Synthesis of Search Heuristics for Temporal Planning via Reinforcement Learning

Andrea Micheli, Alessandro Valentini
Fondazione Bruno Kessler, Trento, Italy
amicheli@fbk.eu, 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 widen the reach of current temporal planners with encouraging results.

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.

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, under the assumption of knowing the maximum number of objects in each problem. 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 dramatically outperformed.

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 (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 0.1. *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 dura-

tion 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 0.2. *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 0.3. *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 0.4. *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 0.5. *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 0.6. *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;*
- *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 a classic best-first search in this state space (see (Valentini, Micheli, and Cimatti 2020) for the details). We indicate the successor function as $\text{SUCC}(s)$.

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 solving 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 0.7. *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).

Definition 0.8. *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.

Our aim is to automatically learn an approximation of $h^*_{\mathcal{P}_{\mathcal{D}}^k}$.

Heuristics from Value Functions

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 $\mathcal{P}_{\mathcal{D}}^k$ 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 0.9. *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*

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

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 0.10. *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.

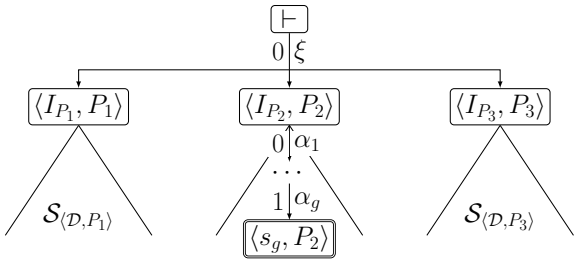

Figure 1: The state space and rewards of $\mathcal{M}[\mathcal{P}^k_{\mathcal{D}}]$.

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 0.1. *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. □

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. To use a neural network, we need a fixed-size vector representation of a given MDP state. First, we have to scale the state representation so that all the problems in $\mathcal{P}^k_{\mathcal{D}}$ can be represented uniformly despite the possible differences in the number of actions and fluents. We exploit the bound k on the number of objects: for each object o_i, we introduce a fresh Boolean

Algorithm 1 Vectorization of an STN χ

```
1: procedure STN2VECTOR(χ)
2:     r⃗ ← ⟨0 for all actions a_i⟩; τ ← GETMINMAKESPANSOLUTION(χ)
3:     lastSafe ← 0          ▷ A "safe" state is a state where no action is running
4:     balance ← 0   ▷ The difference between the started and terminated actions
5:     for all time points tp sorted by τ[tp] do
6:         if tp is a starting of an action then balance ← balance + 1
7:         else if tp is the termination of an action then balance ← balance − 1
8:         if balance = 0 then
9:             r⃗ ← ⟨0 for all actions a_i⟩; lastSafe ← τ[tp]
10:        else
11:            r⃗[action(tp)] ← τ[tp] − lastSafe
12:        if tp = ω then break                ▷ ω is the last scheduled time-point
13:    return r⃗
```

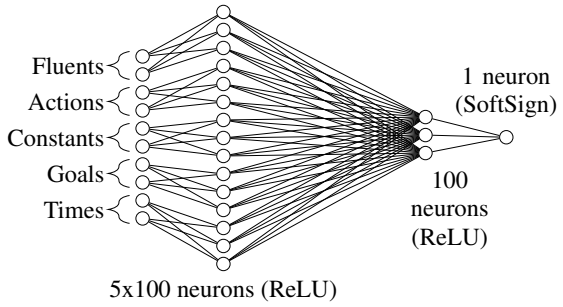

Figure 2: The neural network architecture.

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 transformation, scales any problem in $\mathcal{P}_{\mathcal{D}}^k$ to a problem with exactly k objects and a fixed number of actions, that maintains all the original plans.

We can now ground all the problems obtaining instances with a consistent number of fluents. 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 $\mathcal{P}_{\mathcal{D}}^k$. 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 1. 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.

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 $V^*_{\mathcal{M}[\mathcal{P}_{\mathcal{D}}^k]}$. Note that the expected reward along any path must be in the range $[-1, 1]$ because of the reward shape of $\mathcal{M}[\mathcal{P}_{\mathcal{D}}^k]$, 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 2. The main function RL2PLANHEURISTIC takes a set of training ground instances E and a number of episodes to run for n_{ep}; 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 $\mathcal{M}[\mathcal{P}_{\mathcal{D}}^k]$. 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 likely more difficult.

We use an exponential ϵ-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

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

Algorithm 2 Reinforcement Learning Algorithm

1: **procedure** RL2PLANHEURISTIC(E, n_{ep})
2: $V_{nn} \leftarrow$INITNN( ); $mem \leftarrow$LIST( )
3: $i2s \leftarrow \{i \rightarrow 0 \mid i \in E\}$ ▷ maps i to # of times i was solved
4: **for** $i \in 1, \ldots, n_{ep}$ **do**
5: $\langle s, goals\rangle = inst \leftarrow$PICKKEYINVPROPORTIONALLYTOVALUE($i2s$)
6: $\langle done, solved\rangle \leftarrow \langle False, False\rangle$; $\pi \leftarrow \langle s\rangle$
7: **while not** $done$ **do**
8: $\epsilon \leftarrow \epsilon_{max} \times exp(\frac{\ln(\epsilon_{min}/\epsilon_{max})}{n_{ep}} \times i)$ ▷ ϵ-decay
9: **if** RANDOM( ) $< \epsilon$ **then** $\alpha \leftarrow$SELECTACTIONUSINGHEURISTIC(s)
10: **else** $\alpha \leftarrow$SELECTACTIONUSINGPOLICY(V_{nn}, s)
11: $\langle s', done, \rho\rangle \leftarrow$DOSTEP($\pi, s, \alpha, inst$) ▷ Simulate α move
12: APPEND($mem, \langle s, \rho\rangle$)
13: **if** $\rho[\alpha] = 1$ **then** $solved \leftarrow True$
14: APPEND($\pi, \langle s'\rangle$); $s \leftarrow s'$
15: $V_{nn} \leftarrow$REPLAY(V_{nn}, mem) ▷ Do a learning step
16: **if** $solved$ **then**
17: $i2s[inst] \leftarrow i2s[inst] + 1$ ▷ Update solved # count for $inst$
18: **return** V_{nn}

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

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

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

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

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

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

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 a simulator of the planning instance. In 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 0.10: 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 an impact on the expected reward and hence on Theorem 0.1, but we picked a small enough number 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 45) 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. We use the learned value function to construct a heuristic function for our planning algorithm following Theorem 0.1 with some practical adjustments to take into account the maximum exploration depth (Δ_{RL}) we fixed for the RL 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 0.1 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 $V_{nn} <= 0$ we return a value that is between Δ_h and $2\Delta_h$. Note that h_{nn} never returns ∞, as we cannot formally guarantee that a state is a dead-end, therefore, we use the range of numbers 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 $\Delta_h \geq \Delta_{RL}$. We empirically found that larger values for Δ_h yield better results. This is probably due to the

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

"flattening" of the heuristic value due to the $\min$ operators: the smaller Δ_h, the more values of $h_{nn}(s)$ get compressed to Δ_h, losing the possibility of discriminating among them.

Related Work

Several works aimed at combining learning with planning. Macro-actions (Coles and Smith 2007; Botea et al. 2005) consist in learning "shortcuts" in the search-space. Case-based planning (Spalazzi 2001; Bonisoli et al. 2015) constructs a database of plans for a specific domain that is used to efficiently solve new problems. Some authors (Asai and Fukunaga 2018) also focused on the problem of learning symbolic models from data.

This paper, instead, fits in the learning of heuristics research line. Previous works deal with this problem in the case of classical planning. In their seminal work, de la Rosa, Olaya, and Borrajo (2007) use a case-based database to inform heuristics. Yoon, Fern, and Givan (2008) use machine-learning techniques to learn control policies that are then exploited in a classical, heuristic-search planner. 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 these works, we tackle expressive temporal planning with intermediate conditions and effects and provide a fully-automated technique to learn heuristics from simulations via RL. 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 on the same domain.

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. 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 specializes for a 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 advancement in this area is (Toyer et al. 2018) presenting "Action Schema Networks", where a generalized policy is extracted by means of deep learning from a set of problem instances and a planning-specific transfer-learning technique is used to generalize the policy for new problems. In this paper, we are not tackling generalized planning: we maintain (and rely on) reasoning capabilities in the planner instead of generating a plan that works for all the instances.

Experimental Evaluation

In our experiments, we consider two benchmark planning domains: the MAJSP domain used in (Micheli and Scala 2019) and a new domain called "Kitting". MAJSP consists of a job-shop scheduling problem in which a fleet of moving agents transport items and products between operating machines. In Kitting 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 770 of MAJSP and 1092 instances of Kitting.

We implemented the learning part of our framework in Python3 using an adaptation of TAMER as a 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/aaai21.

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 ($\pi_{nn}(s) = \arg\max_{\alpha} V_{nn}(\alpha[s])$).

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

Figure 3 reports the coverage results for all the ten folds; 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. Moreover, we plotted the learning curve for the cross validation (bottom-left): the y-axis reports the solving rate of the previous 1000 episodes,

MAJSP

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

Kitting

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

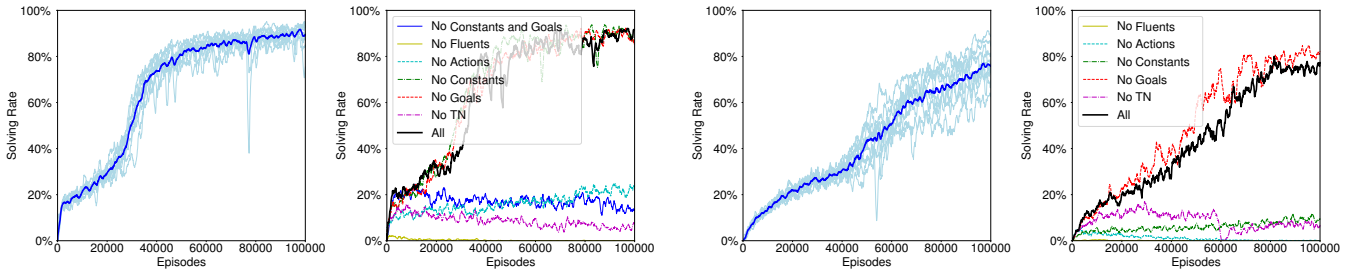

Figure 3: Experimental results for the MAJSP (left) and Kitting (right) domains: coverage table (above), learning curves for the 10 folds (below left) and learning curves with different input configurations (below right).

that is we plot the percentage of episodes that reached a goal state while learning (the dark, bold curve is the average).

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. Interestingly, in the MAJSP domain after 40K 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.

Both learning-based approaches (π_{nn} and TAMER (h_{nn})) are significantly superior to the plain TAMER(h_{add}). The two 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). TAMER (h_{nn}) 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 of the planner with the learned insight. We also highlight that both TAMER (*) approaches are guaranteed to eventually find a plan if it exists, while π_{nn} can diverge or deadlock.

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.

The bottom-right part of Figure 3 shows the learning curves for both the domains. 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 encoding of MAJSP: 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 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.

Conclusions

We presented the first approach to learn heuristic functions for temporal planning. We designed a learning flow that uses a finite set of instances with different number of objects and synthesizes 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.

In the future, we plan to study methods to overcome the need for the bounding in the number of objects and to relax the finiteness assumption we imposed on the training (e.g. allowing an instance sampler in input). A third future direction is to generalize the network architecture, developing a structured way to derive it from the domain definition.

Acknowledgments

This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

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.

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.

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.

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.