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

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

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

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

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

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

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

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

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

In our temporal language specification, conditions and effects can be declared to happen at any time within the duration of an action, and conditions can be durative, so they are
related with an interval of times (we called this feature “Intermediate Conditions and Effects”).

Definition 2.2. An effect on atom $a$ at relative time $\tau$ is a tuple $\langle \tau, a \rangle$ where $\tau$ is either $\text{START} + k$ or $\text{END} - k$ with $k \in \mathbb{Q}_{\geq 0}$. A condition\footnote{We only formalize closed condition intervals; open and semi-open intervals are supported by our implementation.} on atom $a$ in the relative interval $[\tau_1, \tau_2]$ is a tuple $\langle [\tau_1, \tau_2], a \rangle$ where $\tau_i$ is either $\text{START} + k_i$ or $\text{END} - k_i$ with $k_i \in \mathbb{Q}_{\geq 0}$.

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

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

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

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

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

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

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

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

Definition 2.6. A search state is a tuple $\langle \mu, \delta, \lambda, \chi, \omega \rangle$ s.t.:  
• $\mu$ records the ground predicates that are true in the state;  
• $\delta$ is a multiset of ground predicates, representing the active durative conditions to be maintained valid;  
• $\lambda$ is a list of lists of time-points. It constitutes the “agenda” of future commitments to be resolved.  
• $\chi$ is a Simple Temporal Network (STN) defined over time-points that stores and checks the metric and precedence temporal constraints;  
• $\omega$ is the last time-point evaluated in this search branch.

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

For the purpose of this paper, we need to define a set of problems of interest for a given domain: the objective of our learning technique will be to automatically synthesize a heuristic to guide a planner for efficiently solve any instance in the identified set. We make two assumptions on this set. First, we require the set to be finite: in principle one could have an infinite set and a sampler, but some details of our learning algorithm currently assume a finite set of problems.

Second, we assume the number of objects is bounded: this is needed because we use a feed-forward neural network that requires a known input dimension to be constructed. For this reason, we need to assume a maximum number of objects that results in a maximum number of ground predicates and in turn a maximum number of inputs for the neural network.

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

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

Definition 2.8. The optimal distance heuristic for a bounded planning problem set $\mathcal{P}^b_D$ is a function $h^*_{\mathcal{P}^b_D} : \bigcup_{P_i \in \mathcal{P}^b_D} S_{\langle D, P_i \rangle} \times \mathcal{P}^b_D \to \mathbb{R}$ s.t. for each $P_i \in \mathcal{P}^b_D$ and each state $s \in S_{\langle D, P_i \rangle}$, $d \equiv h^*(s, P_i)$ is the minimum number for which $\text{Succ}^d(s)$ is a goal state.

The aim of this paper is to automatically learn an approximation of $h^*$ given a temporal planning domain and a training bounded planning problem set.
3 Planning Heuristics as Reinforcement Learning

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

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

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

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

Let the state-action value of a policy \( \pi \) be as follows.

\[
Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s, a_t = a \right]
\]

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

**Definition 3.2.** Given a bounded planning problem set \( \mathcal{P}_D^k \), its MDP encoding \( \mathcal{M}_{\mathcal{P}_D^k} \) is the MDP \( (S, A, T, R, \pi) \) where:

- \( S = \{\vdash\} \cup \{s_i \mid (D_i, P_i, \pi) \in \mathcal{P}_D^k\} \)
- \( A = \{\xi\} \cup \{a \mid (D_i, P_i, \pi) \in \mathcal{P}_D^k\} \)
- \( T(s, a) = \left\{ \left\{ (I_{P_i}, \pi_{D_i}^{D_i}) \mid P_i \in \mathcal{P}_D^k \right\} \right\} \) if \( s = \vdash, a = \xi \)
- \( T(s, a) = \left\{ \left\{ (a[s], 1) \right\} \right\} \) if \( s \neq \vdash \)

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

- \( R(s, a, s') = \left\{ \begin{array}{ll} 1 & \text{if } s' = (s_i, (O, I, G)) \text{ and } s_i \vdash G \\ -1 & \text{if } \not\exists b, s'' = b[s'] \\ 0 & \text{otherwise} \end{array} \right. \)

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

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

**Theorem 3.1.** For a bounded planning problem set \( \mathcal{P}_D^k \), the following equation holds.

\[
h^*_D(s) = \begin{cases} \log(V^*_D(s)) & \text{if } V^*_D(s) > 0 \\ \infty & \text{otherwise} \end{cases}
\]

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

On a deterministic system with the reward shape of \( \mathcal{M}_{\mathcal{P}_D^k} \), the optimal policy is the policy reaching a goal state in the minimum number of steps. Let \( (s_0, \cdots, s_g) \) be the optimal path from state \( s_0 \) to the nearest state satisfying a goal \( s_g \). The discounted reward in \( s_i \) clearly is \( V^*_D(s_i) = \gamma^{g-i} \), and the distance to the goal is \( h^*_D(s_i) = g - i \). If a state \( s \) cannot reach any goal, then \( V^*_D(s) \leq 0 \). Hence, we can retrieve the distance from the discounted reward as per the theorem statement. \( \square \)
Algorithm 2 Vectorization of an STN $\chi$

1. procedure STN2VECTOR($\chi$)
2. $F \leftarrow \langle \text{for all actions } a_i \rangle$; $\tau \leftarrow \text{GETMINMAKESPAN}\,\text{SOLUTION}(\chi)$
3. lastSafe $\leftarrow 0$ \triangleright A "safe" state is a state where no action is running
4. balance $\leftarrow 0$ \triangleright The difference between the started and terminated actions
5. for all time points $tp$ sorted by $\tau[tp]$ do
6. if $tp$ is a starting of an action then balance $\leftarrow balance + 1$
7. else if $tp$ is the termination of an action then balance $\leftarrow balance - 1$
8. if balance $= 0$ then
9. $\vec{r} \leftarrow \langle \text{for all actions } a_i \rangle$
10. lastSafe $\leftarrow \tau[tp]$
11. else
12. $\vec{r}[\text{action}(tp)] \leftarrow \tau[tp] - \text{lastSafe}$
13. if $tp = \omega$ then break \triangleright $\omega$ is the last scheduled time-point
14. return $\vec{r}$

4 Reinforcement Learning Algorithm

In this section, we detail a dedicated RL algorithm derived from classical Value Iteration that uses a Neural Network to estimate the optimal value function $V_{M_{p_k}^*}^\pi$. The overarching idea is to use RL to estimate $V_{M_{p_k}^*}^\pi$ and from that estimation, derive an estimation of $h_{p_k}^\pi$.

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

At this point, we are left with a set of problems that can be grounded, resulting in a consistent number of fluents. The neural network we will use to represent the RL policy requires a vector representation of a state of the MDP $M_{p_k}^*$. Given a search state $\langle \mu, \delta, \lambda, \chi, \omega \rangle$ and a problem $P_t$, we define the vectorization of the MPD state as follows. First, we vectorize the predicate values (i.e. the $\mu$ part of the state), we pick a fixed ordering for the ground predicates of the biggest possible problem in $P_{k}^*\mathcal{D}$. Note that the cardinality of the ground states is exactly $k^t$ with $x \equiv \sum_{p \in P} \text{arity}(p)$. We set the input vector value to 1 (resp. 0) if the corresponding ground predicate is true (resp. false). The second part of the vector is a representation of the status of the events (i.e. the $\lambda$). For each possible ground action we have a vector element set to the size of the corresponding list of time-points

2A constant is a fluent that is assigned in the initial state and never changed. ANML explicitly supports constants.
Algorithm 3 Reinforcement Learning Algorithm

1: procedure RL2PLANHEURISTIC(tis, N episodex)
2: \( V_{nn} \leftarrow \text{IN2NN}( \) ) \( \triangleright \) Creates NN with the described architecture
3: mem \( \leftarrow \text{LIST}( ) \) \( \triangleright \) The algorithm memory experience
4: \( i2s \leftarrow \{ i \rightarrow \theta \mid i \in \text{tis} \} \) \( \triangleright \) maps \( i \) to \( \theta \) of times \( i \) was solved
5: for \( i \in 1, \ldots, N \text{episodex} \) do
6: \( (s, \text{goals}) \leftarrow \text{inst} \leftarrow \text{PickKeyInvProportionallyToValue}(i2s) \)
7: \( \text{done, solved} \leftarrow \langle \text{False, False} \rangle \)
8: \( \pi \leftarrow \langle s \rangle \)
9: while not done do
10: \( e \leftarrow e_{max} \times e^{\ln(\epsilon_{min}/\epsilon_{episodex})} \times i \) \( \triangleright \) Decay \( \epsilon_{max} \rightarrow \epsilon_{min} \)
11: if \( \text{RANDOM}() \leq e \) then \( \triangleright \) With probability \( e \)
12: \( c \leftarrow \text{SelectActionUsingHeuristic}(s) \)
13: else
14: \( c \leftarrow \text{SelectActionUsingPolicy}(V_{nn}, s) \)
15: \( (s', \text{done}, \rho) \leftarrow \text{DoStep}(\pi, s, \alpha, \text{inst}) \triangleright \) Simulate \( \alpha \) move
16: \( \text{Append}(\text{mem}, (s, \rho)) \)
17: if \( \rho|\alpha| = 1 \) then \( \triangleright \) if solved then
18: solved \( \leftarrow \text{True} \)
19: \( \text{Append}(\pi, (s')) \)
20: \( s \leftarrow s' \)
21: \( V_{nn} \leftarrow \text{REPLAY}(V_{nn}, \text{mem}) \) \( \triangleright \) Do a learning step
22: if solved then
23: \( i2s[\text{inst}] \leftarrow i2s[\text{inst}] + 1 \) \( \triangleright \) Update solved \# count for \( \text{inst} \)
24: return \( V_{nn} \)
25: procedure PickKeyInvProportionallyToValue(h)
26: \( V \leftarrow \{ v \mid i \rightarrow v \in b \} \) \( \triangleright \) Get the values of the map \( b \)
27: \( m \leftarrow \text{CEIL}(1.1 \times \text{MAX}(V)) \) \( \triangleright \) Allow a 10\% slack
28: \( t \leftarrow m \times |b| - (\sum_{v \in V} v) \) \( \triangleright \) Factor to normalize probabilities
29: \( \text{perc} \leftarrow \{ i \rightarrow \frac{m - t}{m} \mid i \rightarrow v \in b \} \) \( \triangleright \) Probability to pick each element \( i \)
30: return \( \text{RandomSelectionBasedOnPercentage}(\text{perc}) \)
31: procedure SelectActionUsingHeuristic(s)
32: \( h \leftarrow \text{EMPTYMap}( ) \) \( \triangleright \) A map from successor states to their heuristic values
33: for all \( \alpha \) \( \langle \text{GetApplicableEvents}(s) \rangle \) do
34: \( s' \leftarrow \text{SimulateActionApply}(s, \alpha) \)
35: \( h|\alpha| \leftarrow h|\alpha| + 1 \) \( \triangleright \) The experience is collected in a finite-size memory \( \text{mem} \) that caches pairs \((s, \rho)\) where \( s \) is a state and \( \rho \) is a mapping of all the applicable events in \( s \) to their immediate reward. Differently from a standard RL algorithm, we manipulate the probability of selecting a specific instance among the ones in the training set by favoring the ones that have been solved (i.e., that reached a reward of 1) less often. This amounts to dynamically adapting the probability distribution of the \( \xi \) transition in the MDP \( M_{\pi}^{s} \).
36: RL2PLANHEURISTIC takes a set of training ground instances \( tis \) and a number of episodes to run for \( N_{episodex} \); its goal is to evolve a value function represented as a neural network \( V_{nn} \) that approximates the optimal value function.
37: The experience is collected in a finite-size memory \( \text{mem} \) that caches pairs \((s, \rho)\) where \( s \) is a state and \( \rho \) is a mapping of all the applicable events in \( s \) to their immediate reward.
38: Differently from a standard RL algorithm, we manipulate the probability of selecting a specific instance among the ones in the training set by favoring the ones that have been solved (i.e., that reached a reward of 1) less often. This amounts to dynamically adapting the probability distribution of the \( \xi \) transition in the MDP \( M_{\pi}^{s} \).
39: RL2PLANHEURISTIC takes a set of training ground instances \( tis \) and a number of episodes to run for \( N_{episodex} \); its goal is to evolve a value function represented as a neural network \( V_{nn} \) that approximates the optimal value function.
40: RL2PLANHEURISTIC takes a set of training ground instances \( tis \) and a number of episodes to run for \( N_{episodex} \); its goal is to evolve a value function represented as a neural network \( V_{nn} \) that approximates the optimal value function.
41: We use an exponential epsilon-decay strategy to balance between random exploration and policy exploitation, but we exploit the planning heuristic \( (h_{add} \) in our case) to skew the probabilities among the possible events. This is done in the SelectActionUsingHeuristic function that re-uses the \( \text{PickKeyInvProportionallyToValue} \) function to randomly pick an action with a probability inversely proportional to the heuristic value.3
42: The trajectory simulation is standard and uses the TAMER planning engine as a simulator. In the memory \( \text{mem} \), we store for each state in the trajectory the reward of each possible successor state. We forcibly bound the length of the traces to a maximum depth given by the \( \text{GetMaxDepth} \) function; we want to avoid the exploration of very long (or even infinite) paths, in fact, by allowing an arbitrary number of steps we might get trapped in loops yielding 0 reward and never finish an episode. In the following, we indicate the maximum depth used to bound the paths as \( \Delta_{RL} \).
43: We use a reward function that is slightly adjusted with respect to the one presented in definition 3.2; in particular, we grant a small \((10^{-5}) \) in total) reward for the sub-goals (a sub-goal is an element of \( G_{\text{app}} \)) achieved for the first time in a trace and we give 0 reward for traces that reach the maximum depth. This is done by the \( \text{GetRewardFunction} \) that analyzes the trace and checks, for each sub-goal, if it is achieved for the first time or not. Note that this change has a small numerical impact on the expected reward and hence on theorem 3.1, but we picked a number that is small enough
44: Since the heuristic estimates the distance to the goal, we prefer events leading to successor states having a small heuristic value.
to be practically negligible while giving useful intermediate reward signals.

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

\[
V_{t+1}(s) \leftarrow \max_{\alpha} (R(s, \alpha, s') + \gamma \times V_t(s'))
\]

where \( \alpha \) ranges over the applicable events in \( s \) and \( s' \) is the successor state of \( s \) obtained by applying \( \alpha \).

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

\[
h_{nn}(s) = \begin{cases} 
\min(\log_{\gamma}(V_{nn}(\bar{s})), \Delta_h) & \text{if } V_{nn}(\bar{s}) > 0 \\
\Delta_h & \text{if } V_{nn}(\bar{s}) = 0 \\
\Delta_h - \min(\log_{\gamma}(-V_{nn}(\bar{s})), \Delta_h) & \text{otherwise}
\end{cases}
\]

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

**5 Related Work**

Several works aimed at combining learning with planning.

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

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

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

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

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

6 Experimental Evaluation

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

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

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

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

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

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

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

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

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

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

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

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

## 7 Conclusions

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

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