Synthesis of Search Heuristics for Temporal Planning via Reinforcement Learning

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

Abstract

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

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

1 Introduction

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

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

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

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

2 Problem Definition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm 1 TAMER search algorithm

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

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

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

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

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

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

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

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

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

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

3 Planning Heuristics as Reinforcement Learning

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

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

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

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

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

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

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

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

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

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

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

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

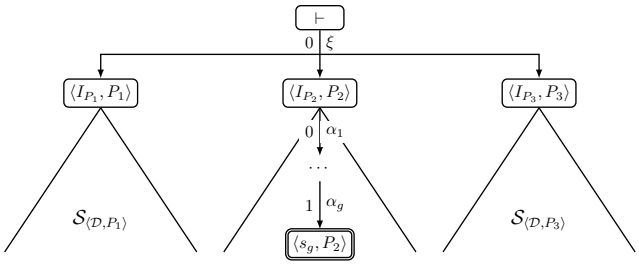


Figure 1: The state space and rewards of $\mathcal{M}_{\mathcal{P}^k_{\mathcal{D}}}$.

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

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

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

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

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

Algorithm 2 Vectorization of an STN χ

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

4 Reinforcement Learning Algorithm

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

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

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

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

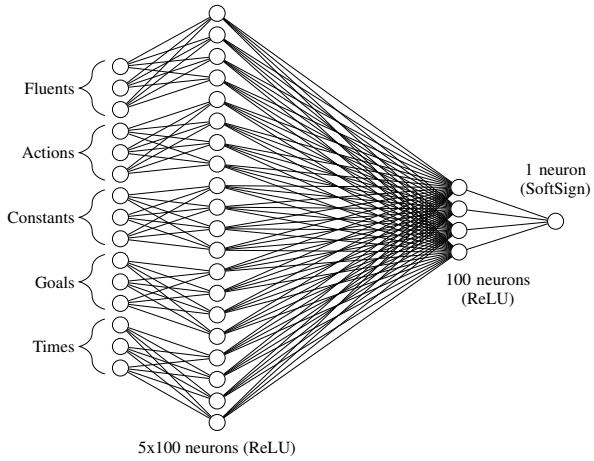


Figure 2: The neural network architecture.

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

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

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

Algorithm 3 Reinforcement Learning Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

to be practically negligible while giving useful intermediate reward signals.

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

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

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

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

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

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

5 Related Work

Several works aimed at combining learning with planning.

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

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

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

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

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

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

6 Experimental Evaluation

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

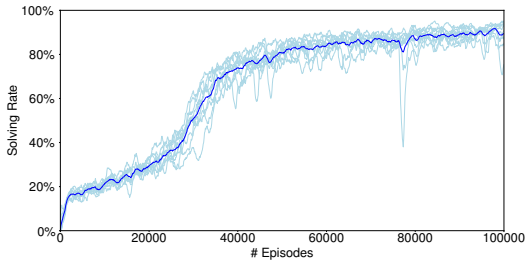
We consider two benchmark planning domains. The first one is the MAJSP domain used in (Micheli and Scala 2019) and (Valentini, Micheli, and Cimatti 2020); the domain consists of a job-shop scheduling problem in which a fleet of moving agents transport items and products between operating machines. We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

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

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

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

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



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

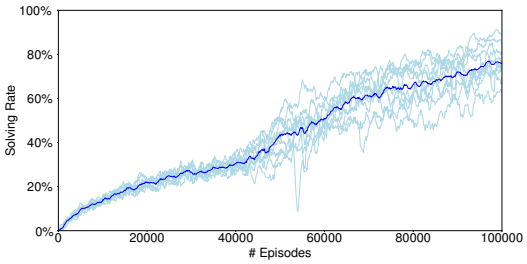
Figure 3: Results on the MAJSP domain: learning curves (above) and coverage (table). We plot the curve for each fold in light blue and the average solving rate in dark blue. For each fold and each approach, we report the total number of solved instances and the average plan length.

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

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

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

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



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

Figure 4: Results for the Kitting domain.

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

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

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

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

7 Conclusions

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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