{B}$, but $\mathcal{B}$ is not over $\mathcal{A}$.

We also consider when A* is optimal up to the last layer, i.e., where only nodes with an f-value lower than the optimal solution cost are taken into account. That is, we replace $N(X, I)$ by $N'(X, I)$ where $N'(X, I) = \{n \in N(X, I) \mid f(n) < F^*\}$. This is related to the notion of non-pathological instances introduced by Dechter and Pearl, which are those instances where A* does not expand any node n with $f(n) = F^*$. However, paradoxically, non-pathological instances are very unlikely to occur in practice. For that reason, on the context of A* algorithms, we prefer to directly consider optimality up to the last layer, simply ignoring the effort that A* will make in the last f-layer, which most of the times strongly depends on the tie-breaking.

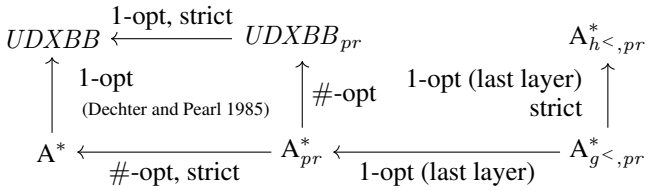


Figure 1: Summary of optimal efficiency relationships. All results assume consistent instances.

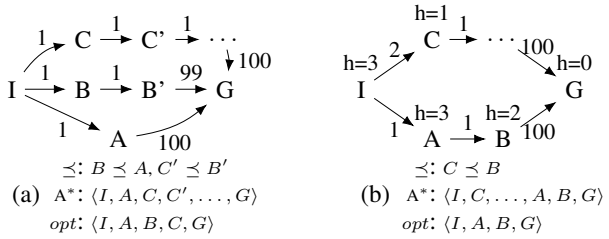


Figure 2: Counterexamples that show cases where A^*_{pr} is not optimally efficient, when pruning according to the dominance relation below each figure. The "..." region represents an arbitrarily large region of the state space that will be expanded by A^*_{pr}, but could be avoided with a different pruning or expansion order strategies. In (a) $h = 0$ for all states, in (b) each node is labeled with its h value.

Families of Algorithms

We introduce a new family of algorithms that extends $UDXBB$ with dominance pruning.

Definition 4 ($UDXBB_{pr}$)**.** *$UDXBB_{pr}$ is a family of algorithms that extends UDXBB with the ability to perform dominance pruning, i.e., to discard any node n_s if another node n_t has been generated such that n_t prunes n_s.*

Note that $UDXBB_{pr}$ algorithms cannot access the dominance relation directly or indirectly, i.e., they are not allowed to perform inference based on the fact that $h^*(t) \leq h^*(s)$ whenever $s \preceq t$. Our analysis focuses on using dominance pruning, excluding other future uses of dominance relations.

Proposition 1. *$UDXBB_{pr}$ is strictly 1-optimal over $UDXBB$ on all instances.*

Proof. 1-optimality follows trivially from the fact that $UDXBB$ is contained in $UDXBB_{pr}$, since $UDXBB_{pr}$ algorithms could choose not to prune any node if they desire so. To show this to be strict, it suffices to show an instance where there are nodes n_s, n_t with $f(n_t) \leq f(n_s) \leq F^*$ such that n_t prunes n_s. It is very easy to construct such example, e.g. see the instances in Figures 2, and 3. □

Optimal Efficiency of A^*_{pr}

Thorough the paper, we assume consistent instances, i.e., that the heuristic function and dominance relation are consistent. Figure 1 summarizes our results. Our theoretical analysis concludes that, in terms of node expansions using dominance pruning is strictly better than not using dominance. Our main result is that, on consistent instances, the expansion order of A^* is #-optimally efficient, meaning that there exist some tie-breaking of A^* that expands a minimum number of expansions. We begin by showing some counter-examples on instances that do not satisfy our consistency criteria to highlight why consistency is required. Then we discuss how to characterize the states that must be expanded to find a solution and prove it to be optimal; prove our main result; and discuss what tie-breaking strategies are more appropriate for A^* with dominance pruning.

Counter-examples due to Inconsistencies

We begin considering the two things that characterize A^*_{pr} algorithms from the set of $UDXBB_{pr}$ algorithms, and that may cause A^*_{pr} to not be optimally efficient in inconsistent instances:

1. A node is pruned whenever possible, and sometimes not pruning a node may lead to less overall expansions.
2. The expansion order of A^* may not be optimally efficient anymore when considering dominance pruning.

Figure 2 shows examples where A^*_{pr} does expand more nodes than necessary for these two reasons. The example in Figure 2a illustrates a state space and dominance relation $\preceq$ for which pruning a node whenever possible is not an optimal strategy, independently of the expansion order (for simplicity we set $h = 0$ for all states). After expanding the initial state I, one can prune node B because it is dominated by A. However, if B is pruned, B' won't be generated under any expansion order so C' and all its arbitrarily many successors will be expanded.

Our second example, illustrated in Figure 2b, shows a case where it is good to prune nodes whenever possible but the expansion order of A^* leads to a sub-optimal number of expansions. The optimal expansion order is $\langle I, A, B, G\rangle$. C does not need to be expanded even though $f(C) < F^*$ because C will be pruned ($C \preceq B$ and $g(B) \leq g(C)$). However, A^*_{pr} will expand C after expanding the initial state I, since $f(C) < f(A)$ and B has not been generated yet.

All these scenarios can be attributed to "inconsistencies" within the dominance relation $\preceq$ or between $\preceq$ and the heuristic function h. In Figure 2b the dominance relation and heuristic do not agree on the comparison between B and C. The dominance relation proves that B is at least as close to the goal as C, but the heuristic function estimates that C is closer to the goal. In the case of Figure 2a, the dominance relation is inconsistent because the information that A is closer to the goal than B is lost after one expansion and neither A nor any of its successors could be used to prune B' or C'.

Solution Sets

We first identify which states need to be expanded to prove optimality by any search that does not have access to any additional information, other than a heuristic function h and the ability to prune nodes. Traditionally, this is done by identifying *must-expand* states that must be expanded for every algorithm to prove optimality, or *must-expand* pairs as done

in the bidirectional search setting (Eckerle et al. 2017). However, in our case there are many choices that can be made for dominance pruning, so now the difference between must-expand nodes and the nodes that belong to any concrete solution is not restricted to the last f-layer.

We define instead solution sets, which take into consideration all nodes that must be expanded by any $UDXBB_{pr}$ algorithm to find a solution, including the last f-layer. Let $\mathcal{S}$ be a set of nodes. We use $[\mathcal{S}]$ to denote the set extended with its immediate successors, i.e., $[\mathcal{S}] = \mathcal{S} \cup \{n_{s'} \mid n_s \rightarrow N_{s'}, n_s \in \mathcal{S}\}$. The intuition is that, if $\mathcal{S}$ is the set of nodes that have been expanded at some point during the execution of a $UDXBB$ algorithm, then $[\mathcal{S}]$ is the set of nodes that have been generated. In other words, if $\mathcal{S}$ represents the contents of the closed list, then $[\mathcal{S}] \setminus \mathcal{S}$ contains the set of nodes in the open list and all pruned nodes.

Definition 5 ($UDXBB_{pr}$ Solution Set)**.** *A set of nodes $\mathcal{S}$ is a $UDXBB_{pr}$ solution set for an instance I if:*

(a) $\forall n_s \in \mathcal{S} \setminus \{n_{s^I}\}, \exists n_t \in \mathcal{S}, n_t \xrightarrow{l} n_s$.

(b) $\exists n_s \in [\mathcal{S}], s \in S^G$ *and* $g(n_s) = F^*$,

(c) $\forall n_s \in [\mathcal{S}] \setminus \mathcal{S}, f(n_s) \geq F^*$ *or* $\exists n_t \in \mathcal{S}, n_t$ *prunes* n_s.

Condition (a) requires that every expanded node in $\mathcal{S}$ was generated by expanding one of its parents. Condition (b) requires that an optimal solution was found. Condition (c) ensures that the solution found is proven to be optimal, because all nodes in the open list after expanding $\mathcal{S}$ have a large enough f-value or are pruned by dominance.

Theorem 1. *Let I be any admissible instance. Then, expanding a solution set is a necessary and sufficient condition for admissible $UDXBB_{pr}$ algorithms, i.e., for any A in $UDXBB_{pr}$, $N(A, I)$ is a solution set.*

Proof Sketch. If (a), (b), and (c) hold, then an optimal solution has been found due to (a) and (b), and the stopping condition holds, since all states remaining in the open list have an f-value greater or equal to the incumbent solution.

If (a) does not hold, then a state has been expanded without being in open, which is impossible in any $UDXBB$ algorithm. If (b) does not hold, then an optimal solution has not been found. If (c) does not hold, then there exists some s in the open list that may lead to a solution cost lower than F^*, so the solution was not proven to be optimal. □

We remark that the proof above relies on the fact that $UDXBB_{pr}$ algorithms are not allowed to use dominance relations for anything except dominance pruning. Otherwise, the criteria (c) of a solution set could be made weaker, increasing the set of possible solution sets.

$\mathbf{A}^*_{pr}$ is Optimally Efficient on Consistent Instances

Before proving our main result of this section, we analyze some properties that hold for consistent instances. An important one is that, whenever h and $\preceq$ are consistent with each other, nodes with larger f-value cannot prune nodes with lower f-value.

Lemma 1. *Let I be a consistent instance. Let n_s, n_t be any two nodes such that n_t prunes n_s. Then, $f(n_t) \leq f(n_s)$.*

Proof. Since n_t prunes n_s, it holds that $g(n_t) \leq g(n_s)$ and $s \preceq t$. By consistency, $h(t) \leq h(s)$, so $f(n_t) \leq f(n_s)$. □

Next, we show that pruning is transitive.

Lemma 2. *If $\preceq$ is transitive, n_u prunes n_t and n_t prunes n_s, then n_u prunes n_s.*

Proof. By the assumption it follows that $g(n_u) \leq g(n_t) \leq g(n_s)$, and $s \preceq t \preceq u$. Therefore, $g(n_u) \leq g(n_s)$ and, by transitivity of $\preceq$, $s \preceq u$. So n_u prunes n_s. □

Next, we show that all states in the smallest solution set must be expanded only with its optimal g-value.

Lemma 3. *Let I be a consistent instance. Then, there exists a solution set $\mathcal{S}$ for I of minimum size such that for all $n_s \in \mathcal{S}$, $g(n_s) = g^*(s)$.*

Proof. Assume the contrary. Then, some n_s has been expanded with a sub-optimal value, $g^*(s) < g(n_s)$. Therefore, a predecessor along the optimal path from s^I to s has not been expanded. Let n_t be the first such predecessor. By consistency of h, we know that f-values monotonically increase along a path, so $f(n_t) \leq f^*(n_s) < f(n_s)$. As $n_t \notin \mathcal{S}$, by condition (c) of a solution set, n_t was pruned, i.e., $\exists n_u \in \mathcal{S}$ s.t. n_u prunes n_t. As $\preceq$ is a cost-simulation, then n_u has some successor that would prune n_s, so there must be a node in $\mathcal{S}$ that prunes n_s. Therefore, $\mathcal{S} \setminus \{n_s\}$ is also a solution set, contradicting the fact that $\mathcal{S}$ is of minimum size. □

We next show that pruning a node whenever possible is an optimally efficient strategy because there exists a solution set $\mathcal{S}$ of minimum size that does not contain any node that can be pruned by another node in $[\mathcal{S}]$, unless both nodes prune each other. To show this, we consider Algorithm 1.

Algorithm 1: Replace

Input: $\mathcal{S}_0$,$n_s \in \mathcal{S}_0$, $n_t \in [\mathcal{S}_0]$ where $\mathcal{S}_0$ is a solution set and n_t prunes n_s
Output: Solution set $\mathcal{S}_i$ that does not contain n_s

```
S_1 := (S_0 ∪ {n_t}) \ {n_s};
i = 1;
while ∃ n_{s^i} ∈ S_i, ∄ n_{u^i} ∈ S_i, n_{u^i} --l--> n_{s^i} do
    Choose such an n_{s^i} with minimum g-value ;
    Choose n_{t^i} in [S_i] such that n_{t^i} prunes n_{s^i} ;
    S_{i+1} := S_i ∪ {n_{t^i}} \ {n_{s^i}} ;
    i = i + 1 ;
return S_i ;
```

Lemma 4. *Let $\mathcal{S}_0$ be a solution set for a consistent instance. Let $n_s \in \mathcal{S}_0$, $n_t \in [\mathcal{S}_0]$ such that n_t prunes n_s. Then, Algorithm 1 returns a solution set $\mathcal{S}_k$ such that: $|\mathcal{S}_k| \leq |\mathcal{S}_0|$; $n_s \notin \mathcal{S}_k$; $n_t \in \mathcal{S}_k$; and If $n_t \in \mathcal{S}_0$ then $|\mathcal{S}_k| < |\mathcal{S}_0|$.*

Proof Sketch. The size of the solution set cannot increase during the execution of Algorithm 1, i.e., $|\mathcal{S}_{i+1}| \leq |\mathcal{S}_i|$ because a node is removed at each iteration and at most one node is added. If $n_t \in \mathcal{S}_0$ then $|\mathcal{S}_1| = |\mathcal{S}_0| - 1$, since

n_s was removed and no node was added, so in that case $|\mathcal{S}_k| \leq |\mathcal{S}_1| < |\mathcal{S}_0|$.

Properties (b) and (c) of a solution set are preserved by all intermediate $\mathcal{S}_i$ because n_{s^i} is replaced by n_{t^i} such that n_{t^i} prunes n_{s^i}, so by Lemma 1 and 2, n_{t^i} can do anything n_{s^i} could. Property (a) holds when the algorithm terminates, since it is the stopping condition for the loop. The algorithm always terminates because all nodes n_{s^i} removed in the loop are descendants of n_s which were present in $\mathcal{S}_0$, and there are only finitely many.

Finally, it remains to be proven that, at every iteration in line 4, there exists some n_{t^i} in $[\mathcal{S}_i]$ such that n_{t^i} prunes n_{s^i}. Note that n_{s^i} is a descendant of n_s that has no parent in $\mathcal{S}_i$. Since all nodes in $\mathcal{S}_0$ have a parent, and all n_{t^i} added along the way too, this means that the parent of n_{s^i} was some n_{s^j} removed in a previous iteration $j < i$, being replaced by n_{t^j}. Since $\preceq$ is a cost-simulation relation, n_{t^j} must have a successor n_{t^i} that prunes n_{s^i}. □

Lemma 5. *Let $\mathcal{S}$ be a solution set of minimum size for a consistent instance. Then there do not exist n_s, n_t in $\mathcal{S}$ such that n_t prunes n_s.*

Proof. Assume that n_t prunes n_s. By Lemma 4, using the procedure above, we can construct another solution set $\mathcal{S}'$ such that $|\mathcal{S}'| < |\mathcal{S}|$, contradicting that $\mathcal{S}$ has minimum size. □

Lemma 6. *Let I be a consistent instance. Then, there exists a solution set $\mathcal{S}$ of minimum size for I such that there does not exist any $n_s \in \mathcal{S}$ and $n_t \in [\mathcal{S}]$ such that n_t prunes n_s and n_s does not prune n_t.*

Proof Sketch. Assume the opposite, let $\mathcal{S}$ be a solution set such that there exist $n_s \in \mathcal{S}$ and $n_t \in [\mathcal{S}]$ where n_s prunes n_t and n_t does not prune n_s. By Lemma 5 $n_t \notin \mathcal{S}$. By condition (c) of a solution set we know that either $f(n_t) \geq F^*$ or there exists $n_u \in \mathcal{S}$ such that n_u prunes n_t.

Case 1: There exists $n_u \in \mathcal{S}$ such that n_u prunes n_t. By transitivity, n_u prunes n_s, so one can construct a minimal solution set with Lemma 4 of smaller size, contradicting that $\mathcal{S}$ is a solution set of minimal size.

Case 2: $f(n_t) \geq F^*$. Then, by Lemma 1, $f(n_s) \geq f(n_t) \geq F^*$. If $f(n_t) > F^*$, we can remove n_s and all its descendants from $\mathcal{S}_0$ to obtain a smaller solution set, contradicting the fact that it is a solution set of minimal size. Therefore, $f(n_s) = f(n_t) = F^*$. Note that a solution set of minimum size only contains a node with $f(n_s) = F^*$ when n_s is on the solution path returned by the algorithm. This path can be replaced by another of the same length and cost that goes through n_t by repeatedly calling Algorithm 1. □

Now we are ready to prove our main result.

Theorem 2. *A^*_{pr} is #-optimal on consistent instances over $UDXBB_{pr}$.*

Proof. We show that there exists a solution set $\mathcal{S}$ of minimum size for which there exists a tie-breaking strategy under which A^*_{pr} with h and $\preceq$ expands exactly $\mathcal{S}$. By Lemma 6, we choose $\mathcal{S}$ so that there does not exist any $n_s \in \mathcal{S}$ and $n_t \in [\mathcal{S}]$ s.t. n_t prunes n_s and n_s does not prune n_t. Assume a tie-breaking that prefers expanding nodes in $\mathcal{S}$ over any other node, and prefers pruning nodes not in $\mathcal{S}$. Formally, our tie-breaking strategy selects for expansion any node not in $\mathcal{S}$ such that it can be pruned. If no such node exists, it selects a node (with minimal f value) from $\mathcal{S}$ that cannot be pruned. This tie-breaking always succeeds because otherwise, the open list does not contain any node with minimal f value that is outside $\mathcal{S}$ and can be pruned or that it is in $\mathcal{S}$ and cannot be pruned. Then, the node selected for expansion either: (A) it is in $\mathcal{S}$ but can be pruned due to some node in open or closed; (B) it is not in $\mathcal{S}$ and cannot be pruned.

Case (A). There exists n_t that prunes some $n_s \in \mathcal{S}$. By Lemma 5, we know that $n_t \notin \mathcal{S}$. As n_t is in the open list after having expanded a subset of $\mathcal{S}$, $n_t \in [\mathcal{S}]$ and, by our choice of solution set with Lemma 6, n_s prunes n_t. By Lemma 1, $f(n_t) \leq f(n_s)$, so with our tie-breaking strategy A^* would have selected n_t instead, reaching a contradiction.

Case (B). Let n_s be a node that is expanded by A^*_{pr} but it is not in $\mathcal{S}$. If $f(n_s) = F^*$, then a node along the optimal solution contained in $\mathcal{S}$ should have been chosen instead. If $f(n_s) < F^*$, by condition (c) of a solution set, there exists $n_t \in \mathcal{S}$ such that n_t prunes n_s. Again, if n_t is in open or closed, n_s will be pruned reaching a contradiction. Otherwise, there must be an ancestor along the path from s^I to n_t in open with its optimal g-value. Such an $n_u \in \mathcal{S}$, must have $f(n_u) \leq f(n_t) \leq f(n_s)$, so according to our tie-breaking n_u would have been chosen for expansion instead of n_s (n_u cannot be pruned by the same argument as in case (A)). □

Corollary 1. *A^*_{pr} is strictly #-optimal over A^* on consistent instances.*

Proof. This follows directly from the fact that A^*_{pr} is #-optimal over $UDXBB_{pr}$ and $UDXBB_{pr}$ is strictly 1-optimal over the family of $UDXBB$ algorithms, which contains all algorithms in the family of A^* algorithms. □

Optimal Tie-Breaking Strategies

For A^* with consistent heuristics the tie-breaking strategy is only relevant in the last f-layer. Ideally, once the minimum f-value in the open list is equal to F^*, only nodes on a path to the goal will be selected for expansion. Practical implementations often prefer expanding nodes with lowest h-value, aiming to reduce the effort in the last layer. In domain-independent planning, where a factored model of the state space is available to offer additional information to the algorithm, some other strategies have been suggested, like using (possibly inadmissible) heuristic functions that estimate plan length instead of plan cost (Asai and Fukunaga 2017; Corrêa, Pereira, and Ritt 2018). They showed that tie-breaking can be quite significant for the overall performance, specially in domains with 0-cost actions.

A^*_{pr}, however, is more sensitive to the choice of tie-breaking strategy, since it may matter along previous layers. This brings up the question of what is a good tie-breaking strategy for A^*_{pr}. We define $\text{A}^*_{g^<,pr}$ as A^*_{pr} breaking ties in favor of states with minimum g-value.

Theorem 3. *$A^*_{g^<,pr}$ is 1-optimal efficient up to the last layer over A^*_{pr} on consistent instances.*

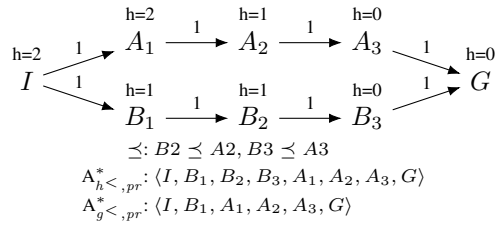


Figure 3: Counter-example for the 1-optimal efficiency of $A^*_{h^<,pr}$ up to the last layer on consistent instances.

Proof Sketch. Let $\mathcal{S}$ to be a solution set for A^*_{pr}, and let $\mathcal{S}'$ be the subset of nodes in solution set up to the last layer, $\mathcal{S}' = \{n_s \in \mathcal{S} \mid f(n_s) < F^*\}$. We show that there is an expansion order compatible with $A^*_{g^<,pr}$ that expands all nodes in $\mathcal{S}'$ before expanding any other node. For this, the same proof from Theorem 2 applies up to case (B). For case (B), we know that $f(n_s) < F^*$ and, by the same argument as in the proof of Theorem 2, some $n_u \in \mathcal{S}$ must remain in the open list with $f(n_u) \leq f(n_t) \leq f(n_s)$. At this point the tie-breaking matters since whenever $f(n_u) = f(n_s)$, the tie-breaking policy should allow selecting n_u over n_s. Since n_u is an ancestor of n_t, $g(n_u) \leq g(n_t)$, and since n_t prunes n_s, $g(n_u) \leq g(n_t) \leq g(n_s)$. Then, expanding n_u instead of n_s is still valid according to the $g^<$ tie-breaking strategy. □

However, the same is not true for every tie-breaking strategy for A^*. For example, let $A^*_{h^<,pr}$ be the family of A^*_{pr} algorithms with a tie-breaking strategy that always prefers a state with minimum h-value. As argued above this is the tie-breaking preferred by most implementations of A^* without dominance pruning, but it cannot guarantee anymore that the number of expansions up to the last layer will be minimal.

Theorem 4. *$A^*_{h^<,pr}$ is not optimally efficient up to the last layer on consistent instances.*

Proof Sketch. Figure 3 shows a counter-example of a consistent instance where all tie-breaking strategies compatible with $A^*_{h^<,pr}$ expand a node that $A^*_{g^<,pr}$ would not expand. After expanding I and B_1, the open list contains two nodes: B_2 and A_1, both with an f-value of 3. At this point, A_2 has not been generated yet so B_2 cannot be pruned. However, $A^*_{h^<,pr}$ will expand B_2 and B_3 (and in general the entire plateau of states with $f = 3$ underneath B_2), before expanding A_1. Note that this happens for nodes with $f = 3 < 4 = F^*$, i.e. nodes before the last f-layer. □

Corollary 2. *$A^*_{g^<,pr}$ is strictly 1-optimally efficient up to the last layer over $A^*_{h^<,pr}$ on consistent instances.*

Proof Sketch. 1-optimality follows directly from Theorem 3, since $A^*_{h^<,pr}$ is contained in A^*_{pr}. The fact that optimality is strict follows from Theorem 4. □

Thus, there are two conflicting objectives. Up to the last layer, it is provably beneficial to break ties in favor of lower g-value. On the last layer, empirical analysis show that it is better to break ties in favor of lower h-value. Which one should be given priority depends on the particular domain, dominance relation and heuristic. Our preliminar experiments show that in common planning domains, it is often beneficial to break ties in favor of lower h-value even when dominance pruning is used.

Conclusions

We analyzed the optimal efficiency of A^* with dominance pruning, A^*_{pr}. Assuming a consistent heuristic is not sufficient, because there may be inconsistencies in the dominance relation as well, which may cause A^*_{pr} to perform unnecessary expansions. We defined a new criterion of consistency for heuristic and dominance relation pairs, which ensures that A^*_{pr} will be optimally efficient in terms of the number of expanded nodes. We also show that tie-breaking in favor of nodes with lower g value is provably preferable to minimize the number of expansions up to the last layer. This contrasts with common strategies, which favor nodes with lowest h-value to minimize expansions in the last layer.

As in the optimal efficiency result for A^*, our analysis is based only on the number of state expansions and it ignores the actual runtime. There are of course other algorithms which may outperform A^* according to different performance measures. For example, the IDA* algorithm (Korf 1985) and other extensions like Budgeted Tree Search (Helmert et al. 2019; Sturtevant and Helmert 2020) outperform A^* in terms of memory usage. EPEA* (Goldenberg et al. 2014) aims to minimize the number of nodes generated, which is arguably more relevant to runtime than expanded nodes, but it requires additional domain-specific knowledge. Finally, other algorithms may outperform A^* in terms of runtime, e.g., when the benefits of reducing the number of node expansions does not compensate the overhead of computing the heuristic or performing pruning, which may require a quadratic cost in the number of generated states in the worst case. Nevertheless, for concrete problems and/or dominance relations it may be possible to perform the pruning more efficiently (e.g., dividing states in classes so that each state needs to be compared only against a small subset of alternatives), and one could extend rational algorithms that reason about when it is worth to compute the heuristic (Barley, Franco, and Riddle 2014; Karpas et al. 2018) to consider dominance as well.

Finally, in this work we extended the basic framework with the ability of dominance pruning using a dominance relation, but it could also be extended in other ways. For example, if backward search is possible, there are a variety of bidirectional heuristic search algorithms that can outperform A^* in terms of node expansions (Eckerle et al. 2017; Chen et al. 2017). One could consider several extensions of this paradigm regarding different forms of dominance, e.g., introducing variants that make use of more general forms of dominance (Torralba 2017), or alternative methods to exploit this information. This may open new avenues of research on how to use dominance relations beyond dominance pruning in order to make the most of them.

Acknowledgments

Álvaro Torralba was employed by Saarland University and the CISPA Helmholtz Center for Information Security during part of the development of this paper. I would like to thank Nathan Stutervant for his insights on DXBB algorithms. Thanks to the anonymous reviewers at SoCS'20 and HSDIP'20 as well as to the Basel reading group for their comments that helped me to improve the paper.

References

Asai, M.; and Fukunaga, A. 2017. Tie-Breaking Strategies for Cost-Optimal Best First Search. *Journal of Artificial Intelligence Research* 58: 67–121.

Barley, M. W.; Franco, S.; and Riddle, P. J. 2014. Overcoming the Utility Problem in Heuristic Generation: Why Time Matters. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.

Chen, J.; Holte, R. C.; Zilles, S.; and Sturtevant, N. R. 2017. Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 489–495. AAAI Press/IJCAI.

Corrêa, A. B.; Pereira, A. G.; and Ritt, M. 2018. Analyzing Tie-Breaking Strategies for the A* Algorithm. In Lang, J., ed., *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, 4715–4721. ijcai.org.

Dechter, R.; and Pearl, J. 1985. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the Association for Computing Machinery* 32(3): 505–536.

Eckerle, J.; Chen, J.; Sturtevant, N. R.; Zilles, S.; and Holte, R. C. 2017. Sufficient Conditions for Node Expansion in Bidirectional Heuristic Search. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, 79–87. AAAI Press.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N. R.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research* 50: 141–187.

Hall, D.; Cohen, A.; Burkett, D.; and Klein, D. 2013. Faster Optimal Planning with Partial-Order Pruning. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*. Rome, Italy: AAAI Press.

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.

Helmert, M.; Lattimore, T.; Lelis, L. H. S.; Orseau, L.; and Sturtevant, N. R. 2019. Iterative Budgeted Exponential Search. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, 1249–1257. ijcai.org.

Karpas, E.; Betzalel, O.; Shimony, S. E.; Tolpin, D.; and Felner, A. 2018. Rational deployment of multiple heuristics in optimal state-space search. *Artificial Intelligence* 256: 181–210.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 27(1): 97–109.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and Symmetries in Classical Planning. In Bonet, B.; and Koenig, S., eds., *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, 3371–3377. AAAI Press.

Sievers, S.; Wehrle, M.; Helmert, M.; Shleyfman, A.; and Katz, M. 2015. Factored Symmetries for Merge-and-Shrink Abstractions. In Bonet, B.; and Koenig, S., eds., *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, 3378–3385. AAAI Press.

Sturtevant, N.; and Helmert, M. 2020. A Guide to Budgeted Tree Search. In Harabor, D.; and Vallati, M., eds., *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS'20*, 75–81. AAAI Press. URL https://www.aaai.org/Library/SOCS/socs20contents.php.

Torralba, Á. 2017. From Qualitative to Quantitative Dominance Pruning for Optimal Planning. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4426–4432. AAAI Press/IJCAI.

Torralba, Á.; and Hoffmann, J. 2015. Simulation-Based Admissible Dominance Pruning. In Yang, Q., ed., *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, 1689–1695. AAAI Press/IJCAI.

Approximate bi-criteria search by efficient representation of subsets of the Pareto-optimal frontier

Oren Salzman[†]
[†]Technion, Israel Institute of Technology
Haifa 32000, Israel
salzman@cs.technion.ac.il

Abstract

We consider the bi-criteria shortest-path problem where we want to compute shortest paths on a graph that simultaneously balance two cost functions. While this problem has numerous applications, there is usually no path minimizing both cost functions simultaneously. Thus, we typically consider the set of paths where no path is strictly better than the others in both cost functions, a set called the Pareto-optimal frontier. Unfortunately, the size of this set may be exponential in the number of graph vertices and the general problem is NP-hard. While existing schemes to approximate this set exist, they may be slower than exact approaches when applied to relatively small instances and running them on graphs with even a moderate number of nodes is often impractical. The crux of the problem lies in how to efficiently approximate the Pareto-optimal frontier. Our key insight is that the Pareto-optimal frontier can be approximated using *pairs* of paths. This simple observation allows us to run a best-first-search while efficiently and effectively pruning away intermediate solutions in order to obtain an approximation of the Pareto frontier for any given approximation factor. We compared our approach with an adaptation of BOA*, the state-of-the-art algorithm for computing exact solutions to the bi-criteria shortest-path problem. Our experiments show that as the problem becomes harder, the speedup obtained becomes more pronounced. Specifically, on large roadmaps, when using an approximation factor of 10% we obtain a speedup on the average running time of more than $\times 19$.

1 Introduction & Related Work

We consider the bi-criteria shortest-path problem, an extension to the classical (single-criteria) shortest-path problem where we are given a graph $G = (V, E)$ and each edge has two cost functions. Here, we are required to compute paths that balance between the two cost functions. The well-studied problem (Chinchuluun and Pardalos 2007) has numerous applications. For example, given a road network, the two cost functions can represent travel times and distances and we may need to consider the set of paths that allow to balance between these costs. Other applications include planning of power-transmission lines (Bachmann et al. 2018) and planning how to transport hazardous material in order to balance between minimizing the travel distance and the risk of exposure for residents (Bronfman et al. 2015).

There usually is no path minimizing all cost functions simultaneously. Thus, we typically consider the set of paths where no path is strictly better then the others for both cost functions, a set called the *Pareto-optimal frontier*. Unfortunately, the problem is NP-hard (Serafini 1987) as the cardinality of the size of the Pareto-optimal frontier may be exponential in $|V|$ (Ehrgott 2005; Breugem, Dollevoet, and van den Heuvel 2017) and even determining whether a path belongs to the Pareto-optimal frontier is NP-hard (Papadimitriou and Yannakakis 2000).

Existing methods either try to efficiently compute the Pareto-optimal frontier or to relax the problem and only compute an approximation of this set.

Efficient computation of the Pareto-optimal frontier. To efficiently compute the Pareto-optimal frontier, adaptations of the celebrated A* algorithm (Hart, Nilsson, and Raphael 1968) were suggested. Stewart et al. (1991) introduced Multi-Objective A* (MOA*) which is a multiobjective extension of A*. The most notable difference between MOA* and A* is in maintaining the Pareto-optimal frontier to intermediate vertices. This requires to check if a path π is *dominated* by another path $\tilde{\pi}$. Namely, if both of $\tilde{\pi}$'s costs are smaller than π's costs. As these dominance checks are repeatedly performed, the time complexity of the checks play a crucial role for the efficiency of such bi-criteria shortest-path algorithms. MOA* was later revised (Mandow and De La Cruz 2005; Mandow and De La Cruz 2010; Pulido, Mandow, and Pérez-de-la Cruz 2015) with the most efficient variation, termed bi-Objective A* (BOA*) (Ulloay et al. 2020) allowing to compute these operations in $O(1)$ time when a consistent heuristic is used.[1]

Approximating the Pareto-optimal frontier. Initial methods in computing an approximation of the Pareto-optimal frontier were directed towards devising a Fully Polynomial Time Approximation Scheme[2]

[1] A heuristic function is said to be consistent if its estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour.

[2] An FPTAS is an approximation scheme whose time complexity is polynomial in the input size and also polynomial in $1/\varepsilon$

(FPTAS) (Vazirani 2001). Warburton (1987) proposed a method for finding an approximate Pareto optimal solution to the problem for any degree of accuracy using scaling and rounding techniques. Perny and Spanjaard (2008) presented another FPTAS given that a finite upper bound L on the numbers of arcs of all solution-paths in the Pareto-frontier is known. This requirement was later relaxed (Tsaggouris and Zaroliagis 2009; Breugem, Dollevoet, and van den Heuvel 2017) by partitioning the space of solutions into cells according to the approximation factor and, roughly speaking, taking only one solution in each grid cell. Unfortunately, the running times of FPTASs are typically polynomials of high degree, and hence they may be slower than exact approaches when applied to relatively-small instances and running them on graphs with even a moderate number of nodes (e.g., $\approx 10,000$) is often impractical (Breugem, Dollevoet, and van den Heuvel 2017).

A different approach to compute a subset of the Pareto-optimal solution is to find all extreme supported non-dominated points (i.e., the extreme points on the convex hull of the Pareto-optimal set) (Sedeno-Noda and Raith 2015). Taking a different approach Legriel et al. (2010) suggest a method based on satisfiability/constraint solvers. Alternatively, a simple variation of MOA*, termed MOA*$_\varepsilon$ allows to compute an approximation of the Pareto-optimal frontier by prunning intermediate paths that are approximately dominated by already-computed solutions (Perny and Spanjaard 2008). However, as we will see, this allows to prune only a small subset of paths that may be pruned.

Finally, recent work (Bökler and Chimani 2020) conducts a comprehensive computational study with an emphasis on multiple criteria. Similar to the aforementioned FPTASs, their framework still partitions the space prior to running the algorithm.

Key contribution. To summarize, exact methods compute a solution set whose size is often exponential in the size of the input. While one would expect that approximation algorithms will allow to dramatically speed up computation times, in practice their running times are often slower than exact solutions for FPTAS's because they partition the space of solution into cells according to the approximation factor in advance. Alternative methods only prune paths that are approximately dominated by already-computed solutions.

Our key insight is that we can efficiently partition the space of solution into cells during the algorithm's execution (and not a-priori). This allows us to efficiently and effectively prune away intermediate solutions in order to obtain an approximation of the Pareto frontier for any given approximation factor ε (this will be formalized in Sec. 2). This is achieved by running a best-first search on *path pairs* and not individual paths. Such path pairs represent a subset of the Pareto frontier such that any solution in this subset is approximately dominated by the two paths. Using concepts that draw inspiration from a recent search algorithm from the robotics literature (Fu et al. 2019), we propose Path-Pair A* (PP-A*). PP-A* dramatically reduces the computational complexity of the best-first search by merging path pairs while still ensuring that an approximation of the Pareto-optimal frontier is obtained for any desired approximation.

For example, on a roadmap of roughly 1.5 million vertices, PP-A* approximates the Pareto optimal frontier within a factor of 1% in roughly 13 seconds on average on a commodity laptop. We compared our approach with an adaptation of BOA* (Ulloay et al. 2020), the state-of-the-art algorithm for computing exact solutions to the bi-criteria shortest-path problem, which we term BOA*$_\varepsilon$. BOA*$_\varepsilon$ computes near optimal solutions by using the approach suggested in (Perny and Spanjaard 2008). Our experiments show that as the problem becomes harder, the speedup that PP-A* may offer becomes more pronounced. Specifically, on the aforementioned roadmap and using an approximation factor of 10%, we obtain a speedup on the average running time of more than $\times 19$ and a maximal speedup of over $\times 25$.

2 Problem definition

Let $G = (V, E)$ be a graph, $c_1 : E \to \mathbb{R}$ and $c_2 : E \to \mathbb{R}$ be two cost functions defined over the graph edges. A path $\pi = v_1, \dots v_k$ is a sequence of vertices where consecutive vertices are connected by an edge. We extend the two cost functions to paths as follows:

$$c_1(\pi) = \sum_{i=1}^{k-1} c_1(v_i, v_{i+1}) \quad \text{and} \quad c_2(\pi) = \sum_{i=1}^{k-1} c_2(v_i, v_{i+1}).$$

Unless stated otherwise, all paths start at the same specific vertex v_{start} and π_u will denote a path to vertex u.

Definition 1 (Dominance)**.** *d We say that π_u strictly dominates $\tilde{\pi}_u$ if (i) π_u weakly dominates $\tilde{\pi}_u$ and (ii) $c_1(\pi_u) < c_1(\tilde{\pi}_u)$ or $c_2(\pi_u) < c_2(\tilde{\pi}_u)$.*

Definition 2 (Approximate dominance)**.** *Let π_u and $\tilde{\pi}_u$ be two paths to vertex u and let $\varepsilon_1 \geq 0$ and $\varepsilon_2 \geq 0$ be two real values. We say that π_u $(\varepsilon_1, \varepsilon_2)$-dominates $\tilde{\pi}_u$ if (i) $c_1(\pi_u) \leq (1+\varepsilon_1) \cdot c_1(\tilde{\pi}_u)$ and (ii) $c_2(\pi_u) \leq (1+\varepsilon_2) \cdot c_2(\tilde{\pi}_u)$. When $\varepsilon_1 = \varepsilon_2$, we will sometimes say that π_u (ε_1)-dominates $\tilde{\pi}_u$ and call ε_1 the approximation factor.*

Definition 3 ((approximate) Pareto-optimal frontier)**.** *The Pareto-optimal frontier Π_u of a vertex u is a set of paths connecting v_{start} and u such that (i) no path in Π_u is strictly dominated by any other path from v_{start} to u and (ii) every path from v_{start} to u is weakly dominated by a path in Π_u. Similarly, for $\varepsilon_1 \geq 0$ and $\varepsilon_2 \geq 0$ the approximate Pareto optimal frontier[3] $\Pi_u(\varepsilon_1, \varepsilon_2) \subseteq \Pi_u$ is a subset of u's Pareto frontier such that every path in Π_u is $(\varepsilon_1, \varepsilon_2)$-dominated by a path in $\Pi_u(\varepsilon_1, \varepsilon_2)$.*

For brevity we will use the terms (approximate) Pareto frontier to refer to the (approximate) Pareto optimal frontier. For a visualization of these notions, see Fig. 1.

We are now ready to formally define our search problems.

where ε is the approximation factor.

[3]Our definition of an approximate Pareto optimal frontier slightly differs from existing definitions (Breugem, Dollevoet, and van den Heuvel 2017) which do not require that the approximate Pareto frontier is a subset of the Pareto-optimal frontier.

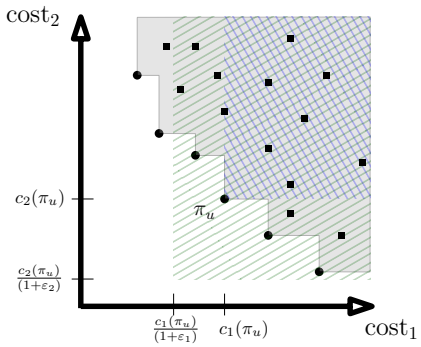


Figure 1: Dominance, approximate dominance and Pareto frontier. Given start and target vertices, we consider each path π_u as a 2D point $(c_1(\pi_u), c_2(\pi_u))$ according to the two cost functions (points and squares). The set of all possible paths dominated and approximately dominated by path π_u are depicted in blue and green, respectively. The Pareto frontier Π_u is the set of all black points that collectively dominate all other possible paths (squares in grey region).

Problem 1 (Bi-criteria shortest path)**.** *Let G be a graph, $c_1, c_2 : E \to \mathbb{R}$ two cost functions and v_{start} and v_{goal} be start and goal vertices, respectively. The* bi-criteria shortest path *problem calls for computing the Pareto frontier $\Pi_{v_{\text{goal}}}$.*

Problem 2 (Bi-criteria approximate shortest path)**.** *Let G be a graph, $c_1, c_2 : E \to \mathbb{R}$ two cost functions and v_{start} and v_{goal} be start and goal vertices, respectively. Given $\varepsilon_1 \geq 0$ and $\varepsilon_2 \geq 0$, the* bi-criteria approximate shortest path *problem calls for computing an approximate Pareto frontier $\Pi_{v_{\text{goal}}}(\varepsilon_1, \varepsilon_2)$.*

3 Algorithmic Background

In this section we describe two approaches to solve the bi-criteria shortest-path problem (Problem 1). With the risk of being tedious, we start with a brief review of best-first search algorithms as both state-of-the-art bi-criteria shortest path algorithms, as well as ours, rely heavily on this algorithmic framework. We note that the description of best-first search we present here can be optimized but this version will allow us to better explain the more advanced algorithms.

A best-first search algorithm (Alg. 1) computes a shortest path from v_{start} to v_{goal} by maintaining a priority queue, called an OPEN list, that contains all the nodes that have not been expanded yet (line 1). Each node is associated with a path π_u from v_{start} to some vertex $u \in V$ (by a slight abuse of notation we will use paths and nodes interchangeability which will simplify algorithm's descriptions in the next sections). This queue is ordered according to some cost function called the f-value of the node. For example, in Dijkstra and A*, this is the computed cost from v_{start} (also called its g-value) and the computed cost from v_{start} added to the heuristic estimate to reach v_{goal}, respectively.

At each iteration (lines 3-13), the algorithm extracts the most-promising node from OPEN (line 3), checks if it has the potential to be a better solution than any found so far

Algorithm 1 Best First Search

Input: $(G = (V, E), v_{\text{start}}, v_{\text{goal}}, \ldots)$

1: OPEN $\leftarrow$ new node $\pi_{v_{\text{start}}}$
2: **while** OPEN $\neq \emptyset$ **do**
3: $\quad \pi_u \leftarrow$ OPEN.**extract_min**()
4: $\quad$ **if** **is_dominated**(π_u) **then**
5: $\quad\quad$ **continue**
6: $\quad$ **if** $u = v_{\text{goal}}$ **then** ▷ reached goal
7: $\quad\quad$ **merge_to_solutions**(π_u, solutions)
8: $\quad\quad$ **continue**
9: $\quad$ **for** $e = (u, v) \in$ neighbors(u, G) **do**
10: $\quad\quad \pi_v \leftarrow$ **extend**(π_u, e)
11: $\quad\quad$ **if** **is_dominated**(π_v) **then**
12: $\quad\quad\quad$ **continue**
13: $\quad\quad$ **insert**(π_v, OPEN)
14: **return** all extreme paths in solutions

(line 4). If this is the case and we reached v_{goal}, the solution set is updated (in single-criteria shortest path, once a solution is found, the search can be terminated). If not, we extend the path represented by this node to each of it's neighbors (line 10). Again, we check if it has the potential to be a better solution than any found so far (line 11). If this is the case, it is added to the OPEN list.

Different single-criteria search algorithms such as Dijkstra, A*, A*$_\varepsilon$ as well as bi-criteria search algorithms such BOA* fall under this framework. They differ with how OPEN is ordered and how the different functions (highlighted in Alg. 1) are implemented.

Bi-Objective A* (BOA*) To efficiently solve Problem 1, bi-Objective A* (BOA*) runs a best-first search. The algorithm is endowed with two heursitic functions h_1, h_2 estimating the cost to reach v_{goal} from any vertex according to c_1 and c_2, respectively. Here, we assume that these heuristic functions are admissible and consistent. This is key as the efficiency of BOA* relies on this assumption.

Given a node π_u, we define $g_i(\pi_u)$ to be the computed distance according to c_i. It can be easily shown that in best-first search algorithms $g_i := c_i(\pi_u)$. Additionally, we define $f_i(\pi_u) := g_i(\pi_u) + h_i(\pi_u)$. Although the cost and the g-value of a path can be used interchangeably, we will use the former to describe general properties of paths and the later to describe algorithm operations. Nodes in OPEN are ordered lexicographically according to (f_1, f_2) which concludes the description of how **extract_min** and **insert** (lines 3 and 13, respectively) are implemented.

Domination checks, which are typically time-consuming in bi-criteria search algorithms are implemented in $O(1)$ per node by maintaining for each vertex $u \in V$ the minimal cost to reach u according to c_2 computed so for. This value is maintained in a map $g_2^{\min} : V \to \mathbb{R}$ which is initialized to ∞ for each vertex. This allows to implement the function **is_dominated** for a node π_u by testing if

$$g_2(\pi_u) \geq g_2^{\min}(u) \text{ or } f_2(\pi_u) \geq g_2^{\min}(v_{\text{goal}}). \quad (1)$$

The first test checks if the node is dominated by an already-extended node and replaces the CLOSED list typically used in A*-like algorithms. The second test checks if the node has the potential to reach the goal with a solution whose cost is not dominated by any existing solution. Finally, the function **merge_to_solutions** simply adds a newly-found solution to the solution set.

Computing the approximate Pareto frontier Perny and Spanjaard (2008) suggest to compute an approximate Pareto frontier by endowing the algorithm with an approximation factor ε. When a node is popped from OPEN, we test if its f-value is ε-dominated by any solution that was already computed. While this algorithm was presented before BOA* and hence uses computationally-complex dominance checks, we can easily use this approach to adapt BOA* to compute an approximate Pareto frontier. This is done by replacing the dominance check in Eq. 1 with the test

$$g_2(\pi_u) \geq g_2^{\min}(v) \text{ or } (1+\varepsilon) \cdot f_2(\pi_u) \geq g_2^{\min}(v_{\text{goal}}). \quad (2)$$

We call this algorithm BOA*$_\varepsilon$.

4 Algorithmic Framework

4.1 Preliminaries

Recall that (single-criteria) shortest-path algorithms such as A* find a solution by computing the shortest path to all nodes that have the potential to be on the shortest path to the goal (namely, whose f-value is less than the current estimate of the cost to reach v_{goal}). Similarly, bi-criteria search algorithms typically compute for each node the subset of the Pareto frontier that has the potential to be in $\Pi_{v_{\text{goal}}}$.

Now, near-optimal (single-criteria) shortest-path algorithms such as A*$_\varepsilon$ (Pearl and Kim 1982) attempt to speed this process by only *approximating* the shortest path to intermediate nodes. Similarly, we suggest to construct only an approximate Pareto frontier for intermediate nodes which, in turn, will allow to dramatically reduce computation times. Looking at Fig. 1, one may suggest to run an A*-like search and if a path π_u on the Pareto frontier Π_u of u is approximately dominated by another path $\tilde{\pi}_u \in \Pi_u$, then discard π_u. Unfortunately, this does not account for paths in Π_u that may have been approximately dominated by π_u and hence discarded in previous iterations of the search. Existing methods use very conservative bounds to prune intermediate paths. For example, as stated in Sec. 1, if a bound L on the length of the longest path exists, we can use this strategy by replacing $(1+\varepsilon)$ with $(1+\varepsilon)^{1/L}$ to account for error propagation (Perny and Spanjaard 2008).

In contrast, we suggest a simple-yet-effective method to prune away approximately-dominated solutions using the notion of a partial Pareto frontier which we now define.

Definition 4 (Partial Pareto frontier PPF)**.** *Let $\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}} \in \Pi_u$ be two paths on the Pareto frontier of vertex u such that $c_1(\pi_u^{\texttt{tl}}) < c_1(\pi_u^{\texttt{br}})$ (here, `tl` and `br` are shorthands for "top left" and "bottom right" for reasons which will soon be clear). Their* partial Pareto frontier $\text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}} \subseteq \Pi_u$ *is a subset of a Pareto frontier such that if $\pi_u \in \Pi_u$ and $c_1(\pi_u^{\texttt{tl}}) < c_1(\pi_u) < c_1(\pi_u^{\texttt{br}})$ then $\pi_u \in \text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}}$. The paths $\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}$ are called the* extreme paths *of $\text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}}$. For a visualization, see Fig. 2.*

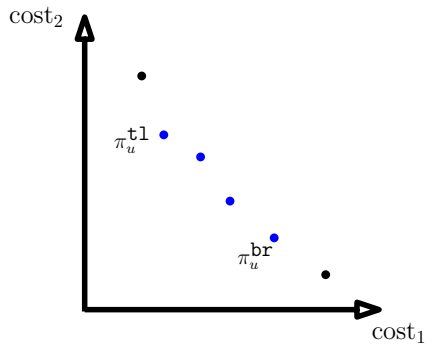


Figure 2: The partial Pareto frontier of two paths $\pi_u^{\texttt{tl}}$ and $\pi_u^{\texttt{br}}$ is the set of all paths (blue dots) on the Pareto frontier (blue and black dots) between these paths. Lemma 1 implies that any path represented by a blue dot is approximately dominated by $\pi_u^{\texttt{tl}}$ and $\pi_u^{\texttt{br}}$ for $\varepsilon_1 = \frac{c_1(\pi_u^{\texttt{br}}) - c_1(\pi_u^{\texttt{tl}})}{c_1(\pi_u^{\texttt{tl}})}$ and $\varepsilon_2 = \frac{c_2(\pi_u^{\texttt{tl}}) - c_2(\pi_u^{\texttt{br}})}{c_2(\pi_u^{\texttt{br}})}$.

Definition 5 (Bounded PPF)**.** *A partial Pareto frontier $\text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}} \subseteq \Pi_u$ is $(\varepsilon_1, \varepsilon_2)$-*bounded *if*

$$\varepsilon_1 \geq \frac{c_1(\pi_u^{\texttt{br}}) - c_1(\pi_u^{\texttt{tl}})}{c_1(\pi_u^{\texttt{tl}})} \text{ and } \varepsilon_2 \geq \frac{c_2(\pi_u^{\texttt{tl}}) - c_2(\pi_u^{\texttt{br}})}{c_2(\pi_u^{\texttt{br}})}.$$

Lemma 1. *If $\text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}}$ is an $(\varepsilon_1, \varepsilon_2)$-bounded partial Pareto frontier then any path in $\text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}}$ is $(\varepsilon_1, \varepsilon_2)$-dominated by both $\pi_u^{\texttt{tl}}$ and $\pi_u^{\texttt{br}}$.*

Proof. Let $\pi_u \in \text{PPF}_u^{\pi_u^{\texttt{tl}}, \pi_u^{\texttt{br}}}$. By definition, we have that $c_1(\pi_u^{\texttt{tl}}) < c_1(\pi_u)$ and that $\varepsilon_1 \geq \frac{c_1(\pi_u^{\texttt{br}}) - c_1(\pi_u^{\texttt{tl}})}{c_1(\pi_u^{\texttt{tl}})}$. Thus,

$$c_1(\pi_u^{\texttt{br}}) \leq (1+\varepsilon_1) \cdot c_1(\pi_u^{\texttt{tl}}) < (1+\varepsilon_1) \cdot c_1(\pi_u).$$

As $c_2(\pi_u^{\texttt{br}}) < c_2(\pi_u)$, we have that $\pi_u^{\texttt{br}}$ approximately dominates π_u.

Similarly, by definition, we have that $c_2(\pi_u) > c_2(\pi_u^{\texttt{br}})$ and that $\varepsilon_2 \geq \frac{c_2(\pi_u^{\texttt{tl}}) - c_2(\pi_u^{\texttt{br}})}{c_2(\pi_u^{\texttt{br}})}$. Thus,

$$c_2(\pi_u^{\texttt{tl}}) \leq (1+\varepsilon_2) \cdot c_1(\pi_u^{\texttt{br}}) < (1+\varepsilon_2) \cdot c_1(\pi_u).$$

As $c_1(\pi_u^{\texttt{tl}}) < c_1(\pi_u)$, we have that $\pi_u^{\texttt{tl}}$ approximately dominates π_u. □

4.2 Algorithmic description

In contrast to standard search algorithms which incrementally construct shortest paths from v_{start} to the graph vertices, our algorithm will incrementally construct $(\varepsilon_1, \varepsilon_2)$-bounded partial Pareto frontiers. Lemma 1 suggests a method to efficiently represent and maintain these frontiers for any approximation factors ε_1 and ε_2. Specifically, for a vertex u, PP-A* will maintain *path pairs* corresponding

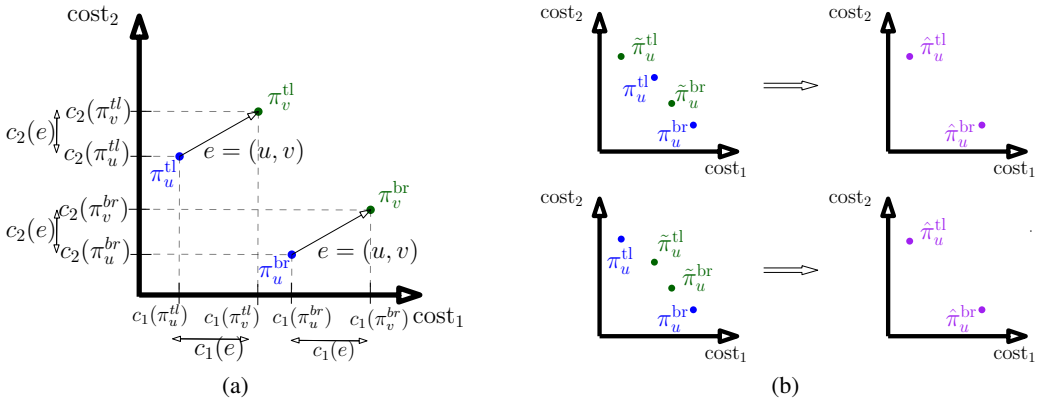


Figure 3: Operations on path pairs. (a) Extend operation. The path pair $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ (blue) is extended by edge $e = (u, v)$ to obtain the path pair $(\pi_v^{\mathrm{tl}}, \pi_v^{\mathrm{br}})$ (green). (b) Merge operation. Two examples of merging the path pair $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ (blue) with the path pair $(\tilde{\pi}_u^{\mathrm{tl}}, \tilde{\pi}_u^{\mathrm{br}})$ (green) to obtain the path pair $(\hat{\pi}_u^{\mathrm{tl}}, \hat{\pi}_u^{\mathrm{br}})$ (purple).

to the extreme paths in partial Pareto frontiers. For each path pair $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ we have that $c_1(\pi_u^{\mathrm{tl}}) \leq c_1(\pi_u^{\mathrm{br}})$ and $c_2(\pi_u^{\mathrm{tl}}) \geq c_2(\pi_u^{\mathrm{br}})$.

Before we explain how path pairs will be used let us define operations on path pairs: The first operation we consider is *extending* a path pair $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ by an edge $e = (u, v)$, which simply corresponds to extending both π_u^{tl} and π_u^{br} by e. The second operation we consider is *merging* two path pairs $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ and $(\tilde{\pi}_u^{\mathrm{tl}}, \tilde{\pi}_u^{\mathrm{br}})$. This operation constructs a new path pair $(\hat{\pi}_u^{\mathrm{tl}}, \hat{\pi}_u^{\mathrm{br}})$ such that

$$\hat{\pi}_u^{\mathrm{tl}} = \begin{cases} \pi_u^{\mathrm{tl}} & \text{if } c_1(\pi_u^{\mathrm{tl}}) \leq c_1(\tilde{\pi}_u^{\mathrm{tl}}) \\ \tilde{\pi}_u^{\mathrm{tl}} & \text{if } c_1(\tilde{\pi}_u^{\mathrm{tl}}) < c_1(\pi_u^{\mathrm{tl}}), \end{cases}$$

and

$$\hat{\pi}_u^{\mathrm{br}} = \begin{cases} \pi_u^{\mathrm{br}} & \text{if } c_2(\pi_u^{\mathrm{br}}) \leq c_2(\tilde{\pi}_u^{\mathrm{br}}) \\ \tilde{\pi}_u^{\mathrm{br}} & \text{if } c_2(\tilde{\pi}_u^{\mathrm{br}}) < c_2(\pi_u^{\mathrm{br}}). \end{cases}$$

For a visualization, see Fig. 3.

We are finally ready to describe PP-A*, our algorithm for bi-criteria approximate shortest-path computation (Problem 2). We run a best-first search similar to Alg. 1 but nodes are path pairs. We start with the trivial path pair $(v_{\mathrm{start}}, v_{\mathrm{start}})$ and describe our algorithm by detailing the different functions highlighted in Alg. 1. For each function, we describe what needs to be performed and how this can be efficiently implemented when consistent heuristics are used (see Sec. 3). Finally, the pseudocode of the algorithm is provided in Alg. 2 with the efficient implementations provided in Alg. 3-6.

Ordering nodes in OPEN: Recall that a node is a path pair $(\pi_u^{\mathrm{tl}}, \pi_u^{\mathrm{br}})$ and that each path π has two f values which correspond to the two cost functions and the two heuristic functions. Nodes are ordered lexicographically according to

$$(f_1(\pi_u^{\mathrm{tl}}), f_2(\pi_u^{\mathrm{br}})). \tag{3}$$

Domination checks: Recall that there are two types of domination checks that we wish to perform (i) checking if a node is dominated by a node that was already expanded and (ii) checking if a node has the potential to reach the goal with a solution whose cost is not dominated by any existing solution.

In our setting a path pair $\mathrm{PP_u}$ is dominated by another path pair $\tilde{\mathrm{PP}}_u$ if the partial Pareto frontier represented by $\mathrm{PP_u}$ is contained in the partial Pareto frontier represented by $\tilde{\mathrm{PP}}_u$ (see Fig. 4). We can efficiently test if $\mathrm{PP_u} = (\pi_{\mathrm{u}}^{\mathrm{tl}}, \pi_{\mathrm{u}}^{\mathrm{br}})$ is dominated by any path to u found so far, by checking if

$$g_2(\pi_u^{\mathrm{br}}) \geq g_2^{\min}(u). \tag{4}$$

This only holds when using the assumption that our heuristic functions are admissible and consistent and using the way

Algorithm 2 PP-A*

Input: $(G = (V, E), v_{\mathrm{start}}, v_{\mathrm{goal}}, c_1, c_2, h_1, h_2, \varepsilon_1, \varepsilon_2)$

```
solutions_pp ← ∅                                              ▷ path pairs
OPEN ← new path pair (v_start, v_start)
while OPEN ≠ ∅ do
  (π_u^tl, π_u^br) ← OPEN.extract_min()
  if is_dominated_PP-A*(π_u^tl, π_u^br) then
    continue
  if u = v_goal then                                          ▷ reached goal
    merge_to_solutions_PP-A*(π_u^tl, π_u^br, solutions_pp)
    continue
  for e = (u, v) ∈ neighbors(s(n), G) do
    (π_v^tl, π_v^br) ← extend_PP-A*((π_u^tl, π_u^br), e)
    if is_dominated_PP-A*(π_v^tl, π_v^br) then
      continue
    insert_PP-A*((π_v^tl, π_v^br), OPEN)
solutions ← ∅
for (π_{v_goal}^tl, π_{v_goal}^br) ∈ solutions_pp do
  solutions ← solutions ∪ {π_{v_goal}^tl}
return solutions
```

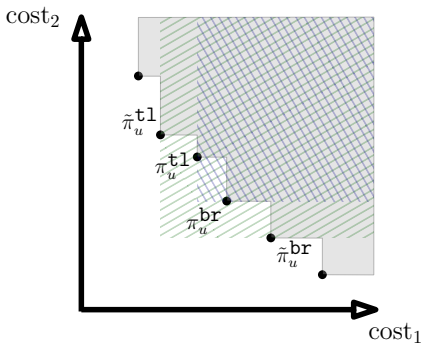


Figure 4: Testing dominance of partial Pareto frontiers using path pairs. The partial Pareto frontier $\Pi_u^{\pi_u^{\mathtt{tl}},\pi_u^{\mathtt{br}}}$ is contained in the partial Pareto frontier $\Pi_u^{\tilde{\pi}_u^{\mathtt{tl}},\tilde{\pi}_u^{\mathtt{br}}}$. Thus, the region represented by $\pi_u^{\mathtt{tl}}, \pi_u^{\mathtt{br}}$ is contained in the region represented by $\tilde{\pi}_u^{\mathtt{tl}}, \tilde{\pi}_u^{\mathtt{br}}$.

we order our OPEN list.

We now continue to describe how we test if a path pair has the potential to reach the goal with a solution whose cost is not dominated by any existing solution. Given a path pair $\mathrm{PP_u} = (\pi_{\mathrm{u}}^{\mathtt{tl}}, \pi_{\mathrm{u}}^{\mathtt{br}})$ a lower bound on the partial Pareto frontier at v_{goal} that can be attained via $\mathrm{PP_u}$ is obtained by adding the heuristic values to the costs of the two paths in $\mathrm{PP_u}$. Namely, we consider two paths $\pi_{v_{\mathrm{goal}}}^{\mathtt{tl}}, \pi_{v_{\mathrm{goal}}}^{\mathtt{br}}$ such that $c_i(\pi_{v_{\mathrm{goal}}}^{\mathtt{tl}}) := c_i(\pi_u^{\mathtt{tl}}) + h_i(u)$ and $c_i(\pi_{v_{\mathrm{goal}}}^{\mathtt{br}}) := c_i(\pi_u^{\mathtt{br}}) + h_i(u)$. Note that these paths may not be attainable and are a lower bound on the partial Pareto frontier that can be obtained via $\mathrm{PP_u}$. Now, if the partial Pareto frontier $\mathrm{PPF}_{v_{\mathrm{goal}}}^{\pi_{v_{\mathrm{goal}}}^{\mathtt{tl}},\pi_{v_{\mathrm{goal}}}^{\mathtt{br}}}$ is contained in the union of the currently-computed partial Pareto frontiers at v_{goal}, then $\mathrm{PP_u}$ is dominated. Similar to the previous dominance check, this can be efficiently implemented by testing if

$$(1+\varepsilon_2)\cdot(f_2(\pi_u^{\mathtt{br}})) \geq g_2^{\min}(v_{\mathrm{goal}}). \tag{5}$$

Inserting nodes in OPEN: Recall that we want to use the notion of path pairs to represent a partial Pareto frontier. Key to the efficiency of our algorithm is to have every partial Pareto frontier as large as possible under the constraint that they are all $(\varepsilon_1, \varepsilon_2)$-bounded. Thus, when coming to insert a path pair $\mathrm{PP_u}$ into the OPEN list, we check if there exists a path pair $\tilde{\mathrm{PP}}_{\mathrm{u}}$ such that $\mathrm{PP_u}$ and $\tilde{\mathrm{PP}}_{\mathrm{u}}$ can be merged and the resultant path pair is still $(\varepsilon_1, \varepsilon_2)$-bounded.

If this is the case, we remove $\tilde{\mathrm{PP}}_{\mathrm{u}}$ and replace it with the merged path pair.

Merging solutions: Since we want to minimize the number of path pairs representing $\Pi_{v_{\mathrm{goal}}}(\varepsilon_1, \varepsilon_2)$ we suggest an optimization that operates similarly to node insertions. When a new path pair $\mathrm{PP}_{\mathrm{v_{goal}}}$ representing a partial Pareto frontier at v_{goal} is obtained, we test if there exists a path pair in the solution set $\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}}$ such that $\mathrm{PP_u}$ and $\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}}$ can be merged and the resultant path pair is still $(\varepsilon_1, \varepsilon_2)$-bounded.

Algorithm 3 is_dominated_PP-A*

Input: $(\mathrm{PP_u} = (\pi_{\mathrm{u}}^{\mathtt{tl}}, \pi_{\mathrm{u}}^{\mathtt{br}}))$

if $(1+\varepsilon_2)\cdot f_2(\pi_u^{\mathtt{br}}) \geq g_2^{\min}(v_{\mathrm{goal}})$ **then**
  return `true` ▷ dominated by solution
if $g_2(\pi_u^{\mathtt{br}}) \geq g_2^{\min}(u)$ **then**
  return `true` ▷ dominated by existing path pair
return `false`

Algorithm 4 extend_PP-A*

Input: $(\mathrm{PP_u} = (\pi_{\mathrm{u}}^{\mathtt{tl}}, \pi_{\mathrm{u}}^{\mathtt{br}}), \mathrm{e} = (\mathrm{u}, \mathrm{v}))$

$\pi_v^{\mathtt{tl}} \leftarrow$**extend**$(\pi_u^{\mathtt{tl}})$
$\pi_v^{\mathtt{br}} \leftarrow$**extend**$(\pi_u^{\mathtt{br}})$
return $(\pi_v^{\mathtt{tl}}, \pi_v^{\mathtt{br}})$

Algorithm 5 insert_PP-A*

Input: $(\mathrm{PP_v}$, OPEN$)$

for each path pair $\tilde{\mathrm{PP}}_{\mathrm{v}} \in$ OPEN **do**
  $\mathrm{PP}_{\mathrm{v}}^{\mathrm{merged}} \leftarrow \mathrm{merge}(\tilde{\mathrm{PP}}_{\mathrm{v}}, \mathrm{PP_v})$
  if $\mathrm{PP}_{\mathrm{v}}^{\mathrm{merged}}$.is_bounded$(\varepsilon_1, \varepsilon_2)$ **then**
    OPEN.remove$(\tilde{\mathrm{PP}}_{\mathrm{v}})$ ▷ remove existing path pair
    OPEN.**insert**$(\mathrm{PP}_{\mathrm{v}}^{\mathrm{merged}})$
    return
OPEN.**insert**$(\mathrm{PP_v})$
return

Algorithm 6 merge_to_solutions_PP-A*

Input: $(\mathrm{PP}_{\mathrm{v_{goal}}}$, solutions_pp$)$

for each path pair $\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}} \in$ solutions_pp **do**
  $\mathrm{PP}_{\mathrm{v_{goal}}}^{\mathrm{merged}} \leftarrow \mathrm{merge}(\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}}, \mathrm{PP_v})$
  if $\mathrm{PP}_{\mathrm{v_{goal}}}^{\mathrm{merged}}$.is_bounded$(\varepsilon_1, \varepsilon_2)$ **then**
    solutions_pp.remove$(\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}})$
    solutions_pp.insert$(\mathrm{PP}_{\mathrm{v_{goal}}}^{\mathrm{merged}})$
    return
solutions_pp.insert$(\mathrm{PP}_{\mathrm{v_{goal}}})$
return

If this is the case, we remove $\tilde{\mathrm{PP}}_{\mathrm{v_{goal}}}$ and replace it with the merged path pair.

Returning solutions: Recall that our algorithm stores solutions as path pairs and not individual paths. To return an approximate Pareto frontier, we simply return one path in each path pair. Here, we arbitrarily choose to return $\pi_{v_{\mathrm{goal}}}^{\mathtt{tl}}$ for each path pair $(\pi_{v_{\mathrm{goal}}}^{\mathtt{tl}}, \pi_{v_{\mathrm{goal}}}^{\mathtt{br}})$.

4.3 Analysis

Showing that PP-A* indeed computes an approximate Pareto frontier using the domination checks suggested in Eq. 4 and 5, may be done by using similar arguments as those presented in (Ulloay et al. 2020). However, such a proof is omitted due to lack of space and we refer the reader

New York City (NY)								
264,346 states, 730,100 edges								
ε	avg n_{sol}		avg t		min t		max t	
	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*
0	158	158	1,047	405	2	0	13,563	5,038
0.01	19	20	291	353	3	0	3,662	4,577
0.025	10	10	168	295	2	0	2,207	4,101
0.05	6	6	111	240	3	0	1,523	3,538
0.1	4	4	69	174	2	0	932	2,694
San Francisco Bay (BAY)								
321,270 states, 794,830 edges								
ε	avg n_{sol}		avg t		min t		max t	
	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*
0	117	117	1,213	423	3	0	21,751	7,584
0.01	16	17	222	369	4	0	2,927	6,805
0.025	9	9	127	321	3	0	1,530	5,614
0.05	5	6	85	272	3	0	1,109	4,570
0.1	3	4	54	199	3	0	576	3,056
Colorado (COL)								
435,666 states, 1,042,400 edges								
ε	avg n_{sol}		avg t		min t		max t	
	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*
0	318	318	3,368	1,144	5	1	56,153	17,348
0.01	15	16	372	944	5	1	3,633	16,304
0.025	7	8	192	768	5	1	1,690	15,037
0.05	4	5	116	608	5	1	991	13,718
0.1	3	3	69	470	4	1	593	11,977
Florida (FL)								
1,070,376 states, 2,712,798 edges								
ε	avg n_{sol}		avg t		min t		max t	
	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*
0	357	357	12,177	3,545	12	3	270,450	68,467
0.01	12	13	1,000	3,228	12	3	17,092	64,642
0.025	6	6	479	2,738	11	3	8,060	59,908
0.05	3	4	263	1,985	12	3	3,945	39,214
0.1	2	2	144	1,172	11	2	1,780	21,665

Table 1: Average number of solutions (n_{sol}) and runtime (in ms) comparing BOA$^*_\varepsilon$ and PP-A* on 50 random queries sampled for four different roadmaps for different approximation factors.

to the extended version of this text (Salzman 2020).

5 Evaluation

Experimental setup. To evaluate our approach we compare it to BOA$^*_\varepsilon$ as BOA* was recently shown to dramatically outperform other state-of-the-art algorithms for bicriteria shortest path (Ulloay et al. 2020). All experiments were run on an 1.8GHz Intel(R) Core(TM) i7-8565U CPU Windows 10 machine with 16GB of RAM. All algorithm implementations were in C++. We use road maps from the 9'th DIMACS Implementation Challenge: Shortest Path[4]. The cost components represent travel distances (c_1) and times (c_2). The heuristic values are the exact travel distances and times to the goal state, computed with Dijkstra's algorithm. Since all algorithms use the same heuristic values, heuristic-computation times are omitted.

General comparison. Similar to the experiments of Ulloa et al (2020) we start by comparing the algorithms for four different roadmaps containing between roughly $250K$ and $1M$ vertices. Table 1 summarizes the number of solutions in the approximate Pareto frontier and average, minimum and maximum running times of the two algorithms using the following values[5] $\varepsilon \in \{0, 0.01, 0.025, 0.05, 0.1\}$. Here, approximation values of zero and 0.01 correspond to computing the entire Pareto frontier and approximating it using a value of 1%, respectively.

[4]http://users.diag.uniroma1.it/challenge9/download.shtml.

North East (NE)						
1,524,453 states, 3,897,636 edges						
ε	avg t		min t		max t	
	PP-A*	BOA*	PP-A*	BOA*	PP-A*	BOA*
0	192.6	59.5	0.04	0.02	2,4189.9	592.6
0.01	13.1	68.3	0.03	0.01	111.6	600.9
0.025	5.6	57.3	0.02	0.01	46.9	510.9
0.05	2.7	40.8	0.02	0.01	22.6	345.1
0.1	1.3	25.8	0.02	0.01	9.0	229.8

Table 2: Runtime (in seconds) comparing BOA* and PP-A* on 50 random queries sampled for the NE map.

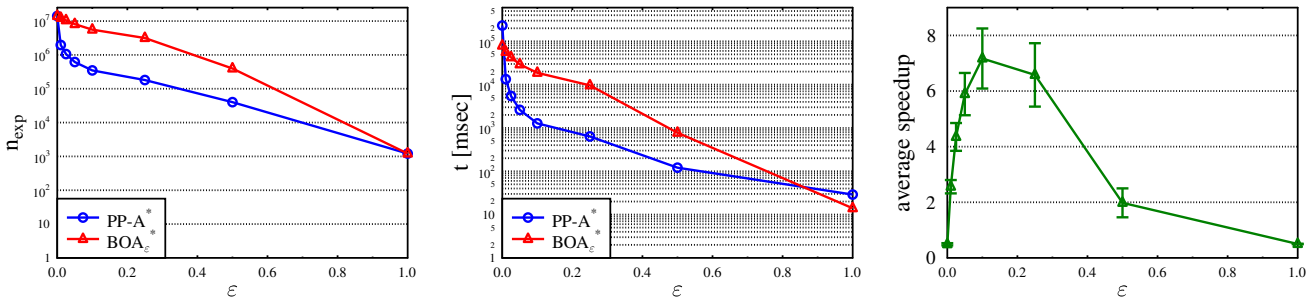
When computing the entire Pareto frontier BOA* is roughly three times faster than PP-A* on average. This is to be expected as PP-A* stores for each element in the priority queue two paths and requires more computationally-demanding operations. As the approximation factor is increased, the average running time of PP-A* drops faster, when compared to BOA$^*_\varepsilon$ and we observe a significant average speedup. Interestingly, when looking at the minimal running time, BOA$^*_\varepsilon$ significantly outperforms PP-A*. This is because in such settings the approximate Pareto frontier contains one solution, which BOA$^*_\varepsilon$ is able to compute very fast. Other nodes are approximately dominated by this solution and the algorithm can terminate very quickly. PP-A*, on the other hand, still performs merge operations which incur a computational overhead. When looking at the maximal running time, we can see an opposite trend where PP-A* outperforms BOA$^*_\varepsilon$ by a large factor.

Pinpointing the performance differences between PP-A* and BOA$^*_\varepsilon$. The first set of results suggest that as the problem becomes harder, the speedup that PP-A* may offer becomes more pronounced. We empirically quantify this claim by moving to a larger map called the North East (NE) map which contains 1,524,453 states and 3,897,636 edges where we obtain even larger speedups (see Table 2).

We plot both the number of nodes expanded (which typically is proportional to running time of A*-like algorithms) as well as the running time of each algorithm as a function of the approximation factor (see, Fig. 5a and 5b, respectively). Here we used $\varepsilon \in \{0, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1\}$.

We observe that the number of nodes expanded monotonically decreases when the approximation factor is increased for both algorithms. This is because additional nodes may be pruned which in turn, prunes all nodes in their subtree. It is important to discuss *how* these nodes are pruned: Recall that BOA$^*_\varepsilon$ prunes nodes according to Eq. 2. Thus, increasing the approximation factor only allows to prune more nodes ac-

[5]While PP-A* allows a user to specify two approximation factors corresponding to the two cost functions, this is not the case for BOA*. Thus, in all experiments we use a single approximation factor ε and set $\varepsilon_1 = \varepsilon_2 = \varepsilon$.



(a) (b) (c)

Figure 5: North East (NE) plots. (a) The average number of expanded nodes (n_{exp}) and (b) the average time for both algorithms as a function of the approximation factor. Notice the logarithmic scale in the y-axis for both plots. (c) The average speedup of PP-A* when compared to BOA*$_\varepsilon$ as a function of the approximation factor. Error bars denote one standard error (error bars in (a) and (b) are not visible due to the logarithmic scale).

cording to the already-computed solutions and not according to the paths computed to intermediate nodes. In contrast, PP-A* prunes nodes according to Eq. 4 and 5. Thus, in addition to more path pairs being merged, increasing the approximation allows to prune more path pairs according to the already-computed solutions as well as the path pairs computed to intermediate vertices. Thus, for relatively-small approximation factors that are greater than zero (in our setting, $0 < \varepsilon < 0.5$, we see that BOA* expands a significantly higher number of nodes than PP-A* which explains the speedups we observed. However, for large approximation factors, there is typically only one solution in the approximate Pareto frontier. This solution, which is found quickly by BOA*$_\varepsilon$, allows to prune almost all other paths which results in BOA*$_\varepsilon$ running much faster than PP-A*. This trend is visualized in Fig. 5c.

6 Future research

6.1 Bidirectional search

We presented PP-A* as a unidirectional search algorithm, however a common approach to speed up search algorithms is to perform two simultaneous searches: a forward search from v_{start} to v_{goal} and a backward search from v_{goal} to v_{start} (Pohl 1971). Thus, an immediate task for future research is to suggest a bidirectional extension of PP-A*. Here we can build upon recent progress in bi-directional search algorithms for bi-criteria shortest-path problems (Sedeño-Noda and Colebrook 2019).

6.2 Beyond two optimization criteria

We presented PP-A* as a search algorithm for two optimization criteria, however the same concepts can be used for multi-criteria optimization problems. Unfortunately, it is not clear how to perform operations such as dominance checks efficiently since the methods presented for BOA* do not extend to such settings.

References

[Bachmann et al. 2018] Bachmann, D.; Bökler, F.; Kopec, J.; Popp, K.; Schwarze, B.; and Weichert, F. 2018. Multi-objective optimisation based planning of power-line grid expansions. *ISPRS International Journal of Geo-Information* 7(7):258.

[Bökler and Chimani 2020] Bökler, F., and Chimani, M. 2020. Approximating multiobjective shortest path in practice. In *Symposium on Algorithm Engineering and Experiments, (ALENEX)*, 120–133.

[Breugem, Dollevoet, and van den Heuvel 2017] Breugem, T.; Dollevoet, T.; and van den Heuvel, W. 2017. Analysis of FPTASes for the multi-objective shortest path problem. *Computers & Operations Research* 78:44–58.

[Bronfman et al. 2015] Bronfman, A.; Marianov, V.; Paredes-Belmar, G.; and Lüer-Villagra, A. 2015. The maximin HAZMAT routing problem. *European Journal of Operational Research* 241(1):15–27.

[Chinchuluun and Pardalos 2007] Chinchuluun, A., and Pardalos, P. M. 2007. A survey of recent developments in multiobjective optimization. *Annals of Operations Research* 154(1):29–50.

[Ehrgott 2005] Ehrgott, M. 2005. *Multicriteria Optimization (2. ed.)*. Springer.

[Fu et al. 2019] Fu, M.; Kuntz, A.; Salzman, O.; and Alterovitz, R. 2019. Toward asymptotically-optimal inspection planning via efficient near-optimal graph search. In *Robotics: Science and Systems (RSS)*.

[Hart, Nilsson, and Raphael 1968] 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.

[Legriel et al. 2010] Legriel, J.; Le Guernic, C.; Cotton, S.; and Maler, O. 2010. Approximating the pareto front of multi-criteria optimization problems. In *International Con-*

ference on Tools and Algorithms for the Construction and Analysis of Systems, 69–83.

[Mandow and De La Cruz 2005] Mandow, L., and De La Cruz, J. L. P. 2005. A new approach to multiobjective A* search. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 218–223.

[Mandow and De La Cruz 2010] Mandow, L., and De La Cruz, J. L. P. 2010. Multiobjective A* search with consistent heuristics. *Journal of the ACM (JACM)* 57(5):1–25.

[Papadimitriou and Yannakakis 2000] Papadimitriou, C. H., and Yannakakis, M. 2000. On the approximability of tradeoffs and optimal access of web sources. In *Symposium on Foundations of Computer Science (FoCS)*, 86–92.

[Pearl and Kim 1982] Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE transactions on pattern analysis and machine intelligence* (4):392–399.

[Perny and Spanjaard 2008] Perny, P., and Spanjaard, O. 2008. Near admissible algorithms for multiobjective search. In *European Conference on Artificial Intelligence (ECAI)*, volume 178, 490–494.

[Pohl 1971] Pohl, I. 1971. Bi-directional search. *Machine intelligence* 6:127–140.

[Pulido, Mandow, and Pérez-de-la Cruz 2015] Pulido, F.-J.; Mandow, L.; and Pérez-de-la Cruz, J.-L. 2015. Dimensionality reduction in multiobjective shortest path search. *Computers & Operations Research* 64:60–70.

[Salzman 2020] Salzman, O. 2020. Approximate bi-criteria search by efficient representation of subsets of the pareto-optimal frontier. *CoRR* abs/2006.10302.

[Sedeño-Noda and Colebrook 2019] Sedeño-Noda, A., and Colebrook, M. 2019. A biobjective dijkstra algorithm. *European Journal of Operational Research* 276(1):106–118.

[Sedeno-Noda and Raith 2015] Sedeno-Noda, A., and Raith, A. 2015. A dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem. *Computers & Operations Research* 57:83–94.

[Serafini 1987] Serafini, P. 1987. Some considerations about computational complexity for multi objective combinatorial problems. In *Recent advances and historical development of vector optimization*. Springer. 222–232.

[Stewart and White III 1991] Stewart, B. S., and White III, C. C. 1991. Multiobjective A*. *Journal of the ACM (JACM)* 38(4):775–814.

[Tsaggouris and Zaroliagis 2009] Tsaggouris, G., and Zaroliagis, C. D. 2009. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. *Theory Comput. Syst.* 45(1):162–186.

[Ulloay et al. 2020] Ulloay, C. H.; Yeohz, W.; Baier, J. A.; Zhang, H.; Suazoy, L.; and and, S. K. 2020. A simple and fast bi-objective search algorithm. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

[Vazirani 2001] Vazirani, V. V. 2001. *Approximation algorithms*. Springer.

[Warburton 1987] Warburton, A. 1987. Approximation of pareto optima in multiple-objective, shortest-path problems. *Operations research* 35(1):70–79.

An Atom-Centric Perspective on Stubborn Sets

Gabriele Röger, Malte Helmert, Jendrik Seipp, Silvan Sievers
University of Basel
Basel, Switzerland
{gabriele.roeger,malte.helmert,jendrik.seipp,silvan.sievers}@unibas.ch

Abstract

Stubborn sets are an optimality-preserving pruning technique for factored state-space search. Their applicability in classical planning is limited by their computational overhead. We describe a new algorithm for computing stubborn sets that is based on the state variables of the state space, while previous algorithms are based on its actions. Typical factored state spaces tend to have far fewer state variables than actions, and therefore our new algorithm is much more efficient than the previous state of the art, making stubborn sets a viable technique in many cases where they previously were not.

An archival version of this paper has been published at SoCS 2020 (Röger et al. 2020a).

Introduction

Heuristic search is a common approach for classical planning. Especially in *optimal* planning, the search suffers from a state explosion problem that occurs if states can be reached by applying the same actions in different orders. Even with close-to-perfect heuristics, the number of nodes that must be explored by pure heuristic search (only relying on node expansions and an admissible heuristic) can grow exponentially in the size of the task (Helmert and Röger 2008). Hence, search algorithms are often enhanced with pruning techniques that reduce the size of the explored state space.

One family of such pruning techniques is *partial order reduction*, which allows the search to ignore some paths to the goal by not considering all permutations of the actions. Intuitively, the idea is to avoid interleaving the solution of independent subproblems but instead solving one subproblem after the other. Partial order reduction was originally introduced by Valmari (1989) for Petri nets in the context of computer-aided verification. Alkhazraji et al. (2012) transferred his concept of *strong stubborn sets* to classical planning. Later on, Wehrle and Helmert (2014) generalized them with more fine-grained criteria that are still sufficient for optimality-preserving pruning. With suitable decisions at certain choice points, strong stubborn sets strictly dominate the *expansion core method* (Chen and Yao 2009; Wehrle and Helmert 2012), a partial order reduction technique introduced earlier for planning (Wehrle et al. 2013).

A stubborn set for a state is a set of actions such that all other actions can safely be ignored at its expansion. The concept is inherently action-centric and so are the underlying definitions and algorithms. In this paper, we adopt a more atom-centric perspective on their computation, which gives rise to a significantly faster algorithm. As an additional enhancement, we also contribute a new atom selection strategy, which has a tendency to produce smaller stubborn sets and leads to more pruning in our experiments.

Background

We consider SAS$^+$ planning tasks (Bäckström and Nebel 1995), extended with non-negative action costs. A task is defined over a finite set $\mathcal{V}$ of *variables*, each associated with a finite domain $\mathcal{D}(v)$. A pair (v, d) with $v \in \mathcal{V}$ and $d \in \mathcal{D}(v)$ is called an *atomic proposition*, or *atom* for short, and we use $\mathcal{P}$ to denote the set of all atomic propositions (over an implicit set of variables $\mathcal{V}$). We call all atoms (v, d') with $d' \in \mathcal{D}_v \setminus \{d\}$ the *siblings* of atom (v, d).

A *partial state* s maps every variable v from a set $\mathit{vars}(s) \subseteq \mathcal{V}$ to a value $s[v]$ from $\mathcal{D}(v)$. If $\mathit{vars}(s) = \mathcal{V}$, we call s a *state*. When it is suitable, we also consider a partial state s as the set of atoms $\{(v, s[v]) \mid v \in \mathit{vars}(s)\}$ and write $(v, d) \in s$ for $s[v] = d$.

A *task* is given as a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_{\mathrm{I}}, s_{\mathrm{G}} \rangle$ where $\mathcal{V}$ is the finite set of variables, $\mathcal{A}$ a finite set of *actions*, s_{I} a state called the *initial state* and s_{G} a partial state called the *goal*. Each *action* $a \in \mathcal{A}$ is defined by its *cost* $c(a) \in \mathbb{R}_0^+$ and two partial states $\mathit{pre}(a)$ and $\mathit{eff}(a)$ called its *precondition* and *effect*. If $(v, d) \in \mathit{eff}(a)$ for some atom (v, d), we say that action a *achieves* (v, d). If $(v, d) \in \mathit{pre}(a)$, we say a *depends on* (v, d). W.l.o.g. we require that no action both depends on and achieves the same atom.

An action a is *applicable* in state s if $\mathit{pre}(a) \subseteq s$. Then the *successor state* s' is given as $s'[v] = \mathit{eff}(a)[v]$ for all $v \in \mathit{vars}(\mathit{eff}(a))$ and $s'[v] = s[v]$ for all other variables. Slightly abusing notation, we write $a(s)$ for the successor state resulting from applying action a in state s.

A *goal state* is a state s with $s_{\mathrm{G}} \subseteq s$. A *plan* is a sequence of actions that are subsequently applicable in s_{I} and where the resulting state is a goal state. The cost of a plan is the sum of the individual action costs. A plan is *optimal* if it has minimal cost among all plans. Wehrle and Helmert (2014) pointed out that for correct pruning it is sometimes impor-

tant to only consider so-called *strongly optimal* plans, which are optimal plans with a minimal number of 0-cost actions among all optimal plans. If there is no plan for a task, the task is *unsolvable*. The aim of *optimal planning* is to find an optimal plan or to prove that the task is unsolvable.

Strong stubborn sets aim to prune permuted plans from the search. On a lower level, the permutation of actions is related to the following notion of *interference*.

Definition 1 (interference, Wehrle and Helmert 2014)**.** *Let a_1 and a_2 be actions and let s be a state of a planning task Π. We say that a_1 and a_2 interfere in s if they are both applicable in s, and*

- *a_1* disables *a_2, i.e., a_2 is not applicable in $a_1(s)$, or*
- *a_2 disables a_1, or*
- *a_1 and a_2* conflict *in s, i.e., $a_2(a_1(s))$ and $a_1(a_2(s))$ are both defined but differ.*

If two actions that are both applicable in a state s do *not* interfere in s, we can apply them in any order and will in both cases reach the same state.

The second relevant notion are *necessary enabling sets*. These are related to disjunctive action landmarks (Helmert and Domshlak 2009), which are sets of actions of which at least one must be applied in every plan. Similarly, necessary enabling sets are sets of actions of which at least one must be applied *before a given action is applied* in every action sequence from a *given set*.

Definition 2 (necessary enabling set, Wehrle and Helmert 2014)**.** *Let Π be a planning task, let a be one of its actions, and let Seq be a set of action sequences applicable in the initial state of Π.*

A necessary enabling set *for a and Seq is a set N of actions such that every action sequence in Seq which includes a as one of its actions also includes some action $a' \in N$ before the first occurrence of a.*

For this paper, we build on the generalized definition of strong stubborn sets by Wehrle and Helmert (2014) but for clarity we omit the concept of *envelopes*, which permit to safely ignore some actions. Empirically, the known methods for exploiting envelopes did not provide much benefit (Wehrle and Helmert 2014), and they can easily be reintegrated in our work.

Definition 3 (strong stubborn set)**.** *Let s be a state of planning task $\Pi = \langle \mathcal{V}, \mathcal{A}, s_\mathrm{I}, s_\mathrm{G} \rangle$ and let $\Pi_s = \langle \mathcal{V}, \mathcal{A}, s, s_\mathrm{G} \rangle$. A* strong stubborn set *in s is a set $A \subseteq \mathcal{A}$ of actions that satisfies the following conditions.*

If Π_s is unsolvable or s is a goal state, then every A is a strong stubborn set. Otherwise, let Opt be the set of strongly optimal plans for Π_s and let $\mathcal{S}_{Opt}$ be the set of states that are visited by at least one plan in Opt. The following conditions must be true for A to be a strong stubborn set.

- C1 *A contains at least one action from at least one plan from Opt.*
- C2 *For every $a \in A$ that is not applicable in s, A contains a necessary enabling set for a and Opt.*
- C3 *For every $a \in A$ applicable in s, A contains all actions from $\mathcal{A}$ that interfere with a in any state $s' \in \mathcal{S}_{Opt}$.*

Wehrle and Helmert (2014) showed that the cost of an optimal solution does not change if for every state in the state space we only preserve the outgoing transitions that correspond to an action from a strong stubborn set. Put differently, in each state visited during the search we can prune all actions that are *not* in the strong stubborn set, while preserving the guarantee to find optimal solutions.

In practice, it is impossible to efficiently determine a minimal strong stubborn set because we do not know *Opt* and $\mathcal{S}_{Opt}$. However, if C2 and C3 hold for an overapproximation of these sets, they must also hold for the required sets.

Since the set $\mathcal{S}_{Opt}$ cannot be efficiently computed, for C3 it is common to use a state-independent overapproximation of interference. Alkhazraji et al. (2012) and Wehrle et al. (2013) use a purely syntactic criterion: actions a and a' *potentially conflict* in any state if there is a variable $v \in \mathit{vars}(\mathit{eff}(a)) \cap \mathit{vars}(\mathit{eff}(a'))$ such that $\mathit{eff}(a)[v] \neq \mathit{eff}(a')[v]$. Action a *potentially disables* a' if there is a variable $v \in \mathit{vars}(\mathit{eff}(a)) \cap \mathit{vars}(\mathit{pre}(a'))$ such that $\mathit{eff}(a)[v] \neq \mathit{pre}(a')[v]$. Two actions a and a' then *potentially interfere* if they potentially conflict, a potentially disables a', or a' potentially disables a. With this definition, two actions potentially interfere if *there exists some* state in which they interfere.

Wehrle and Helmert (2014) strengthen this approach with mutex information: if the preconditions of two actions are mutually exclusive, they cannot both be applicable in a reachable state, so they never interfere in these states.

It is also already intractable to determine whether a given action is an element of *Opt*. We can still determine a necessary enabling set for a and *Opt* by collecting all achievers of an atom that is not true in s but which a depends on. While this set is not minimal, it can be computed efficiently and indeed this is the strategy employed by previous algorithms for constructing strong stubborn sets in planning. Similarly, C1 is satisfied by picking an atom from the goal that is not true in s and including all actions that achieve this atom, hence including at least one action from every plan.

Existing Action-Centric Algorithm

To satisfy the properties of strong stubborn sets, previous algorithms start from an action set that satisfies C1 and successively add actions to satisfy C2 and C3 until a fixed point is reached. We will adopt the same high-level approach but will differ from this action-centric approach on a lower level.

Before we go into details, we first introduce and analyze the action-centric algorithm. Previously published pseudo-code (Alkhazraji et al. 2012; Al-Khazraji 2017) does not have the level of detail we require for our discussion, but an implementation by Wehrle and Helmert (2014) is available as part of Fast Downward (Helmert 2006). We extracted the pseudo-code as Algorithm 1 from Fast Downward 19.12.[1]

The algorithm collects all actions to be included in the strong stubborn set for a non-goal state s in a collection *stubborn*. The actions for which it still needs to ensure C2 and C3 are tracked in a collection *queue*. To avoid clutter in the pseudo-code, we assume that *stubborn*, *queue* and the components of the task are globally accessible.

[1] http://www.fast-downward.org/Releases/19.12

Algorithm 1 Action-centric algorithm

```
 1: function ComputeStubbornSet(s)
 2:     stubborn = empty collection
 3:     queue= empty collection

 4:     procedure MarkAsStubborn(a)
 5:         if a ∉ stubborn then
 6:             stubborn.add(a)
 7:             queue.add(a)

 8:     procedure EnqueueInterferers(a)
 9:         for a' ∈ A potentially interfering with a do
10:             MarkAsStubborn(a')

11:     procedure EnqueueAchievers(atom)
12:         for a ∈ A with atom ∈ eff(a) do
13:             MarkAsStubborn(a)

14:     atom = some unsatisfied goal atom
15:     EnqueueAchievers(atom)
16:     while queue is not empty do
17:         a = queue.pop()                          ▷ any element
18:         if a is applicable in s then
19:             EnqueueInterferers(a)
20:         else
21:             ▷ Enqueue a necessary enabling set for a
22:             atom = some unsatisfied atom from pre(a)
23:             EnqueueAchievers(atom)
24:     return stubborn
```

The overall process for generating a strong stubborn set starts with collecting a set of actions to satisfy C1 (lines 14–15). As long as the other conditions are not yet guaranteed for some action a (lines 16–17), it includes further actions to ensure C2 (lines 20–23) or C3 (lines 18–19), depending on whether a is applicable in state s or not.

Whenever an action should be included in the result (marked as stubborn), the algorithm checks if it has already been included previously and if not includes it and enqueues it for further processing into *queue* (lines 4–7).

As mentioned above, necessary enabling sets are generated by starting from an atom and collecting all actions that achieve it (lines 11–13).

Complexity Analysis

In the complexity analysis, we use $p_{\max}$ for the maximal size of a partial state occurring as precondition or effect of any action. In typical planning tasks, this is quite a low number. In general, it can be bounded by the number $|\mathcal{V}|$ of variables.

For an efficient implementation of Algorithm 1, we assume that all state-independent information is precomputed and stored once for every task (i.e., only once for the entire search, not once for every node that is expanded). This affects the set of achievers for every atom (used in line 12) and the interference relation (used in line 9).

The achievers can be determined by one pass over all actions that scans the effect and registers the action accordingly. This requires time $O(|\mathcal{A}|p_{\max})$ and the result can be stored in space $O(|\mathcal{P}||\mathcal{A}|)$.

Exploiting pre-sorted action preconditions and effects, the interference relation can be computed in $O(|\mathcal{A}|^2 p_{\max})$, ranging over all pairs of actions and syntactically testing their potential interference in time $O(p_{\max})$. With no influence on Big-O, we can halve the effort by exploiting that the relation is symmetric. The result can be stored in $O(|\mathcal{A}|^2)$. Fast Downward uses a lazy implementation that only performs the computation for an action once it is required.

We now analyze the time complexity of a single call of ComputeStubbornSet. With suitable data structures for *stubborn* (e.g., a bitset) and *queue* (e.g., an array-based stack), MarkAsStubborn takes constant time. Then EnqueueInterferers takes time $O(|\mathcal{A}|)$ because there are at most $|\mathcal{A}|$ interfering actions which have been precomputed. Analogously, EnqueueAchievers runs in $O(|\mathcal{A}|)$.

In lines 14 and 22 the algorithm selects an unsatisfied atom from a partial state. Wehrle and Helmert (2014) discussed several such *atom selection strategies*—taken from the literature and new—with different time requirements. To stay general, we account for them with $O(t)$, resulting in time $O(t + |\mathcal{A}|)$ for lines 14–15.

Each iteration of the while loop takes time $O(p_{\max})$ for testing applicability plus $O(t+|\mathcal{A}|)$ accounting for the more expensive else-case of the if statement. As every action is added to *queue* at most once, the overall runtime of ComputeStubbornSet is $O(|\mathcal{A}|(p_{\max} + t + |\mathcal{A}|))$. The space complexity for *stubborn* and *queue* is $O(|\mathcal{A}|)$.

New Atom-Centric Algorithm

The original fixed-point computation from Algorithm 1 tracks (in *queue*) actions that have already been included in the stubborn set but for which it is not yet sure that C2 and C3 are satisfied.

We now reconsider the overapproximation of the interference relation and necessary enabling sets and what this implies for the computation of strong stubborn sets. We begin with potential interference. Using the notion of sibling atoms, we can paraphrase the set of actions that potentially interfere with action a: it consists of all actions a' s.t.

- a' achieves a sibling of an atom in *pre*(a) (a' potentially disables a), or
- a' depends on a sibling of an atom in *eff*(a) (a potentially disables a'), or
- a' achieves a sibling of an atom in *eff*(a) (a and a' potentially conflict).

Observation 1: We can characterize these actions by only considering the occurrence of individual atoms in their precondition or effect.

Observation 2: The same is true for the actions in the necessary enabling set.

Observation 3: The order in which the actions are processed is not important for the fixed-point computation.[2]

[2]The order can influence *dynamic* atom selection strategies, but we are not aware of any work that aims for a specific order.

Algorithm 2 Atom-centric algorithm

```
 1: function ComputeStubbornSet(s)
 2:     stubborn = empty collection
 3:     todo_achievers, todo_dependers= ∅
 4:     seen_for_achievers, seen_for_dependers= ∅

 5:     procedure HandleAction(a, s)
 6:         if a ∉ stubborn then
 7:             stubborn.add(a)
 8:             if a is applicable in s then
 9:                 EnqueueInterferers(a)
10:             else
11:                 ▷ Enqueue a necessary enabling set for a
12:                 atom = an unsatisfied atom from pre(a)
13:                 EnqueueAchievers(atom)

14:     procedure EnqueueAchievers(atom)
15:         if atom ∉ seen_for_achievers then
16:             todo_achievers.add(atom)
17:             seen_for_achievers.add(atom)

18:     procedure EnqueueDependers(atom)
19:         if atom ∉ seen_for_dependers then
20:             todo_dependers.add(atom)
21:             seen_for_dependers.add(atom)

22:     procedure EnqueueInterferers(a)
23:         for atom ∈ pre(a) do
24:             for all siblings atom′ of atom do
25:                 EnqueueAchievers(atom′)
26:         for atom ∈ eff(a) do
27:             for all siblings atom′ of atom do
28:                 EnqueueAchievers(atom′)
29:                 EnqueueDependers(atom′)

30:     atom = some unsatisfied goal atom
31:     EnqueueAchievers(atom)
32:     while todo_achievers is not empty or
33:           todo_dependers is not empty do
34:         if todo_achievers is not empty then
35:             atom = todo_achievers.pop()
36:             for a ∈ A with atom ∈ eff(a) do
37:                 HandleAction(a, s)
38:         else
39:             atom = todo_dependers.pop()
40:             for a ∈ A with atom ∈ pre(a) do
41:                 HandleAction(a, s)
42:     return stubborn
```

Based on these three observations, we propose the atom-centric Algorithm 2. The core idea is to achieve synergy effects by deferring the inclusion of actions in the stubborn set, instead tracking the atoms that characterize them.

We use two collections for this purpose: *todo_achievers* contains atoms for which all achievers should get included in the stubborn set, *todo_dependers* contains atoms for which all actions that depend on the atom should get included.

We ensure that every atom gets added to each of these collections at most once by tracking in sets *seen_for_achievers* and *seen_for_dependers* what has already been included earlier. EnqueueAchievers demonstrates this for *todo_achievers*.

EnqueueInterferers and EnqueueAchievers play exactly the same role as in the action-centric algorithm, with the only difference that they do not directly mark actions stubborn but instead mark the corresponding atoms for further processing. This directly translates to the initialization of the algorithm in lines 30 and 31.

The main loop (lines 33–41) processes the atoms from *todo_achievers* and *todo_dependers*, initiating the previously deferred handling of actions. HandleAction adds the action to the stubborn set and triggers the later inclusion of interfering actions and necessary enabling sets to satisfy C2 and C3.

Theorem 1. *Function* ComputeStubbornSet(s) *returns a strong stubborn set for* s.

Proof sketch. Including all achievers of an unsatisfied goal atom ensures C1. Whenever set *stubborn* is extended in HandleAction, this procedure initiates the inclusion of actions to satisfy C2 and C3. The overall fixed-point computation guarantees that these are indeed included before termination. □

Complexity Analysis

For the analysis, we use $p_{\max}$ as before and $d_{\max}$ for the maximal size of all variable domains.

An efficient implementation of the algorithm precomputes the achievers and dependers of each atom (used in lines 36 and 40) once for the entire search. As discussed in the analysis of the action-centric algorithm, this requires time $O(|\mathcal{A}|p_{\max})$ and space $O(|\mathcal{P}||\mathcal{A}|)$ for storing the result. In contrast to the action-centric algorithm, we do not need to compute and store the interference relation.

With suitable data structures, EnqueueAchievers and EnqueueDependers take constant amortized time. As each outer loop of EnqueueInterferers iterates over at most $p_{\max}$ atoms and the inner loop over all but one atom for each of these variables, the procedure runs in $O(p_{\max}d_{\max})$. Again using t for the variable selection time, it is then easy to see that HandleAction runs in time $O(p_{\max}d_{\max} + t)$ for actions that are not yet contained in *stubborn* and $O(1)$ for actions already contained in *stubborn*.

For the fixed-point iteration, each atom can be inserted into *todo_achievers* at most once and into *todo_dependers* at most once, causing runtime $O(|\mathcal{P}|)$ for all parts of the fixed-point loop except the inner loops (lines 36–37 and 40–41).

The runtime for the inner loops can be bounded by the total time spent inside HandleAction. Every action can

time	Action-centric	Atom-centric
Precomp.	$O(\|\mathcal{A}\|^2 p_{\max})$	$O(\|\mathcal{A}\| p_{\max})$
Per node	$O(\|\mathcal{A}\|^2 + \|\mathcal{A}\|(p_{\max} + t))$	$O(\|\mathcal{A}\|(p_{\max} d_{\max} + t) + \|\mathcal{P}\|)$

space	Action-centric	Atom-centric
Precomp.	$O(\|\mathcal{A}\|^2 + \|\mathcal{P}\|\|\mathcal{A}\|)$	$O(\|\mathcal{P}\|\|\mathcal{A}\|)$
Per node	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{A}\| + \|\mathcal{P}\|)$

Table 1: Overview of complexity results.

only be added to *stubborn* once, giving an upper bound of $O(|\mathcal{A}|(p_{\max} d_{\max} + t))$ for the calls to HANDLEACTION that add actions to the stubborn set. Each other call takes constant time, and we can bound the total number of such calls by $O(|\mathcal{A}||p_{\max}|)$: across the execution of the algorithm, every action is considered in lines 36–37 at most once for each of its effects and in lines 40–41 at most once for each of its preconditions. Hence, the overall runtime of COMPUTESTUBBORN is $O(|\mathcal{A}|(p_{\max} d_{\max} + t)) + |\mathcal{P}|$.

The space complexity for *todo_achievers*, *todo_dependers*, *seen_for_achievers* and *seen_for_dependers* is $O(|\mathcal{P}|)$, for *stubborn* it is $O(|\mathcal{A}|)$.

Table 1 shows an overview of all complexity results. The new algorithm clearly dominates the old one in the time and space requirements for the precomputation. For the actual computation of stubborn sets, the new algorithm needs more space, but only linearly in the number of atoms. In the time requirements the algorithms exhibit a very different profile, which lets us expect that the new atom-centric algorithm works better if variable domains are not too large and the task has many more actions than atoms.

Enhancements

In this section, we discuss two possible enhancements of Algorithm 2. The first one is based on the observation that the algorithm frequently enqueues all siblings of an atom, the second one is a new atom selection strategy.

Shortcut Handling of all Siblings

Since we frequently add all siblings of an atom to one of the queues, we can expect a number of duplicates. Avoiding this overhead should be particularly beneficial if variable domains are large.

From the perspective of a variable, we can track some compact (incomplete) information on what has already been enqueued, for example in *todo_achievers*. For this purpose, we use a datastructure *marked_achieved* that stores for each variable v one of the following values:

- $d \in \mathcal{D}(v)$ representing that all siblings of (v, d) have been enqueued,
- $\top$ representing that all atoms for this variable have been enqueued, or
- $\bot$ representing that we do not have any such information.

The information is incomplete in the sense that we do not track the inclusion of individual atoms, so the value can for example be $\bot$ or some $d \in \mathcal{D}(v)$ although we have already seen all atoms for the variable. To update and exploit the stored information, we do not simply enqueue all siblings of *atom* in lines 24–25 and 27–28 of Algorithm 2 but proceed instead as follows:

```
1: (v, d) = atom
2: if marked_achieved[v] = ⊥ then
3:     for all siblings a of atom do
4:         EnqueueAchievers(a)
5:     marked_achieved[v] = d
6: else if marked_achieved[v] = d' ∉ {d, ⊤} then
7:     EnqueueAchievers((v, d'))
8:     marked_achieved[v] = ⊤
```

If we do not have sufficient information, we add all siblings of *atom* as before, but remember that all values apart from the one from *atom* have been added (lines 2–5). If we know that all atoms (value $\top$) or all siblings of *atom* (value d) have already been added, we do not have to do anything. Otherwise, we add the only missing sibling (whose value is stored in *marked_achieved*$[v]$) and remember that we now have added all atoms (lines 6–8).

We proceed analogously when enqueueing all siblings of an atom with ENQUEUEDEPENDERS in lines 27 and 29.

Atom Selection Strategy

If an action from the stubborn set is inapplicable, we need to choose an unsatisfied atom from the action precondition as seed for the inclusion of a necessary enabling set. Wehrle and Helmert (2014) already discussed and evaluated several strategies for this choice point.

We want to propose a new strategy, called *quick skip*. It is easy to see that if the chosen atom has already been seen (included in *seen_for_achievers*), the algorithm does not enqueue anything within ENQUEUEACHIEVERS. This saves computational effort and—maybe even more importantly—it can potentially lead to more pruning because we do not unnecessarily grow the stubborn set. Therefore, in line 12 of Algorithm 2 the quick skip strategy chooses some atom from *pre*$(a) \cap$ *seen_for_achievers* whenever this set is not empty.

This selection strategy is related to the *static small* and *dynamic small* strategies by Wehrle and Helmert, both of which aim to keep the resulting stubborn set small. The static strategy prefers variables that appear in the effects of fewer actions, the dynamic one prefers atoms with a minimal number of achieving actions that have not yet been included in the stubborn set. Our proposed strategy is closer to *dynamic small* but less specific. If there is an atom for which *all* achieving actions have already been scheduled for inclusion, the strategies are equal. Otherwise, our strategy can be combined with any other strategy, leaving another choice point.

Experimental Evaluation

We implemented the atom-centric algorithm on top of Fast Downward 19.12, which already contains an implementation of the action-centric algorithm (called "simple stubborn sets" there). For the evaluation, we use the benchmarks of all optimal tracks of all International Planning Competitions from 1998 to 2018, amounting to 1827 tasks from 65 domains. Experiments were run on Intel Xeon Silver

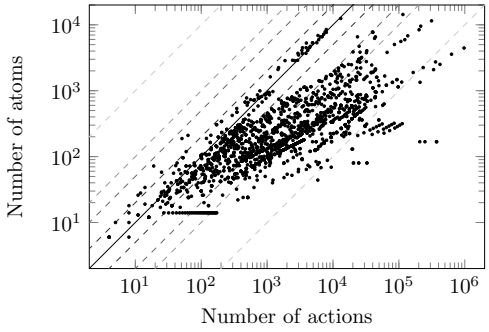


Figure 1: Number of atoms vs. actions for all tasks in the benchmark set. Each mark represents one task. Dashed diagonals show factors 2, 5, 10, and 100.

4114 CPUs using Downward Lab (Seipp et al. 2017). Each planner run is limited to 1800 seconds and 3.5 GiB. The benchmarks, code and experimental data are published online (Röger et al. 2020b).

Before we compare different algorithms and configurations, we evaluate whether the different time complexity of the atom-centric algorithm is promising at all. For this purpose, in Figure 1, we plot the number of atoms against the number of actions in the SAS$^+$ planning tasks produced by Fast Downward. We see that the actions frequently outnumber the atoms, often by several orders of magnitude, so the trade-off looks promising indeed.

Action- vs. Atom-Centric Algorithm

In the first experiment, we examine how the plain action-centric and atom-centric algorithms compare when they compute the same information. Towards this end, we do not use the mutex-based strengthening of interference and use the same strategy for choosing unsatisfied atoms for both algorithms, namely always picking the first unsatisfied atom according to the fixed variable ordering of Fast Downward.

Blind Search With blind search, node expansions are extremely fast, so the relative overhead of computing stubborn sets for each expansion is high. For this reason, we can only expect to benefit from partial order reduction if it leads to significant pruning. On our benchmark set, blind search without pruning solves 710 instances, whereas coverage increases by 26 instances with the atom-centric algorithm (cf. left part of Table 2). Interestingly, computing the same information with the action-centric algorithm leads to a significant coverage decline to 680 instances. As expected, in both cases the total number of expansions is the same and decreases by 17.9% compared to no pruning across all tasks solved by all three configurations.

A closer look at the results per domain reveals that even with our more efficient algorithm, using strong stubborn set pruning is not always beneficial, losing 1–3 tasks in 11 domains and even 5 instances in the freecell domain. The positive net benefit stems from the two parcprinter domains with a coverage increase of 20 and 14 and the two woodworking domains with an increase of 8 and 7 tasks. So it seems that these domains are especially suitable for partial order reduction, whereas in other domains the additional overhead does not pay off. Indeed, in woodworking the goal is to process a set of work pieces, each basically corresponding to an independent subtask. In parcprinter, the aim is to print a set of pages using several components of an involved printing system. The actions for the different pages can often be arbitrarily interleaved, which can be avoided with partial order reduction.

	blind			LM-cut			SCP		
	base	action	atom	base	action	atom	base	action	atom
airport (50)	**22**	21	21	**28**	**28**	**28**	**24**	**24**	**24**
data-network (20)	**7**	6	**7**	**12**	**12**	**12**	**14**	13	**14**
freecell (80)	**20**	9	15	**15**	**15**	**15**	**68**	49	61
hiking (20)	**11**	8	**11**	**9**	**9**	**9**	**14**	11	13
miconic (150)	**55**	50	**55**	**141**	**141**	**141**	143	**144**	**144**
mprime (35)	**19**	18	**19**	**22**	**22**	**22**	**31**	30	**31**
nomystery (20)	**8**	7	**8**	**15**	14	**15**	**20**	**20**	**20**
openstacks-08 (30)	**22**	20	**22**	**22**	20	**22**	**22**	20	**22**
openstacks-11 (20)	**17**	15	**17**	**17**	15	**17**	**17**	15	**17**
org.-synth.-split (20)	**10**	9	9	**16**	15	15	**10**	9	9
parcprinter-08 (30)	10	**30**	**30**	19	**30**	**30**	19	**30**	**30**
parcprinter-11 (20)	6	**20**	**20**	14	**20**	**20**	15	**20**	**20**
parking-11 (20)	**0**	**0**	**0**	**3**	**3**	**3**	**7**	4	**7**
parking-14 (20)	**0**	**0**	**0**	**4**	3	**4**	**6**	4	**6**
pegsol-08 (30)	**27**	**27**	**27**	**29**	28	28	**30**	**30**	**30**
pegsol-11 (20)	**17**	**17**	**17**	**19**	18	18	**20**	**20**	**20**
petri-net-align. (20)	**4**	2	**4**	**9**	**9**	**9**	**0**	**0**	**0**
pipesworld-not. (50)	**17**	14	16	**18**	**18**	**18**	**24**	**24**	**24**
pipesworld-t. (50)	**12**	8	11	**12**	**12**	**12**	**17**	12	16
rovers (40)	6	**7**	**7**	9	**10**	**10**	8	**9**	**9**
satellite (36)	**6**	**6**	**6**	7	**12**	**12**	7	8	**9**
scanalyzer-08 (30)	**12**	8	9	**16**	14	15	**18**	16	**18**
scanalyzer-11 (20)	**9**	5	6	**13**	11	12	**15**	13	**15**
snake (20)	**11**	4	9	**7**	6	**7**	**13**	7	11
spider (20)	**11**	6	9	**11**	**11**	**11**	**16**	13	15
termes (20)	**9**	6	**9**	**6**	5	**6**	**13**	11	**13**
tetris (17)	**9**	6	7	**6**	**6**	5	**11**	9	10
tidybot-11 (20)	**13**	5	12	**14**	**14**	**14**	**15**	13	14
tidybot-14 (20)	**6**	0	4	**9**	8	8	**10**	5	9
transport-11 (20)	**6**	**6**	**6**	**6**	**6**	**6**	**13**	12	**13**
transport-14 (20)	**7**	6	**7**	**6**	**6**	**6**	**9**	8	8
trucks (30)	**6**	5	**6**	**10**	**10**	**10**	**12**	**12**	**12**
woodworking-08 (30)	8	**16**	**16**	18	**27**	**27**	26	**30**	**30**
woodworking-11 (20)	3	**10**	**10**	12	**19**	**19**	19	**20**	**20**
zenotravel (20)	**8**	7	**8**	**13**	**13**	**13**	**13**	**13**	**13**
sum (1088)	414	384	**440**	587	610	**619**	719	678	**727**
other domains (739)	**296**	**296**	**296**	**373**	**373**	**373**	**417**	**417**	**417**
total (1827)	710	680	**736**	960	983	**992**	1136	1095	**1144**

Table 2: Coverage of A* with the blind (left), LM-cut (middle), and SCP (right) heuristics, comparing vanilla search (base) with the addition of plain action-centric (action) and atom-centric (atom) pruning. We highlight maximum coverage separately for each heuristic.

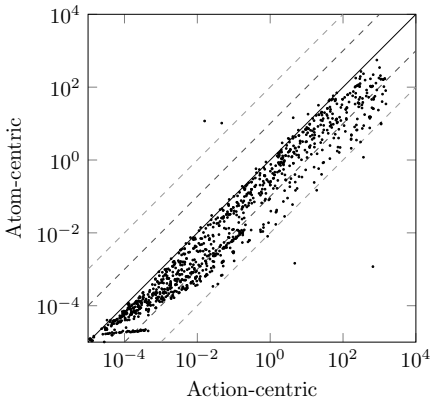


Figure 2: Comparison of pruning time (within an A* search with the SCP heuristic) of the action-centric and the atom-centric algorithm for tasks solved by both approaches. Numbers are in seconds and specify the total time spent computing stubborn sets over all node expansions.

LM-Cut Wehrle and Helmert (2014) used A* search with the LM-Cut heuristic (Helmert and Domshlak 2009) for their evaluation. In this setting, stubborn set pruning is useful overall (cf. middle part of Table 2). 960 tasks are solved without pruning, 983 with the action-centric algorithm and 992 with the atom-centric computation. In a per-domain comparison to the baseline without pruning we never lose more than one task, but coverage increases in six domains. However, this is again most prominent in the parcprinter (+11 and +6) and the woodworking (+9 and +7) domains. The advantage in comparison to the action-centric algorithm stems from eight domains. Conversely, the action-centric variant only solves one more instance in tetris.

Saturated Cost Partitioning We also conducted an analogous experiment for A* with a saturated cost partitioning (SCP) heuristic (Seipp, Keller, and Helmert 2020) over pattern databases (Edelkamp 2001) and Cartesian abstractions (Seipp and Helmert 2018). The pattern databases were generated systematically up to pattern size 2 and via hill climbing (Haslum et al. 2007). This SCP heuristic yields state-of-the-art performance for optimal classical planning, because it is both accurate and very fast to evaluate (much faster than LM-cut, for example).

Similarly to the results for blind search, using the action-centric algorithm decreases coverage (cf. right part of Table 2). In contrast to the results for LM-cut, using the atom-centric algorithm increases the total coverage by only 2 tasks, with a decrease by 1 task in 8 domains, by 2 in snake and even by 7 in freecell. However, if we compare the action-centric against the atom-centric algorithm, we see a clear advantage of the new one in 19 domains, while the opposite is never the case.

Overall As both stubborn set algorithms compute the same information, the difference in performance must be attributed to the different computational overhead. Figure 2 compares the total time spent for computing stubborn sets for each task. This data stems from the experiment with the SCP heuristic; the plots for the other configurations (blind and LM-Cut) look similar. We see that with the atom-centric algorithm we can obtain the same pruning power much faster, often by more than an order of magnitude.

	ds		ss		FD		qs	
		+sib		+sib		+sib		+sib
airport (50)	**25**	**25**	24	24	24	24	24	24
data-network (20)	13	13	**14**	**14**	**14**	**14**	**14**	**14**
freecell (80)	42	43	60	60	**61**	**61**	59	60
ged (20)	15	15	**19**	**19**	**19**	**19**	**19**	**19**
hiking (20)	11	11	12	12	**13**	**13**	12	12
logistics98 (35)	12	12	12	12	12	12	**13**	**13**
miconic (150)	**144**	143	**144**	**144**	**144**	**144**	143	**144**
mprime (35)	30	30	30	30	**31**	30	30	30
mystery (30)	18	18	**19**	**19**	**19**	**19**	**19**	**19**
openstacks-08 (30)	20	20	22	22	22	22	**23**	**23**
openstacks-11 (20)	15	15	17	17	17	17	**18**	**18**
openstacks-14 (20)	3	3	3	3	3	3	**4**	**4**
parcprinter-08 (30)	**30**	**30**	28	28	**30**	**30**	**30**	**30**
parcprinter-11 (20)	**20**	**20**	18	18	**20**	**20**	**20**	**20**
parking-11 (20)	4	4	**7**	**7**	**7**	**7**	**7**	**7**
parking-14 (20)	4	4	**6**	**6**	**6**	**6**	**6**	**6**
pipesworld-not. (50)	21	21	**24**	**24**	**24**	**24**	**24**	**24**
pipesworld-t. (50)	11	11	13	13	**16**	**16**	14	**16**
rovers (40)	**11**	**11**	10	10	9	9	9	9
scanalyzer-08 (30)	15	15	**18**	**18**	**18**	**18**	**18**	**18**
scanalyzer-11 (20)	12	12	**15**	**15**	**15**	**15**	**15**	**15**
snake (20)	4	4	10	10	**11**	**11**	10	**11**
spider (20)	12	12	14	14	**15**	**15**	**15**	**15**
termes (20)	10	10	12	12	**13**	**13**	**13**	**13**
tetris (17)	8	8	**10**	**10**	**10**	**10**	**10**	**10**
tidybot-11 (20)	12	13	**14**	**14**	**14**	**14**	**14**	**14**
tidybot-14 (20)	4	4	8	8	**9**	**9**	**9**	**9**
transport (20)	10	10	**13**	**13**	**13**	**13**	**13**	**13**
sum (927)	536	537	596	596	609	608	605	**610**
other domains (900)	**535**	**535**	**535**	**535**	**535**	**535**	**535**	**535**
total (1827)	1071	1072	1131	1131	1144	1143	1140	**1145**

Table 3: Coverage of A* with the SCP heuristic, comparing atom selection strategies *ds*, *ss*, *FD*, and *qs*, and the combination with shortcut handling of siblings (*sib*).

Enhancements to the Action-Centric Algorithm

In the second experiment, we evaluate the two enhancements to the atom-centric algorithm. The shortcut handling of siblings (*sib*) does not change the behavior of the algorithm and should hence only have an impact on runtime. The new atom selection strategy quick skip (*qs*) should have a tendency to produce smaller stubborn sets, so we would also hope for more pruning. We compare the *qs* strategy to the default strategy (*FD*) and the two strategies dynamic small (*ds*) and static small (*ss*) by Wehrle and Helmert (2014), all of them with and without the *sib* enhancement.

Table 3 shows that with the SCP heuristic, the dynamic small and static small strategies solve many fewer tasks than

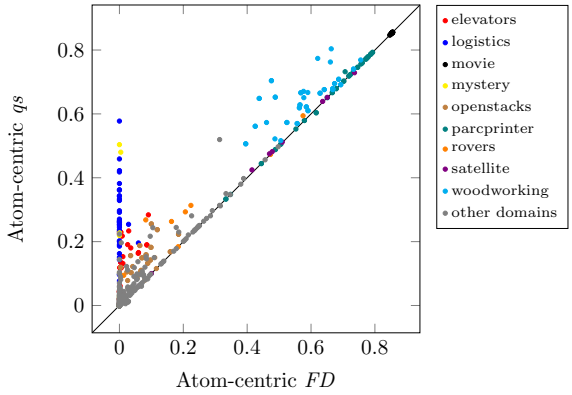


Figure 3: Comparison of pruning ratio of the atom-centric algorithm with strategies *FD* and *qs*, using A* with SCP.

the Fast Downward and quick skip strategies. While the Fast Downward strategy solves 4 more tasks in total than quick skip, quick skip is better than Fast Downward with the LM-cut and blind heuristics (+3 and +2, not shown in Table 3). The shortcut handling of siblings only has a very mild impact on coverage, sometimes negative, sometimes positive, except for the new strategy quick skip where it is exclusively beneficial (except for transport-14 with the blind heuristic, not shown in Table 3). The combination of quick skip with shortcut handling of siblings achieves the highest total coverage with all three heuristics, and dominates the other strategies also in a per-domain comparison, except for 7 domains when using blind search, 3 domains when using A* with LM-cut, and 5 domains when using A* with SCP.

To analyze the pruning ratio of the different methods, we run the search with pruning and accumulate the number of successors of all expanded states as n_{all} and sum up the size of the corresponding stubborn sets as n_{gen}. The pruning ratio is then defined as $1 - n_{\text{gen}}/n_{\text{all}}$, giving values between 0 and 1, where 0 represents no pruning and 1 would mean that all successors were pruned. Figure 3 plots the pruning ratio of the atom-centric algorithm with the *FD* strategy (the best previous selection strategy according to Table 3) to the new quick skip strategy (both with SCP), highlighting domains with larger differences. We observe consistent positive impact on the pruning power, which is particularly pronounced in the logistics and woodworking domains.

Comparison to the State of the Art

In the third experiment, we compare our best configuration (atom-centric algorithm with *qs* and *sib*) to the configuration reported as state of the art for computing strong stubborn sets by Wehrle and Helmert (2014), namely "SSS-EC full/mutex" (EC), which computes stubborn sets in a way that dominates the expansion core method (Chen and Yao 2009) and enhances action interference with mutexes. With our best configuration, total coverage increases significantly for all three heuristics, in particular for the two faster-to-compute ones (+51 with blind search, +9 with LM-Cut, +38 with SCP). A deeper analysis reveals that our configuration

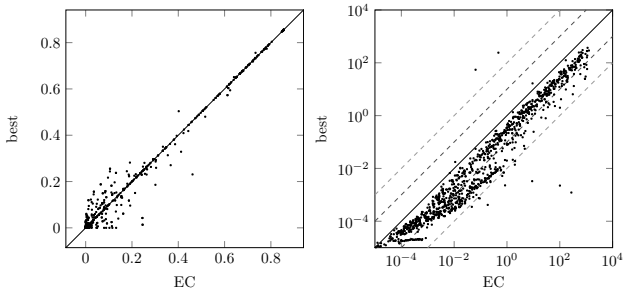


Figure 4: Comparison of pruning ratio (left) and pruning time (right) of EC vs. our best configuration with SCP.

is on-par with respect to pruning power, but requires a much lower computation time. To illustrate this, we compare the pruning ratio and pruning time for A* with SCP in Figure 4. For the other heuristics the results look qualitatively similar.

Since not all domains are equally suited for partial-order reduction, many recent IPC planners (e.g., Alkhazraji et al. 2014) disable pruning if after 1000 expanded states the pruning ratio is at most 20%. We evaluated the impact of this approach on our best configuration (atom-centric algorithm with both enhancements) and on the previous state of the art (EC). The method has only a mild impact on our algorithm: overall, coverage increases, but in some domains fewer tasks are solved. This is very different for the slower EC method, which greatly benefits from this approach, bringing it almost on par with our configuration.

Discussion and Future Work

We proposed an atom-centric algorithm that computes the same stubborn sets as an earlier action-centric algorithm with a different profile time and space complexity profile. The new algorithm requires less space, and we saw that it is much faster on common planning benchmarks. One limitation of our algorithm is that it is no longer possible to enhance the interference relation with mutex information. However, already without any enhancements, our algorithm outperforms the best previous algorithm (EC), which makes use of such mutex information (1145 vs. 1107 solved tasks with A* + SCP). The new atom selection strategy *quick skip* does not only further speed up the computation but also leads to smaller stubborn sets and thus to more pruning.

In classical planning, stubborn sets have not only been used for state-space search but have also been adapted for fork-decoupled search (Gnad, Hoffmann, and Wehrle 2019). Beyond the classical planning fragment, they have been applied to fully observable non-deterministic planning (Winterer et al. 2017), planning with resources (Wilhelm, Steinmetz, and Hoffmann 2018) as well as for goal recognition design (Keren, Gal, and Karpas 2018). In future work, it would be interesting to examine whether the general idea of an atom-centric perspective can also be beneficially applied in these settings.

Acknowledgments

We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639).

References

Al-Khazraji, Y. 2017. *Analysis of Partial Order Reduction Techniques for Automated Planning*. Ph.D. Dissertation, University of Freiburg.

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A stubborn set algorithm for optimal planning. In *Proc. ECAI 2012*, 891–892.

Alkhazraji, Y.; Katz, M.; Mattmüller, R.; Pommerening, F.; Shleyfman, A.; and Wehrle, M. 2014. Metis: Arming Fast Downward with pruning and incremental computation. In *IPC-8 planner abstracts*, 88–92.

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

Chen, Y., and Yao, G. 2009. Completeness and optimality preserving reduction for planning. In *Proc. IJCAI 2009*, 1659–1664.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 84–90.

Gnad, D.; Hoffmann, J.; and Wehrle, M. 2019. Strong stubborn set pruning for star-topology decoupled state space search. *JAIR* 65:343–392.

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.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proc. AAAI 2008*, 944–949.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Keren, S.; Gal, A.; and Karpas, E. 2018. Strong stubborn sets for efficient goal recognition design. In *Proc. ICAPS 2018*, 141–149.

Röger, G.; Helmert, M.; Seipp, J.; and Sievers, S. 2020a. An atom-centric perspective on stubborn sets. In *Proc. SoCS 2020*, 57–65.

Röger, G.; Helmert, M.; Seipp, J.; and Sievers, S. 2020b. Code, benchmarks and experiment data for the SoCS 2020 paper "An Atom-Centric Perspective on Stubborn Sets". https://doi.org/10.5281/zenodo.3744571.

Seipp, J., and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *JAIR* 62:535–577.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated cost partitioning for optimal classical planning. *JAIR* 67:129–167.

Valmari, A. 1989. Stubborn sets for reduced state space generation. In *Proc. APN 1989*, 491–515.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In *Proc. ICAPS 2012*, 297–305.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proc. ICAPS 2014*, 323–331.

Wehrle, M.; Helmert, M.; Alkhazraji, Y.; and Mattmüller, R. 2013. The relative pruning power of strong stubborn sets and expansion core. In *Proc. ICAPS 2013*, 251–259.

Wilhelm, A.; Steinmetz, M.; and Hoffmann, J. 2018. On stubborn sets and planning with resources. In *Proc. ICAPS 2018*, 288–297.

Winterer, D.; Alkhazraji, Y.; Katz, M.; and Wehrle, M. 2017. Stubborn sets for fully observable nondeterministic planning. In *Proc. ICAPS 2017*, 330–338.

Simplified Planner Selection

Patrick Ferber[1,2]
patrick.ferber@unibas.ch

University of Basel[1]
Switzerland

Saarland Informatics Campus[2]
Saarland University
Germany

Abstract

There exists no planning algorithm that outperforms all others. Therefore, it is important to know which algorithm works well on a task. A recently published approach uses either image or graph convolutional neural networks to solve this problem and achieves top performance. Especially the transformation from the task to an image ignores a lot of information. Thus, we would like to know what the network is learning and if this is reasonable. As this is currently not possible, we take one step back. We identify a small set of simple graph features and show that elementary and interpretable machine learning techniques can use those features to outperform the neural network based approach. Furthermore, we evaluate the importance of those features and verify that the performance of our approach is robust to changes in the training and test data.

1 Introduction

Planning is concerned with finding a sequence of actions that leads from some initial state to a goal. Over the last decades researchers invented a zoo of different algorithms. All of those exhibiting different strengths and weaknesses. No single algorithm excels on all tasks. To combine the strengths of multiple planning algorithms, in the further course called planners, the idea of having a portfolio of planners to solve a task has emerged. The most common type of portfolios is based on the idea that a planner solves a task either quickly or not at all. Thus, if a planner is not quickly finding a solution, then we could try another planner. Those portfolios posses a set of complementing planners and for each planner they predefined how long the planner runs and in which order they are executed (Helmert et al. 2011; Seipp et al. 2015; Howe et al. 1999). The disadvantage of this approach is that it splits the available time among the planners in its portfolio. It can happen that for some tasks no planners in a portfolio can quickly solve the task. In this case, it would be better to choose the single planner with the highest chance to solve the task and let it run for all the available time. A second portfolio approach has a collection of planners and predicts for a given task how long each planner requires for solving the task or how confident the model is that a planner will solve the task. Then, a single planner is selected and executed. The main obstacle in this approach is finding a good representation of the task for the predicting model. Fawcett et al. (2014) gathered a large set of handcrafted features. They trained models on those features to predict for some planners how long the planners require to solve a given task. To avoid handcrafting features and potentially ignoring important features Sievers et al. (2019a) translated a given task into a graph which can potentially be translated back into the original task. They interpreted the adjacency matrix of the graph as image, scale the image down to 128x128 pixels, and train a convolutional neural network (CNN) to predict which planner will solve the given task. The idea is that the neural network detects automatically good features. The success of their approach is astonishing. The interpretation of the graph as an 128x128 pixel image ignores a lot of information. Many entries of the adjacency matrix are combined in the same pixel and the information which type a node has is discarded. The success of their CNN shows that the remaining information in the image are sufficient for the planner selection. In a succeeding paper, the input transformation from a graph to an image was eliminated by using graph convolution networks (GCN) (Kipf and Welling 2017) and feeding the graph directly into the neural network (Ma et al. 2020). This caused a modest performance improvement and implies that the images contains sufficient information.

The obvious questions are: What could be the features the neural networks are learning from these graphs resp. images? Can we use those features in combination with simpler machine learning techniques and achieve similar or even better performance? Neural networks are complex function approximators (Cybenko 1989). Today, we are not able to understand which features they have learned. We modify the questions and show that we can find simple features in the graphs such that simple machine learning techniques outperform the neural network based approach. We evaluate how important our features are and verify that our models are robust to changes in the training and test data.

The rest of the paper is structured as follows. Section 2 provides background on planning and on the graph encodings we are working with. Section 3 explains the training setup of our experiments. The experiments and their results are described in section 4 and we summarize our findings in section 5.

2 Background

A PDDL task Π^{PDDL} (McDermott et al. 1998) is defined as a tuple $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$. $\mathcal{P}$ is a set of first-order predicates. $\mathcal{A}$ is a set of action schemas. Σ^C is a set of constant objects, Σ^O is a set of non-constant objects. $\Sigma = \Sigma^C \cup \Sigma^O$. An action schema $a \in \mathcal{A}$ with the parameters $p_1, ..., p_n$ is a triple (pre_a, add_a, del_a) with $pre_a \subseteq \mathcal{P}$, $add_a \subseteq \mathcal{P}$, and $del_a \subseteq \mathcal{P}$ and all variables in pre_a, add_a, and del_a are in $\Sigma^C \cup \{p_1, ..., p_n\}$. I is the initial state and G the goal. Both are sets of atomic statements over $\mathcal{P}$ using Σ.

Any PDDL task can be translated into a SAS^+ task (Bäckström and Nebel 1995). A SAS^+ task is a tuple $(\mathcal{V}, \mathcal{A}, s_0, s_*)$ with $\mathcal{V}$ is a set of variables. Each variable $v \in \mathcal{V}$ has a finite domain $dom(v)$. A fact is a pair (v, v') with $v \in \mathcal{V}$ and $v' \in dom(v)$. A *state* assigns every variable $v \in \mathcal{V}$ a value from its domain. A partial assignment assigns every variable in a subset of $\mathcal{V}$ a value from their domains. $\mathcal{A}$ is the set of actions with each action $a \in \mathcal{A}$ is a pair (pre_a, eff_a) and pre_a, eff_a are partial states. s_0 is a state which is called the initial state, and s_* is a partial assignment which is called the goal.

We use the previous formalisms to define graph encodings that describe a given task. The problem description graph (PDG) (Pochter, Zohar, and Rosenschein 2011) is an undirected graph which contains for every fact and variable one node and for every action two nodes (one representing the preconditions and one representing the effects of the action). For every value in the domain of a variable the associated node is connected to the node of the variable. Every precondition and effect node associated to the same operator are connected. The nodes associated to the facts in the precondition of an action are connected to the precondition node of an action. The same is done for the facts in the effect of an action. Additionally, we add two special nodes, one is connected to all nodes representing facts that are true in the initial state; the other is connected to all nodes representing facts that are true in the goal.

The second encoding is based on the abstract structure graph (ASG) of the task (Sievers et al. 2019b). An abstract structure is defined as either a symbol (specific elements are symbols), as a list of abstract structures, or as a set of abstract structures. For a given PDDL file, each object and variables becomes a symbol, but also further elements of the PDDL task become symbols. The more complex parts of the PDDL are constructed from those symbols. E.g. the fact *on(a, b)* is constructed as a list containing the symbols *on*, *a*, and *b*. The abstract structure of a PDDL file is written as graph by creating a node for every symbol and every abstract structure and adding a directed edge $X \rightarrow Y$ between two nodes *X* and *Y* if X requires for its definition Y. A PDDL description of a planning task can be directly translated into an ASG and the ASG can be translated back into the same PDDL description.

3 Training

To make our results comparable Ma et al. (2020), we perform all experiments on the data set published by Ferber et al. (2019) which extends the data produced by Katz et al. (2018). Our code, new data sets, and experimental results are online available(Ferber 2020).

The data set contains tasks from the classical planning tracks of the International Planning Competitions (IPC) until 2018. Additionally, it includes the domains *BRIEFCASE-WORLD*, *FERRY*, and *HANOI* from the IPP benchmark collection (Köhler 1999), the domain *GEDP* (Haslum 2011), domains from the conformant-to-classical planning compilation (T0) (Palacios and Geffner 2009), and the domain *FSC* (Bonet, Palacios, and Geffner 2009).

For each task the runtimes of 29 optimal planners are measured. The measurements were limited to 30 minutes and at most 7744MiB of memory. We restrict ourselves to the subset of 17 planners that Ma et al. (2020) used. Those are 16 Fast Downward (Helmert 2006) configurations. All configurations are using A* search (Hart, Nilsson, and Raphael 1968) and strong stubborn sets (Wehrle and Helmert 2014). Each of the following eight heuristics is used twice, once with structural symmetries pruning (Shleyfman et al. 2015) using DKS (Domshlak, Katz, and Shleyfman 2012) and once with structural symmetries pruning using orbital space search (OSS) (Domshlak, Katz, and Shleyfman 2015): blind heuristic, LM-cut (Helmert and Domshlak 2009), iPDB (Haslum et al. 2007), a zero-one cost partitioning pattern data base (01-PDB) using a genetic algorithm to compute the pattern (Edelkamp 2006), and four Merge & Shrink (M&S) heuristics (Dräger, Finkbeiner, and Podelski 2006; Helmert et al. 2014; Sievers 2017) using bisimulation (BS) (Nissim, Hoffmann, and Helmert 2011), full pruning (Sievers 2017), Θ-combinability (Sievers, Wehrle, and Helmert 2014), partial abstractions (Sievers 2018), DFP (Sievers, Wehrle, and Helmert 2014), and merging based on either *strongly connected components* (SCC) of the causal graph (Sievers, Wehrle, and Helmert 2016), *maximum intermediate abstraction size minimizing* (MIASM) (Fan, Müller, and Holte 2014), or *score-based MIASM* (sbMIASM) (Sievers, Wehrle, and Helmert 2016). The 17th planner is SymBA* (Torralba et al. 2014). Every planner except for 2 M&S configurations use h^2 mutex detection (Alcázar and Torralba 2015).

We removed all tasks from the data set that were not solved by any of the selected planners. 2,439 tasks remain; 145 of those tasks belong to the IPC 2018. For each task the data set contains its PDG, called *grounded*, and its encoding as ASG, called *lifted*. Because our machine learning techniques do not work on graphs, we extract the following 21 basic properties from every graph:

- the number of nodes
- the number of edges
- the graph density ($\frac{\#edges}{\#nodes*(\#nodes-1)}$)
- the number of connected components
- the size of the largest connected component
- the minimum, mean, median, and maximum eccentricity of its nodes (the eccentricity of a node is its maximum distance to any other node; the minimum eccentricity of a graph is called radius, the maximum eccentricity is called diameter)

	LR					RF	MLP		Delfi	
	0	0.1	1	2	5	50	3	5	CNN	GNN
Binary	57.0(0.8)	86.2(0.0)	82.1(0.0)	84.8(0.0)	88.3(0.0)	69.9(4.3)	76.6(8.2)	77.4(8.2)	73.1	80.7
Log	62.8(0.0)	67.6(0.0)	**89.0(0.0)**	80.7(0.0)	81.4(0.0)	66.6(2.4)	64.8(0.0)	64.2(1.9)	–	–
Time	56.4(0.7)	55.2(0.0)	55.2(0.0)	52.4(0.0)	55.2(0.0)	72.1(3.1)	68.3(4.6)	67.4(2.0)	–	–

	LR					RF	MLP		Delfi	
	0	0.1	1	2	5	50	3	5	CNN	GNN
Binary	65.5(0.0)	66.2(0.0)	70.3(0.0)	64.8(0.0)	61.4(0.0)	70.9(4.5)	61.4(0.0)	61.4(0.0)	86.9	87.6
Log	58.6(0.0)	69.7(0.0)	69.7(0.0)	69.7(0.0)	70.3(0.0)	73.7(3.5)	65.2(1.0)	64.8(0.0)	–	–
Time	65.5(0.0)	74.5(0.0)	71.0(0.0)	69.7(0.0)	70.3(0.0)	**79.6(5.3)**	67.9(5.9)	70.3(4.6)	–	–

Table 1: Mean coverage and standard deviation (in %) on the IPC 2018 tasks which are solved by at least one planner. Linear regression (LR) uses L1 regularization weights from 0 to 5; random forest (RF) have 50 trees; and the multi-layer perceptrons (MLP) have 3 resp. 5 hidden layers. The last column shows the published performance of the image (CNN) resp. graph (GCN) based versions of Delfi on binary labels. Top: Performance on the grounded graphs. Bottom: Performance on the lifted graphs.

- the minimum, mean, median, and maximum degree of its nodes
- the minimum, mean, median, and maximum in-degree of its nodes
- the minimum, mean, median, and maximum out-degree of its nodes

We selected those properties because they are easy to understand and fast to compute. The values of some properties can greatly differ, e. g. the number of nodes in the grounded graphs vary between 6 and 87,000. Thus, we augment our set of features, by adding the logarithm of each property (given that the property is always non-zero) and by normalizing each property into the range of 0 to 1. Finally, for every graph we obtain a feature vector with 60 elements. For the runtime, we have the same issues as we had for some properties. The scale of the runtime can vary between fractions of a second and up to half an hour. Therefore, we train the models with three different label transformations: With the original runtime, with the logarithm of the runtime, and with the binary information whether a planner was able to solve a task within the resource limits.

We train plain linear regression models (Galton 1886) and models with L1 regularization (Tibshirani 1996). Linear regression learns for every feature (each property and their transformations) a weight. The output is the weighted sum of the features. L1 regularization adds the L1 norm of the weights as penalty to the optimization process. This causes unnecessary large feature weights to decrease and can filter out irrelevant features. The L1 norm can be scaled with a parameter to make the filtering weaker or stronger.

Second, we train random forests (Breiman 2001). Those are ensembles of decision trees. During training, we optimize each decision tree individually. The final output of the random forest is an averaged decision over all its trees.

The last kind of models we train are multi-layer perceptron. Those are simple neural network consisting of multiple layers of neurons. Each layer is densely connected to the next layer. The value for each neuron is the weighted sum of the neurons connected to it (c. f. linear regression). The value of the neuron is modified by a non-linear function (e. g. $ReLU(x) = max(0, x)$) and is forwarded to the next neurons. We use the Adam optimizer (Kingma and Ba 2015) with a learning rate of 0.001 to optimize the weights of the network.

4 Experiments

First, we evaluate how good our simple techniques are at choosing a single planner to solve a given task and compare our results to previous work. Then, we investigate which features have been used and how important those features are. Next, we examine which planners were actually chosen by our models, and we end by evaluating whether our techniques are robust to changes in the data.

One of our feature augmentations normalizes the values of the graph properties. The test data was not used for finding the normalization parameters. All training configurations are run 10 times and their mean coverage as well as their standard deviation are reported. The experiments are run with 3 GB of memory on a single core of an Intel Xeon E5-2660 CPU. The linear regression models finished training in at most 13 seconds, the random forest models in at most 48 seconds, and the neural networks in at most 20 minutes.

Performance on IPC 2018 Tasks

First, we evaluate how good simple machine learning techniques are at choosing a planner to solve a given task. Like Ma et al. (2020) we use the tasks from the IPC 2018 as test data and all other tasks for training. Neither linear regression nor random forests support validation data, thus, we do not use validation data for the multi-layer perceptrons either.

We train 5 linear regression configurations with L1 regularization weights from 0 to 5, a single random forest with 50 trees, and 2 neural network configurations with 3 resp. 5

hidden layers and 30 neurons in each layer. We use the *sigmoid* activation function and the *cross-entropy* loss to train the networks which make binary decisions. For all other networks we use the *ReLU* activation function and the *mean squared error* loss.

Table 1 shows the performance of all models on the features of the grounded (top) and lifted (bottom) graphs. The two simplest baselines are selecting a random planner for each task which has a coverage of 60.6% and selecting always the planner which performed best on the training data which has a coverage of 64.8%. Most of our trained models outperform both of those baselines. This shows that even simple models can learn useful information. In the grounded setting, linear regression outperforms all other techniques if it is trained on binary or logarithmically transformed labels; on the true labels it is not able to learn anything and performs even worse than the random baseline. Notable, linear regression with our features is even outperforming the Delfi baselines. This does not mean that Delfi is approximating our features, but, it shows that even simple machine learning techniques with understandable features obtain top performances.

The lifted setting is more difficult for linear regression. Its performance is worse in general. In this setting our best performing models are random forests, but even those are not able to outperform the Delfi baselines on the lifted graphs. This means the neural networks of Delfi on the lifted graphs are able to exploit some features that we do not know about. It is an advantage of Delfi that the user does not need to define a set of properties.

Feature Reduction

Now that we have well performing models, the questions arise which features are required by the models and how important are those features? The answers help us to understand which properties of the graphs describe useful information and which properties can be skipped to speed up the predictions.

Linear regression models allow us to easily investigate their learned weights, thus, we will take a look into the best performing linear regression models for grounded and for the lifted graphs. Additionally, we add the best grounded configuration with binary labels and the lifted configuration with logarithmically transformed labels and and L1 weight of 1 to the comparison. We cannot interpret the magnitude of a weight as importance, because the magnitude of our features varies greatly. Instead, for every feature we sum up how often it has been used by the models. For each configuration we have trained 10 models and each model has internally one linear regression model for each of the 17 planners. Thus, the maximum number of times a feature can be used is 170. The more frequently a feature has been used the more beneficial we can expect it to be. Table 2 shows those sums. Our first observation is that many configurations do not use any normalized feature and rarely use a logarithmically scaled feature. Those transformed features are good candidates to be exclude from training to speed up the predictions.

Upon closer inspection we see that the more precise we

	Grounded		Lifted	
Features	Binary	Log	Log	Time
#nodes	0	170	170	170
#edges	170	170	170	170
density	0	0	0	0
#conn. comp.	0	0	0	170
max(|conn. comp|)	170	170	170	170
radius	80	150	170	170
mean eccentricity	50	0	160	170
median eccentricity	20	20	40	170
diameter	50	20	110	170
min. degree	0	0	0	142
mean degree	0	20	0	169
median degree	0	0	0	117
max. degree	110	170	160	170
min. in-degree	0	0	0	0
mean in-degree	0	0	0	79
median in-degree	0	0	0	0
max. in-degree	40	150	160	170
min. out-degree	0	0	0	0
mean out-degree	0	0	0	31
median out-degree	0	0	0	140
max. out-degree	130	140	170	170
log(#nodes)	0	0	0	170
log(#edges)	0	20	0	140
log(density)	0	50	0	160
log(#conn. comp.)	0	0	0	160
log(max(|conn. comp|))	0	140	0	160
log(radius)	0	0	0	170
log(mean eccentricity)	0	0	0	170
log(median eccentricity)	0	0	0	170
log(diameter)	0	0	0	170
log(min. degree)	0	0	0	140
log(mean degree)	0	0	0	90
log(median degree)	0	0	0	170
log(max. degree)	0	0	0	140
log(max. in-degree)	0	0	0	169
log(mean out-degree)	0	0	0	50
log(median out-degree)	0	0	0	140
log(max. out-degree)	0	10	140	170
norm(#nodes)	0	0	0	66
norm(#edges)	0	0	0	170
norm(density)	0	0	0	170
norm(#conn. comp.)	0	0	0	120
norm(max(|conn. comp|))	0	0	0	104
norm(radius)	0	0	0	152
norm(mean eccentricity)	0	0	0	102
norm(median eccentricity)	0	0	0	152
norm(diameter)	0	0	0	152
norm(min. degree)	0	0	0	170
norm(mean degree)	0	0	0	170
norm(median degree)	0	0	0	170
norm(max. degree)	0	0	0	110
norm(mean in-degree)	0	0	0	110
norm(max. in-degree)	0	0	0	152
norm(mean out-degree)	0	0	0	118
norm(median out-degree)	0	0	0	137
norm(max. out-degree)	0	0	0	10

Table 2: Feature usages for linear regression configurations. The lifted log configuration uses the same L1 weight as the grounded log configuration. All other configurations use their best L1 weight. Four unused features are omitted.

Features	Grounded	Lifted
#nodes	1	A
#edges	2	A
density	3	B
#conn. comp.	4	C
max(|conn. comp|)	1	A
radius	5	A
mean ecc.	5	A
median ecc.	5	A
diameter	5	A
min. deg.	6	D
mean deg.	7	E
median deg.	8	F
max. deg.	9	G
min. indeg.	10	H
mean indeg.	7	E
median indeg.	11	I
max. indeg.	12	G
min. outdeg.	13	J
mean outdeg.	7	E
median outdeg.	14	K
max. outdeg.	9	G

Table 3: Groups of features with a high (> 0.95) positive or negative Pearson Correlation.

Feature Group	Grounded	Feature Group	Lifted
1	-2.8%	A	9.0%
2	-6.2%	B	-0.7%
3	0.0%	C	0.0%
4	0.0%	D	0.0%
5	0.0%	E	-5.5%
6	0.0%	F	-1.4%
7	-2.8%	G	-4.8%
8	0.0%	H	0.0%
9	-18.6%	I	0.0%
10	0.0%	J	0.0%
11	0.0%	K	0.0%
12	-4.8%		
13	0.0%		
14	0.0%		
Baseline	89.0%		74.5%

Table 4: Performance degradation for the best grounded and lifted linear regression configuration, if a group of highly correlated features is removed.

want to predict the runtimes, the more features the linear regression is using. For the prediction of a binary label the models do not use any transformed feature. To predict the logarithm of the runtimes, some transformed features are used. And to predict the actual runtime, almost all features are used. This trend could be seen in multiple configurations and is independent of using the grounded or the lifted encoding.

A final, less obvious observation is that there are some groups of features for which a trained regressor is only selecting some members of each group. This can be especially well seen with the features *radius*, *mean eccentricity*, *median eccentricity*, and *diameter*. Experiments have shown that removing one of those features has close to no impact on the performance. It turned out that some properties of the graph are strongly linearly correlated. We calculated for each pair of features their Pearson correlation and grouped features together which have an absolute Pearson correlation greater than 0.95. Table 3 shows for both encodings which features are grouped together.

To understand how important each feature group is for the final performance, we retrain the models but withhold a single feature group. Table 4 shows how much the performance of an model changes if a feature group is left out. For the grounded graphs, the most important feature is by far the maximum degree in the graph, the second most important feature is the number of edges. For the lifted graphs, the features 'mean degree', 'mean in-degree', and 'mean out-degree' are the most important. But nearly as important are the features 'maximum degree' and 'maximum out-degree'. Some features groups can be removed without any impact on the test performance and removing feature group 'A' even improves the coverage. Ideally, the L1 regularization would assign those features a weight of zero. This might not happen for two reasons. First, the test tasks - from the IPC 2018 - come from a different data distribution than the training tasks. Thus, features useful for the training tasks might not be useful on the test tasks. Secondly, the loss optimized by the linear regression is not the metric we are comparing. The models try to optimize their prediction for every planner, we are only interested in selecting a single planner to solve the task.

Planner choices

To better understand how the models obtain peak performance, we examine which planners are chosen. We want to understand whether those models have learned to choose the right planner for a task. The models do not predict planners at random, because their coverage is not close to the random baseline.

Table 5 shows how often a planner was selected by the best grounded and the best lifted linear regression model. For each planner it shows additionally their coverage on all test tasks (Cov_T) and their coverage on those tasks for which they were selected (Cov_C). For both configurations we see that the models chose their predictions from a subgroup of (mostly) good planners. We have trained 15 linear regression configuration for the grounded and again 15 configurations for the lifted graph encoding. Could it be that by chance

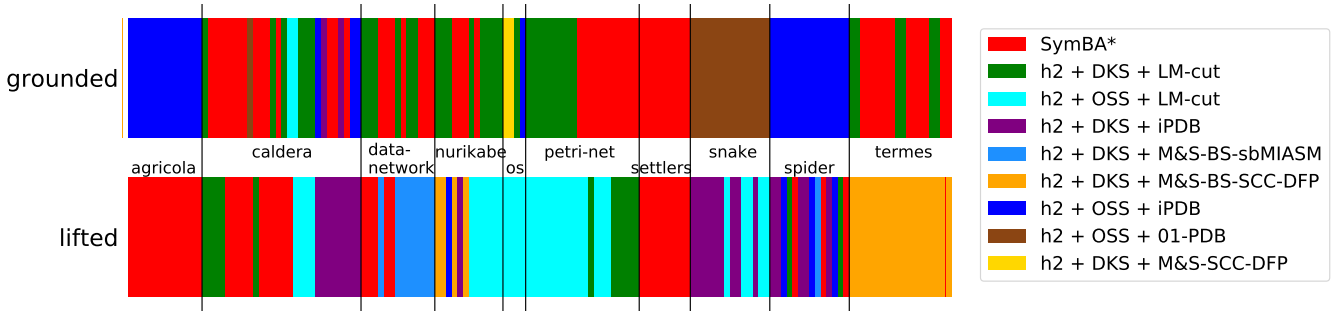


Figure 1: Shows for each task in the test data which planner was selected by the best linear regression configuration for the lifted and grounded graphs. All tasks are sorted by domains and within their domains by name.

Usage	Cov_T	Cov_C	Planner (grounded)
39.3%	82.1%	84.7%	SymBA*
24.8%	64.8%	91.7%	h^2 + DKS + LM-cut
21.4%	70.3%	80.6%	h^2 + OSS + iPDB
10.3%	59.3%	80.0%	h^2 + OSS + 01-PDB
1.4%	70.3%	100.0%	h^2 + DKS + iPDB
1.4%	52.4%	100.0%	h^2 + DKS + M&S-SSC-DFP
1.4%	64.8%	50.0%	h^2 + OSS + LM-cut

Usage	Cov_T	Cov_C	Planner (lifted)
28.3%	82.1%	73.2%	SymBA*
22.5%	64.8%	66.3%	h^2 + OSS + LM-cut
15.9%	70.3%	100.0%	h^2 + DKS + iPDB
15.2%	65.5%	63.6%	h^2 + DKS + M&S-BS-SCC-DFP
9.2%	64.8%	62.7%	h^2 + DKS + LM-cut
6.2%	69.7%	77.8%	h^2 + DKS + M&S-BS-sbMIASM
2.8%	70.3%	100.0%	h^2 + OSS + iPDB

Table 5: Distribution of how often a planner was selected (Usage), the fraction of tasks, the planner solves from the test tasks (Cov_B), and the fraction of tasks the planner solves when being chosen (Cov_C) by the best grounded (top) resp. lifted (bottom) linear regression model.

some models found a subgroup of good planners and randomly chooses planners from its subgroup? If this would be the case, then for each selected planner the coverage on the tasks it was selected for should be approximately the same as its coverage on all test tasks. We see that this is clearly not the case. The models have learned to predict for a task which planner is good. Especially in the grounded setting, if the model selects a planner for a task, then the probability of the planner to solve the task is much higher than the planners coverage probability on all test tasks.

A model that has learned which planner is good for a task should assign the same planner to similar tasks. This means, especially within the same domain it should reliably select the same planner. As each domain contains tasks of varying difficulty, it might happen that within the same domain multiple planners are selected, but this should be noticeably different from randomly choosing a planner. Figure 1 shows for every task of the test set which planner was selected. The tasks are grouped by domains and within domains ordered as defined in the IPC 2018. We see that for many domains the models select a single planner. There are also some domains in which the models start with one planner and at one point switch to another planner.

We can conclude that the models have detected some structure in the features of each task and learned to exploit this to select a good subset of planners and to predict a good planner for a task or even for a set of similar tasks.

Robustness

Until now all experiments have been performed on the same training and test data. One might argue that our results are by chance and with different data the results look different. Thus, we end with two experiments which show that our findings are robust even with changes in the data.

To verify that our performance on the test data does not change significantly with different training data, we split the training data into 10 folds, but enforce that all tasks of the same domain will be assigned to the same fold. Every configuration is trained 10 times, but each time a different fold is ignored. If our approach is robust to changes in the training data, then the performance should not change much. Table 6 shows that indeed the configurations still perform similarly

	LR					RF	MLP	
	0	0.1	1	2	5	50	3	5
Binary	60.6(5.5)	81.8(6.1)	82.0(5.3)	82.6(5.4)	84.6(5.7)	70.4(3.8)	75.6(7.8)	73.9(7.9)
Log	63.1(5.7)	67.2(6.3)	81.1(8.1)	78.5(6.4)	79.8(6.0)	68.8(5.7)	64.8(0.2)	67.0(3.4)
Time	60.0(4.9)	56.1(5.0)	56.0(4.8)	55.5(5.1)	57.3(5.3)	71.5(5.9)	68.7(5.2)	68.8(5.5)

	LR					RF	MLP	
	0	0.1	1	2	5	50	3	5
Binary	63.6(4.0)	70.3(4.7)	70.6(3.5)	70.7(5.2)	66.1(6.0)	73.4(5.1)	63.2(3.7)	64.7(6.4)
Log	58.5(3.8)	69.8(3.1)	68.8(2.9)	69.6(3.0)	70.3(1.8)	74.1(6.0)	64.8(0.0)	64.8(0.0)
Time	63.7(3.7)	73.2(3.8)	70.3(2.9)	69.7(3.2)	70.6(3.6)	77.4(6.1)	66.3(2.4)	69.0(4.1)

Table 6: Mean coverage and standard deviation (in %) on the IPC 2018 tasks. The training data is split into 10 folds such that all tasks of the same domain are in the same fold. For each experiment repetition a different fold is ignored. Top: Performance on the grounded graphs. Bottom: Performance on the lifted graphs.

	LR					RF	MLP	
	0	0.1	1	2	5	50	3	5
Binary	85.6(8.3)	77.3(17.5)	76.3(17.2)	76.0(17.0)	76.6(18.0)	83.4(6.5)	76.8(17.2)	79.7(16.8)
Log	86.7(7.8)	85.9(8.4)	82.4(8.5)	78.2(15.6)	77.8(16.2)	83.4(9.5)	83.8(8.6)	84.8(6.5)
Time	86.3(8.5)	84.2(8.5)	84.3(8.6)	84.5(9.0)	84.3(8.8)	79.2(17.7)	84.9(4.4)	83.3(6.5)

	LR					RF	MLP	
	0	0.1	1	2	5	50	3	5
Binary	81.5(4.1)	75.8(15.4)	75.6(16.5)	74.1(15.2)	73.4(15.4)	77.6(13.3)	72.7(16.1)	72.2(16.1)
Log	81.0(9.9)	73.8(15.5)	82.3(7.8)	82.5(7.8)	82.7(7.8)	75.9(13.6)	82.2(8.5)	82.2(8.5)
Time	82.4(5.6)	78.7(10.6)	74.4(15.6)	74.6(16.8)	74.4(16.7)	78.9(13.2)	81.5(7.7)	78.9(9.0)

Table 7: Mean coverage and standard deviation (in %) on a random test fold. The data set - including the tasks from the IPC 2018 - is split into 10 folds such that all tasks of the same domain are in the same fold. For each experiment repetition a different fold is selected as test set. The other folds are used for training. Top: Performance on the grounded graphs. Bottom: Performance on the lifted graphs.

well. The performance might have decreased a bit, because $1/10^{\text{th}}$ of the training data was ignored. The standard deviation shows us that depending on which part of the training data is ignored the coverage can moderately vary.

Finally, we also change the test data. We split the whole data set into 10 folds, still keeping tasks from the same domain in the same fold. We train for each configuration 10 models. Each model uses a different fold as test data and trains on the remaining nine folds. Table 7 shows that our coverage increases. This might be because the tasks in the IPC 2018 were quite different from those tasks of previous IPCs, thus, learning from tasks of previous IPCs to select planners for tasks of the IPC 2018 is a difficult challenge. Changing the test data could make the learning easier. Another reason could be that planners in the data set were selected because they are good planners and their goodness could have been measured by using the old tasks of the IPC.

5 Summary and Future Work

We have shown that we can use simple machine learning techniques like linear regression to predict for a given tasks which planner to run to solve the task. In the grounded setting this even outperformed the image resp. graph convolution based baselines. Thus, we can have explainable decisions while still keeping top performance. At the same time this is not a justification to forget the image resp. graph convolution approaches. In the lifted setting those perform still better and have the advantage that the user does not need to come up with a set of good features, but the neural networks learn those features themselves.

Additionally to training those models, we studied which features are relevant for the predictions and how important they are. In the grounded setting the maximum degree of the graph was the most important information. On the other hand in the lifted setting there was no single feature with a similarly large impact. Finally, we verified that the models learned which planners to run for a domain.

Some future work is to use more fine grained features of the graph, e.g. number of operator nodes, such that we can reason which properties of the task instead of which property of the graph determine the planner choices.

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

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. ICAPS 2015*, 2–6.

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

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS 2009*, 34–41.

Breiman, L. 2001. Random forests. *Machine Learning* 45(1):5–32.

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* 2(4):303–314.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Proc. ICAPS 2012*, 343–347.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2015. Symmetry breaking in deterministic planning as forward search: Orbit space search algorithm. Technical Report IS/IE-2015-03, Technion.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In *Proc. SPIN 2006*, 19–34.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *Proc. MoChArt 2006*, 35–50.

Fan, G.; Müller, M.; and Holte, R. 2014. Non-linear merging strategies for merge-and-shrink based on variable interactions. In *Proc. SoCS 2014*, 53–61.

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*, 355–359.

Ferber, P.; Mai, T.; Huo, S.; Chen, J.; and Katz, M. 2019. Ipc: A benchmark data set for learning with graph- structured data. In *In Proceedings of the ICML-2019 Workshop on Learning and Reasoning with Graph-Structured Representations*.

Ferber, P. 2020. Supplemental material for the HSDIP workshop paper "Simplified Planner Selection". https://doi.org/10.5281/zenodo.4061613.

Galton, F. 1886. Regression towards mediocrity in hereditary stature. *Journal of the Anthropological Institute of Great Britain and Ireland* 15:246–263.

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.

Haslum, P. 2011. Computing genome edit distances using domain-independent planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

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.

Howe, A. E.; Dahlman, E.; Hansen, C.; Scheetz, M.; and von Mayrhauser, A. 1999. Exploiting competitive planner performance. In *Proc. ECP 1999*, 62–72.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. In *IPC-9 planner abstracts*, 57–64.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *Proc. ICLR 2015*.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations (ICLR 2017)*.

Köhler, J. 1999. Handling of conditional effects and negative goals in IPP. Technical Report 128, University of Freiburg, Department of Computer Science.

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.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *Proc. IJCAI 2011*, 1983–1990.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *JAIR* 35:623–675.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *Proc. AAAI 2011*, 1004–1009.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Proc. AAAI 2015*, 3364–3370.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and symmetries in classical planning. In *Proc. AAAI 2015*, 3371–3377.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019a. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proc. AAAI 2019*, 7715–7723.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019b. Theoretical foundations for structural symmetries of lifted PDDL tasks. In *Proc. ICAPS 2019*, 446–454.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized label reduction for merge-and-shrink heuristics. In *Proc. AAAI 2014*, 2358–2366.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An analysis of merge strategies for merge-and-shrink heuristics. In *Proc. ICAPS 2016*, 294–298.

Sievers, S. 2017. *Merge-and-shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation*. Ph.D. Dissertation, University of Basel.

Sievers, S. 2018. Fast Downward merge-and-shrink. In *IPC-9 planner abstracts*, 85–90.

Tibshirani, R. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58(1):267–288.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A symbolic bidirectional A* planner. In *IPC-8 planner abstracts*, 105–109.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proc. ICAPS 2014*, 323–331.