ECAI 2025
I. Lynce et al. (Eds.)

doi:10.3233/FAIA251102

Exploiting Symbolic Heuristics for the Synthesis of Domain-Specific Temporal Planning Guidance Using Reinforcement Learning

Irene Brugnara[a,*], **Alessandro Valentini**[a] **and Andrea Micheli**[a]

[a]Fondazione Bruno Kessler, Trento, Italy

Abstract. Recent work investigated the use of Reinforcement Learning (RL) for the synthesis of heuristic guidance to improve the performance of temporal planners when a domain is fixed and a set of training problems (not plans) is given. The idea is to extract a heuristic from the value function of a particular (possibly infinite-state) MDP constructed over the training problems.

In this paper, we propose an evolution of this learning and planning framework that focuses on exploiting the information provided by symbolic heuristics during both the RL and planning phases. First, we formalize different reward schemata for the synthesis and use symbolic heuristics to mitigate the problems caused by the truncation of episodes needed to deal with the potentially infinite MDP. Second, we propose learning a residual of an existing symbolic heuristic, which is a "correction" of the heuristic value, instead of eagerly learning the whole heuristic from scratch. Finally, we use the learned heuristic in combination with a symbolic heuristic using a multiple-queue planning approach to balance systematic search with imperfect learned information. We experimentally compare all the approaches, highlighting their strengths and weaknesses and significantly advancing the state of the art for this planning and learning schema.

1 Introduction

Automated temporal planning is the problem of synthesizing a course of action to achieve a desired goal condition, given a formal description of a timed system [16]. Temporal planning finds natural applications in many domains such as robotics, logistic and process automation because time and temporal constraints are commonplace in real-world applications. Despite a long history of research and improvements, scalability is the key issue for automated temporal planners: depending on the modeling assumptions the computational complexity ranges from PSPACE-completeness to undecidability [17]. Using machine learning to specialize a temporal planner for a domain is a promising way to mitigate the scalability issue and automatically devise effective solvers.

An approach to tackle this specialization of temporal planning solvers consists of codifying a set of training problems as a specifically designed Markov Decision Process (MDP) and using Reinforcement Learning [26] to estimate its value function; in turn, it is possible to extract a temporal planning heuristic biased on the distribution of training problems for a search-based temporal planner from the estimated value function [19]. The workflow consists of two phases: an offline phase in which the temporal planner is specialized on the given training dataset in a fully automated way, and then an online phase in which the specialized planner is tasked to solve multiple problems on the same domain. More in detail, this approach builds on the discretization of the search space used by planners such as TAMER [28]: the continuous space of the timed system is abstracted by symbolically representing time and timing constraints using Simple Temporal Networks (STN) [11] and the planner searches in the (partial) orderings of *events*, which are either the beginning or ending of durative actions, or intermediate effects and conditions if they are allowed. The learning framework exploits this discretization of the search space by defining a tree-shaped MDP, where an instance from the training set is randomly selected at the beginning of the episode and then the MDP is isomorphic to the planner search space. An optimal value function for the MDP is shown to be in a functional relation with the optimal heuristic for the training instances, and the authors show that by using neural networks as function approximators and encoding states (including symbolic temporal information and the goal) as feature vectors, the approach can generalize to instances drawn from the same distribution of the training set.

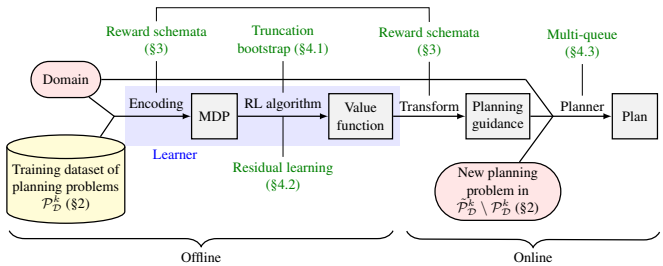

Figure 1. Overall learning and planning framework. In green we indicate our major contributions.

In this paper, we evolve this framework by leveraging information from existing symbolic temporal planning heuristics as shown in Figure 1. We start by considering different reward schemata for the framework (all capturing the temporal plan validity) and by defining a "truncation bootstrap" method to cope with the problem of truncation of episodes: since the MDP derived from a temporal planning problem is potentially infinite, we need to truncate episodes after a certain number of steps, and we use the symbolic heuristic to provide a value function estimation in the last state of the episode avoiding learning issues and allowing for better generalization. Then, we propose to reframe the learning problem from the eager synthesis of a

* Corresponding Author. Email: ibrugnara@fbk.eu.

heuristic function to the learning of an additive residual over an existing symbolic heuristic: the key intuition is to simplify the learning task by providing a template function to correct instead of eagerly regressing the whole function from scratch, resulting in a more regular target function. Finally, we modify the inference phase employing a multiple-queue planning approach, where one queue is ordered using a standard symbolic heuristic in an A^* fashion, while the other is ordered using the learned value function as a ranking function. The intuition is that the second queue exploits the learned information greedily, while the first queue provides a systematic exploration of the search space and compensates for learning imperfections easily compared to a single-queue A^* approach where the systematic search is controlled by the cumulative cost (g). The overall result is a comprehensive learning and planning framework for temporal planning that leverages existing symbolic heuristics at every step.

We empirically evaluate all components of the modified framework in isolation and in combination, showing they dramatically outperform the baseline planner with no learned information and significantly improve the results in [19]. In this setting of per-domain learning, we are interested in optimizing the online solving time, and not the offline learning time (provided it remains feasible).

2 Problem Definition

In this section, we define the learning problem we tackle in this paper. We borrow most of the basics of our problem definition from [19]. We use ANML [24] to specify action-based temporal planning problems: our language allows durative actions (similarly to PDDL 2.1 [13]) and "Intermediate Conditions and Effects", allowing actions to check for conditions and apply effects at arbitrary points in time during action execution. For the sake of brevity, we do not report the details of the formal planning language here; the details can be found in [28]. For this paper, we only need to characterize the set of planning problems we use for training and the search space of a planner on such problems. As customary in planning, we define a planning instance as the composition of a "domain" specification (containing the set of predicates and action schemata) and a "problem" consisting of a set of objects to instantiate the domain elements, an initial state and a goal condition.

Definition 1. *A **planning problem** $\mathcal{P}$ for a domain $\mathcal{D}$ is a tuple $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ where $\mathcal{O}$ is a finite set of objects; $\mathcal{I}$ and $\mathcal{G}$ are sets of ground atoms over predicates of $\mathcal{D}$, which represent the initial state and the goal respectively. A **planning instance** is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$.*

Semantically, given a planning instance we can first perform standard grounding, that is get rid of action parameters by substituting all the possible combinations of objects as parameters. Given the ground planning instance, a valid plan is a simulation of the system starting from the initial state and terminating in a goal state. The full semantics can be found in [28].

Given a ground planning instance, planners such as POPF [8] or TAMER [28] search an interleaving of *events* (also called happenings, time-points or snap-actions) that represent the discrete changes of state in a plan ensuring that the abstract sequence of events can be concretized to a plan by scheduling the temporal constraints. The timings of events are derived by solving a constraint satisfaction problem (a Simple Temporal Network [11], in particular) composed of the temporal constraints and the precedence constraints enforcing causal relations that the planner builds while constructing the sequence of events. In order to construct such a sequence of events and to record the temporal constraints, we represent search states as per definition below, borrowed from [28], and we perform a search in the space of the possible reachable states starting from the initial state. The transitions considered by the planner are the events for a planning problem $\mathcal{P}$ (indicated as $\mathcal{E}_{\langle \mathcal{D}, \mathcal{P} \rangle}$) and 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. *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 (infinite) set of possible search states for a given instance $\langle \mathcal{D}, \mathcal{P} \rangle$ as $\mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$. For deterministic temporal planning, the graph whose set of nodes is $\mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$ and whose set of edges is $\mathcal{E}_{\langle \mathcal{D}, \mathcal{P} \rangle}$ is a tree rooted at the state corresponding to $\mathcal{I}$. This is because most planners do not perform state merging[1] as it is computationally heavy for temporal planning in general [7]. Given a state $\sigma \in \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$, we denote with $\alpha(\sigma)$ the set of applicable events in σ. Given a state $\sigma \in \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$ and an event $\epsilon \in \alpha(\sigma)$, we denote with $\nu(\sigma, \epsilon)$ the state reached when applying ϵ in σ.

We assume that a bound on the number of objects for both training and testing problems is known. This is needed to construct the size of the first layer of the feed-forward neural network that we will learn. Hence, we define the training set as follows.

Definition 3. *The **set of k-bounded planning problems** for a domain $\mathcal{D}$ is $\mathbb{P}^k_{\mathcal{D}} \doteq \{\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle \mid k \geq |\mathcal{O}|\}$.*

We are interested in a particular subset of k-bounded planning problems $\tilde{\mathcal{P}}^k_{\mathcal{D}} \subseteq \mathbb{P}^k_{\mathcal{D}}$ and we assume a uniform distribution over it. In the following, we assume to have a sample of this distribution available as a subset $\mathcal{P}^k_{\mathcal{D}} \subseteq \tilde{\mathcal{P}}^k_{\mathcal{D}}$ which will be our training set. Our aim is to synthesize planning guidance that can aid the search of a planner for problems drawn from the same distribution of $\mathcal{P}^k_{\mathcal{D}}$.

The most common form of planning guidance is a heuristic function. 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) and returns (an estimation of) the number of events needed to reach a goal state from the input state. In this work, we focus on satisficing planning and therefore do not consider optimality metrics.

Definition 4. *The **optimal distance heuristic** for a training set $\mathcal{P}^k_{\mathcal{D}}$ is a function $h^*_{\mathcal{P}^k_{\mathcal{D}}} : \left(\bigcup_{\mathcal{P} \in \mathbb{P}^k_{\mathcal{D}}} \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}\right) \times \mathbb{P}^k_{\mathcal{D}} \to \mathbb{R}$ s.t. for each $\mathcal{P} \in \mathcal{P}^k_{\mathcal{D}}$ and each state $\sigma \in \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$, $d \doteq h^*(\langle \sigma, \mathcal{P} \rangle)$ is the minimum number such that there exists a sequence of events $\langle \epsilon_1, \ldots, \epsilon_d \rangle$ and a sequence of states $\langle \sigma_0, \sigma_1, \ldots, \sigma_d \rangle$ s.t. $\sigma_0 = \sigma$, σ_d is a goal state, $\epsilon_j \in \alpha(\sigma_{j-1})$ and $\sigma_j = \nu(\sigma_{j-1}, \epsilon_j)$ for all $j \in \{1, \ldots, d\}$.*

Our first aim is to automatically learn an approximation of $h^*_{\mathcal{P}^k_{\mathcal{D}}}$, then in Section 4.3 we will present another form of planning guidance we can learn, which is a ranking function. The overall idea is that the learned guidance will generalize to problems "similar" to the ones in $\mathcal{P}^k_{\mathcal{D}}$, i.e. problems in $\tilde{\mathcal{P}}^k_{\mathcal{D}} \setminus \mathcal{P}^k_{\mathcal{D}}$.

[1] If state merging is performed in temporal planning, our algorithms would also work on the DAG-shaped search space; but not in the classical planning case, where the search space is a generic graph.

3 MDP for Heuristic Synthesis

In order to learn informed planning guidance, we encode our training problems as an MDP adapted from [19], which we then use for reinforcement learning. In this MDP, the initial state is chosen uniformly at random from the set of initial states of the training planning problems (for generalization) and then the MDP state space is isomorphic to the search space of a temporal planner for the selected instance.

Formally, given a bounded planning problem set $\mathcal{P}^k_{\mathcal{D}}$, its MDP encoding $\mathcal{M}[\mathcal{P}^k_{\mathcal{D}}]$ is the MDP defined as follows.

- The state space is $S \doteq \{\bar{d}\} \cup \bigcup_{\mathcal{P}\in\mathcal{P}^k_{\mathcal{D}}} \{\langle\sigma,\mathcal{P}\rangle | \sigma \in \mathcal{S}_{\langle\mathcal{D},\mathcal{P}\rangle}\}$, and it is infinite since the planning state space is infinite ($\bar{d}$ is a fresh symbol denoting a special state whose purpose will become clear in the following);
- the action space is $A \doteq \{\bar{m}\} \cup \bigcup_{\mathcal{P}\in\mathcal{P}^k_{\mathcal{D}}} \mathcal{E}_{\langle\mathcal{D},\mathcal{P}\rangle}$ (where $\bar{m}$ is another fresh symbol for a special action);
- the probability distribution over initial states is

$$I(\langle\sigma,\mathcal{P}\rangle) \doteq \begin{cases} \frac{1}{|\mathcal{P}^k_{\mathcal{D}}|} & \text{if } \sigma \text{ is the initial state for } \mathcal{P} \\ 0 & \text{otherwise;} \end{cases}$$

- the transition function is deterministic and is

$$T(\langle\sigma,\mathcal{P}\rangle, a) \doteq \begin{cases} \langle\sigma',\mathcal{P}\rangle & \text{if } a \neq \bar{m} \text{ and } \sigma' = \nu(\sigma,a) \\ \bar{d} & \text{if } a = \bar{m} \text{ and } \langle\sigma,\mathcal{P}\rangle \in S_d \end{cases}$$

where $S_d \doteq \{s \in S | s = \langle\sigma,\mathcal{P}\rangle \text{ and } \alpha(\sigma) = \varnothing\}$ is the set of dead ends (states without applicable events) for the planning problems;

- the terminal states of the MDP are $S_g \cup \{\bar{d}\}$ where $S_g \doteq \{s \in S | s = \langle\sigma,\mathcal{P}\rangle \text{ and } \sigma \text{ is a goal state for } \mathcal{P}\}$ is the set of goal states of the planning problems;
- the reward function $R(s,a,s')$ and the discount rate γ will be defined in the next section, because there are multiple possibilities.

The overarching idea is to use RL to estimate the optimal value function $V^*_{\mathcal{M}[\mathcal{P}^k_{\mathcal{D}}]}$ and from that estimation, derive an estimation of $h^*_{\mathcal{P}^k_{\mathcal{D}}}$ (or other forms of guidance). For this purpose, any RL algorithm would work, but we use the same RL algorithm as in the base framework, which is an adaptation of value iteration with replay buffer. We borrow from [19] the fixed-size vector representation of the MDP state to be taken as input to our neural network, in order to allow for a fair comparison. The vectorization of an MDP state $\langle\sigma,\mathcal{P}\rangle$ with $\mathcal{P} = \langle\mathcal{O},\mathcal{I},\mathcal{G}\rangle$ contains both the encoding of the planner search state σ and the encoding of the problem $\mathcal{P}$. In particular, the encoding of σ contains not only the predicate values but also the temporal aspects including a digest of the STN χ, as per Definition 2. The encoding of $\mathcal{P}$ consists in a vectorization of the set of objects $\mathcal{O}$ and a vectorization of the set of goals $\mathcal{G}$.

From now on, we will simply write V^* in place of $V^*_{\mathcal{M}[\mathcal{P}^k_{\mathcal{D}}]}$, h^* in place of $h^*_{\mathcal{P}^k_{\mathcal{D}}}$ and $\mathcal{M}$ in place of $\mathcal{M}[\mathcal{P}^k_{\mathcal{D}}]$.

3.1 Binary Reward Schemata

Theoretical Formulation. To formulate our MDP, a possible choice of reward function (called the "binary reward") is:

$$R_{bin}(s,a,s') \doteq \begin{cases} 1 & \text{if } s' \in S_g \\ -1 & \text{if } s \in S_d \\ 0 & \text{otherwise.} \end{cases}$$

The MDP using this reward is indicated as $\mathcal{M}_{bin}$ and is discounted with factor $0 < \gamma < 1$. Moreover, its optimal value function V^*_{bin} is in the range $[-1,+1]$: it is positive for "good" states (states leading to a goal) and negative for states from which goals are unreachable (i.e., dead-end states or states that lead to an infinite branch with no goal states). With this choice, [19] proved that the relationship between the optimal value function for the MDP V^*_{bin} and the optimal heuristic function for the training planning problems is

$$h^*(s) = \begin{cases} \log_\gamma(V^*_{bin}(s)) + 1 & \text{if } V^*_{bin}(s) > 0 \\ +\infty & \text{otherwise} \end{cases} \forall s \in S \setminus \{\bar{d}\}. \quad (1)$$

Intuitively, a state s whose distance from the nearest goal is d has $h^*(s) = d$ and optimal expected discounted return $V^*_{bin}(s) = \gamma^{d-1}$.

Practical Adjustments. When estimating V^*_{bin} with a neural network (or any other function approximator) V^{nn}_{bin}, we make some practical adjustments. Since the MDP state space is potentially infinite, learning episodes are truncated after Δ_{RL} steps (Δ_{RL} is a hyperparameter) and the target for the value function update at episode truncation (we will call this "truncation bootstrap") is

$$V^{nn}_{bin}(s_{t+1}) \doteq 0 \quad \text{when } t = \Delta_{RL} \quad (2)$$

where s_t is the state visited at time t. Here, value zero is chosen as a target because it is lower than the value of all "good" states and higher than the value of all states from which any goal is unreachable.

Due to episode truncation, the transformation function from the value function to the heuristic is adjusted in the following way.

$$h^{nn}_{bin}(s) \doteq \begin{cases} \min(\log_\gamma(V^{nn}_{bin}(s)) + 1, \Delta_H) & \text{if } V^{nn}_{bin}(s) > 0 \\ \Delta_H & \text{if } V^{nn}_{bin}(s) = 0 \\ 2\Delta_H - \min(log_\gamma(-V^{nn}_{bin}(s)), \Delta_H) & \text{otherwise} \end{cases} \quad (3)$$

where $\Delta_H \geq \Delta_{RL}$ is a planning parameter. We cap the heuristic of states with positive V^{nn}_{bin} to Δ_H because we want to always prefer a state from which the goal is reachable albeit far than a state whose value is uncertain due to episode truncation. The capping also serves the purpose of attenuating numerical errors due to the logarithm. Negative values are mapped to finite, although large, heuristic values because we cannot reliably say that a state with a learnt negative value is a dead end since learning can be imperfect. Δ_H represents the depth to which we expect the learned value function to be informative. Since the search space is a tree, the RL algorithm can learn up to a distance Δ_{RL} from the initial state. We can have $\Delta_H \geq \Delta_{RL}$ if the neural network is able to generalize beyond that depth. The value function of "good" states is roughly in $[\gamma^{\Delta_H}, 1]$ and the value function of "bad" states is in $[-1, -\gamma^{\Delta_H}]$; values in $[-\gamma^{\Delta_H}, \gamma^{\Delta_H}]$ are "uncertain" i.e. they are not reliable estimates. The hyperparameter Δ_H needs to be carefully chosen: if it is too small, we lose information due to the capping; too large values may provide heuristic values that are too large compared to the g (distance from initial state) values in the A^* algorithm.

3.2 Counting Reward Schemata

Theoretical Formulation. An alternative choice of reward function, which we call the "counting reward", is the following.

$$R_{cnt}(s,a,s') \doteq -1 \quad \forall s,s' \in S, a \in A \quad (4)$$

and the MDP $\mathcal{M}_{cnt}$ is undiscounted, i.e. $\gamma = 1$. We also need to adapt the MDP transition relation to accommodate dead ends: we add a self-cycle on state $\bar{d}$ (which gives reward -1 as all other transitions) and no longer consider it as a terminal state. The optimal value

function will be in the range $V^*_{cnt} \in (-\infty, 0)$ for "good" states and $V^*_{cnt} = -\infty$ for "bad" states and the mapping from the value function to the heuristic becomes:

$$h^*(s) = -V^*_{cnt}(s) \quad \forall s \in S \setminus \{\bar{d}\}. \tag{5}$$

We do not provide a proof of Equation (5) for space reasons. It is well-known that in RL environments modeling unit cost graphs the reward function (4) provides shortest paths; ours is a generalization for infinite trees.

The advantage of this new reward schema is that we directly learn the heuristic function, which is the function we ultimately want for planning (solving the Bellman equation for the optimal value function with the counting reward is equivalent to solving the Bellman equation for h^*). In addition, we avoid the numerical instability of the logarithmic transformation in Equation (3).

Practical Adjustments. The formulation needs to be adapted when using a function approximator V^{nn}_{cnt} for estimating V^*_{cnt}. The MDP $\mathcal{M}_{cnt}$ described in the previous paragraph is not proper, i.e. not all policies lead to a terminal state, since there are infinite branches of the search tree (as already observed in Section 3.1) and there are the self-loops in dead ends. With $\gamma = 1$ and an improper MDP, the optimal value function can diverge, and this is a problem for learning. To solve this issue, we truncate episodes after Δ_{RL} steps (like we do for the binary reward) and we slightly change the MDP encoding for dead ends: we remove the self-cycle on $\bar{d}$ and consider it again as a terminal state, and we provide a big negative reward on dead ends:

$$R_{cnt}(s, a, s') \doteq \begin{cases} -2\Delta_h & \text{if } s \in S_d \\ -1 & \text{otherwise;} \end{cases}$$

where Δ_h is a learning parameter. When $\Delta_h \to \infty$ this modified MDP becomes equivalent to $\mathcal{M}_{cnt}$.

The target for the value function at episode truncation is:

$$V^{nn}_{cnt}(s_{t+1}) \doteq -\Delta_h \quad \text{when } t = \Delta_{RL}. \tag{6}$$

The range of possible values for V^{nn}_{cnt} is $[-\Delta_{RL}, 0]$ for "good states", $[-\Delta_h - \Delta_{RL}, -\Delta_h]$ for "uncertain" states (due to episode truncation), $[-2\Delta_h - \Delta_{RL}, -2\Delta_h]$ for "bad" states. Notice that if the parameter Δ_h is chosen so that $\Delta_h < \Delta_{RL}$, then the range of "uncertain" states partially overlaps with the other two ranges (in particular, it overlaps with the far "good" states). We need to choose Δ_h neither too big otherwise the uncertain states will hardly ever be extracted from the queue in A^* (and same for the bad states which are not guaranteed to be bad since learning is imperfect), nor too small otherwise bad states will be mixed with good states.

A heuristic is extracted from V^{nn}_{cnt} as in the theoretical case:

$$h^{nn}_{cnt}(s) \doteq -V^{nn}_{cnt}(s) \quad \forall s \in S. \tag{7}$$

We highlight that the counting reward schema is specific for temporal planning domains. The fact that in temporal planning the search space is a tree that we truncate allows us to set $\gamma = 1$ and obtain the relationship (7). In classical planning, instead, it is not possible to set $\gamma = 1$ because there can be cycles in the graph of the state space.

In summary, we considered two reward schemata, resulting in different optimal value functions V^* for the MDP, but both providing the optimal heuristic h^* if transformed properly. For both schemata we showed the theoretical MDP for which the relationship between V^* and h^* can be established. When using function approximation to estimate V^*, we introduced some practical adjustments to account for episode truncation (and also to have bounded value function in the case of the counting reward): for the binary reward, the MDP is the same but the transformation function changes; for the counting reward, we change the MDP encoding (the reward and the self-loop) but we keep the same transformation function.

4 Exploiting Symbolic Heuristics

In this section, we explore three different ways of leveraging symbolic heuristic functions to improve the performance of our learning and planning framework.

Hereinafter, we assume a domain-independent heuristic h_{sym} is given (e.g. h_{ff} or h_{add}): we do not require particular formal properties other than correctness on states from which goals are unreachable: if $h_{sym}(s) = +\infty$, then $h^*(s) = +\infty$. Moreover, we can also use different heuristics for the following three techniques, if they are applied in combination.

In [19], a symbolic heuristic is used in the behavior policy, i.e. for choosing the non-greedy action during learning: instead of choosing a random action as in the standard epsilon-greedy policy, an action is sampled with probability inversely proportional to the heuristic value. Another way to exploit the prior knowledge given by symbolic heuristics is to treat any state s where $h_{sym}(s) = +\infty$ like a dead end: the action $\bar{m}$ is added to s and leads to $\bar{d}$ giving the dead end reward, and all applicable events in s are removed from the MDP.

4.1 Truncation Bootstrap

Our first proposal to leverage symbolic heuristics consists in improving the truncation bootstrap i.e. estimating the value function of states at the boundary of the subtree explored by the RL algorithm. Using h_{sym} yields a potentially better estimate than using a constant value like in Equations (2) and (6). Concretely, we use the following targets when $t = \Delta_{RL}$.

$$V^{nn}_{bin}(s_{t+1}) \doteq \begin{cases} \gamma^{h_{sym}(s_{t+1})-1} & \text{if } h_{sym}(s_{t+1}) < +\infty \\ -1 & \text{otherwise} \end{cases}$$

$$V^{nn}_{cnt}(s_{t+1}) \doteq \begin{cases} -h_{sym}(s_{t+1}) & \text{if } h_{sym}(s_{t+1}) < +\infty \\ -2\Delta_h & \text{otherwise} \end{cases}$$

Notice that this implies that the concept of "uncertain" interval of values disappears, in the sense that the value function of such states falls in either the "good" or the "bad" interval depending on the symbolic heuristic.

We highlight that the truncation bootstrap is more relevant in temporal planning than in classical planning because the search tree is unexplored beyond depth Δ_{RL}. In classical planning, one might use the value estimated by the neural network at s_{t+1} as target for updating the value at s_t, while we empirically observed that this does not work for temporal planning.

4.2 Residual Learning

The aim of [19] is to learn the optimal heuristic function eagerly from scratch; instead, we propose to exploit h_{sym}, which already tries to approximate h^*, as a basis on top of which we learn a residual. Concretely, we learn an additive residual r^{nn} (dependent on the reward schema) of the optimal value function with respect to the symbolic heuristic. The intuition is that the function to learn r^{nn} will be more regular than V^{nn} since part of the complexity of h^* is already modeled by h_{sym}, and thus the learning task will be simplified.

With the $\mathcal{M}_{bin}$ formulation (Section 3.1), we set

$$V^*_{bin}(s) = r^*_{bin}(s) + \phi_{bin}(s)$$

where r^*_{bin} represents the learnable part of the value function and ϕ_{bin} is the fixed symbolic part defined as:

$$\phi_{bin}(s) \doteq \begin{cases} \gamma^{h_{sym}(s)-1} & \text{if } h_{sym}(s) < +\infty \\ -1 & \text{otherwise.} \end{cases} \quad (8)$$

Intuitively, Equation (8) is essentially the inverse of Equation (1).

As per the following proposition, if the h_{sym} were perfect, then the perfect residual would be the constant zero function, except for states with negative value. This is because h^* does not estimate the distance to dead ends whereas V^*_{bin} does.

Proposition 1. *If $h_{sym} = h^*$ then $r^*_{bin}(s) = 0 \quad \forall s \in \{s \in S | V^*_{bin}(s) > 0\}$.*

Since the symbolic heuristic is in reality an approximation of h^*, the residual will be either positive or negative depending on whether h_{sym} underestimates or overestimates h^*. In practice, we set $V^{nn}_{bin}(s) \doteq r^{nn}_{bin}(s) + \phi_{bin}(s)$ where V^{nn}_{bin} is the neural network output which estimates the correction of ϕ_{bin}.

Analogously, with the counting reward on $\mathcal{M}_{cnt}$, the following equations and proposition hold.

$$V^*_{cnt}(s) = r^*_{cnt}(s) + \phi'_{cnt}(s)$$
$$\phi'_{cnt}(s) \doteq -h_{sym}(s) \quad \forall s \in S$$

Proposition 2. *If $h_{sym} = h^*$ then $r^*_{cnt}(s) = 0 \quad \forall s \in S$.*

With R_{cnt}, learning a residual of the value function is equivalent to learning a residual of the heuristic function: thanks to Equation (5), $h^*(s) = h_{sym}(s) - r^*_{cnt}(s)$. With R_{bin}, instead, we learn a residual of the value function, not of the heuristic function directly and we need to account for the inverse of the logarithmic transformation.

For the "practical" MDP using R_{cnt}, we set

$$V^{nn}_{cnt}(s) \doteq r^{nn}_{cnt}(s) + \phi_{cnt}(s)$$
$$\phi_{cnt}(s) \doteq \begin{cases} \phi'_{cnt}(s) & \text{if } h_{sym}(s) < +\infty \\ -2\Delta_h & \text{otherwise.} \end{cases}$$

Instead of viewing learning a residual as a way of fixing part of the value function estimation thanks to a symbolic heuristic, there is an alternative interpretation proposed by [15]. The authors show that learning an additive residual of the value function with respect to some potential function (computed from the symbolic heuristic) is equivalent to learning the full value function of the MDP when adding a term to the reward function which depends on the potential function. Therefore, learning a residual can be seen as providing a dense reward computed from an existing heuristic. This can mitigate the issue of sparse reward which is inherent of the MDP encoding of planning, and accelerate the reinforcement learning process since the agent reaches goals more quickly during training. The reward function adopted by [15] is similar to our counting reward, with the difference that they set $\gamma < 1$. For temporal planning we can set $\gamma = 1$ since the search space is a tree that we truncate at depth Δ_{RL}, whereas in classical planning there can be regions of the search space where the RL agent loops indefinitely without reaching a goal nor a dead end and thus the MDP is not proper. As already presented, setting $\gamma = 1$ gives us the property that learning a residual of the value function is the same as learning a residual of the heuristic; instead, they need to use a transformation function on the heuristic function due to discounting, like we do for the binary reward.

Algorithm 1 Multiqueue search algorithm

```
1: procedure SEARCH(w)
2:     I ← GETINIT( );  g(I) ← 0;  i ← 0
3:     Q ← [ NEWPRIORITYQUEUE( ), NEWPRIORITYQUEUE( ) ]
4:     PUSH(Q[0], I, h_sym(I))          ▷ Q[0] is sorted as a weighted A* open list
5:     PUSH(Q[1], I, u(I))              ▷ Q[1] is sorted as a GBFS queue with ranking u
6:     while |Q[0]| > 0 do
7:         c ←POPMIN(Q[i])
8:         i ← (i + 1) mod 2
9:         REMOVE(Q[0], c);  REMOVE(Q[1], c)   ▷ Remove c from both queues
10:        if c is a goal state then return GETPLAN(c)
11:        else for all s ∈ SUCC(c) do
12:            g(s) ← g(c) + 1
13:            PUSH(Q[0], s, (1 − w) × g(s) + w × h_sym(s))
14:            PUSH(Q[1], s, u(s))
```

4.3 Multiqueue and Ranking Function

Given the learned value function, [19] transforms it into a heuristic that is used in a weighted A^* search schema. Here, we propose an alternative approach to exploit the learned value function for planning.

The overall idea is to adopt a multiple-queue planning technique [18] to balance learned information with systematic search guided by a symbolic heuristic h_{sym}. The learned information is in the form of a *search state ranking* that is a function that can order any pair of search states, and it is used in a Greedy Best-First Search schema. Concretely, we directly use the value function synthesized by the RL algorithm as a means of ranking the search states generated by the planner, and we use a round-robin approach to either pick from the queue of states ordered by the standard planning heuristic or from the queue ordered according to the learned ranking.

Definition 5. *Given a bounded planning problem set $\mathcal{P}^k_{\mathcal{D}}$, learning a search-state ranking means approximating a function $u : \bigcup_{\mathcal{P} \in \mathbb{P}^k_{\mathcal{D}}} \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle} \times \mathbb{P}^k_{\mathcal{D}} \to \mathbb{R}$ such that for any pair of search states $s, s' \in \mathcal{S}_{\langle \mathcal{D}, \mathcal{P} \rangle}$ for some $\mathcal{P} \in \mathcal{P}^k_{\mathcal{D}}$, if $h^*_{\mathcal{P}^k_{\mathcal{D}}}(s) < h^*_{\mathcal{P}^k_{\mathcal{D}}}(s')$, then $u(s) < u(s')$.*

A search-state ranking function u for two search states s and s' such that $u(s) < u(s')$ indicates that the ranking regards s as a more promising state than s'. Note that the absolute value of a ranking function $u(s)$ gives no information if not compared with another value of the ranking function, like $u(s')$. In this sense, a ranking function is less versatile than a heuristic and cannot be directly used in a search schema like A^*. However, we can use a learned search-state ranking function u in a multi-queue search schema as shown in Algorithm 1. The algorithm hybridizes a (weighted) A^* queue $Q[0]$, ordered according to h_{sym}, and a $GBFS$ queue $Q[1]$, ordered according to u. In the algorithm, the state expansion alternatively selects from either queue and removes the extracted element from both queues; moreover, each successor state is added to both queues. This allows us to combine the greedy exploitation of the learned information with an explorative, systematic search.

For both our reward schemata, we can extract a search-state ranking function from the value function by simply inverting the value function itself. In symbols, $u_{bin}(s) = -V^*_{bin}(s)$ and $u_{cnt}(s) = -V^*_{cnt}(s)$. This fact is obvious for R_{cnt}, because $u_{cnt}(s) = -V^*_{cnt}(s) = h^*(s)$. For R_{bin}, it is easy to see from Equation (1) that if $V^*_{bin}(s) < V^*_{bin}(s')$, then $h^*(s) > h^*(s')$ (because $\gamma < 1$).

Intuitively, the multi-queue approach can make planning more robust since the symbolic heuristic can compensate for some errors in the learnt value function: when learning has some local imperfections and drifts the search away from goals, the symbolic heuristic can help to escape from such regions of the search space.

5 Implementation Details

Neural Network Architecture. We use the same neural network architecture as [19] to allow for a fair comparison, and the search states (as per Definition 2) are encoded into vectors of real numbers of fixed size. The only difference is the activation function in the output layer, due to the different range of the value function in the case of the counting reward and/or the residual. Given the network output x after the last linear layer, with the binary reward we apply the activation function $y(x) = \omega(x)$, where ω indicates the sigmoid function so that we obtain the output range $y \in [-1, +1]$. With the counting reward we apply the activation $y(x) = (\omega(x) - 1) \cdot \frac{3}{2}\Delta_h$ instead, so that $y \in [-3\Delta_h, 0]$. When learning a residual, in the case of the binary reward we change the activation to $y(x) = \frac{3}{2}\omega(x) - \frac{1}{2}$ so that the output range becomes $y \in [-2, +1]$. In this way, when ϕ_{bin} is close to 1 we can correct the value function towards negative values up to -1 with a residual r^{nn}_{bin} of -2. Vice versa, when ϕ_{bin} is -1, the value function can be corrected towards zero with a residual of up to $+1$. Notice that we need not have $r^{nn}_{bin} > +1$ because if ϕ_{bin} is negative we are sure that V^{nn}_{bin} is negative as well. In the case of the counting reward, a similar reasoning leads to consider an activation $y(x) = (2\omega(x) - 1) \cdot \Delta_h$ so that $y \in [-3\Delta_h, +\Delta_h]$.

Learned Heuristic. When the neural network output represents a residual, the learned value function obtained by combining the residual with the symbolic part ϕ may not be in the correct range, and thus we truncate it as follows before transforming it into a heuristic for planning. In the case of the binary reward, the value can be outside the range $[-1, +1]$ and so we clip values above 1 or below -1 before applying Equation (3). In the case of the counting reward, we clip the positive values to 0 before applying Equation (7). Another practical adjustment is that when $h_{sym}(s) = +\infty$, we set $h^{nn}(s) = +\infty$ without computing the residual. Finally, before computing the predicted value function of a successor state s we always check whether s is a goal state, and in that case we set $h^{nn}(s) = 0$, since the neural network is not guaranteed to output zero value for goal states.

6 Experiments

For our experimental evaluation, we consider an adaptation of the two benchmark planning domains used in [19]: namely, the "MAJSP" and "Kitting" domains, and of the "MatchCellar" IPC domain. All the domains considered are temporally expressive [9]. MAJSP consists of a job-shop scheduling problem in which a fleet of moving agents transport items and products between operating machines. In MatchCellar, a set of fuses have to be mended while light is provided by a match to be lit. In Kitting, some robots have to collect several components distributed in different locations of a warehouse in order to compose a kit and then deliver it to a specific location synchronizing with a human operator. We created 655 instances of MAJSP, 725 of Kitting and 575 of MatchCellar.

We implemented the learning part of our framework in Python3 and the rollout simulator in Rust. We adopt the PyTorch framework [20] for training the neural networks. We use a self-implemented planner inspired from TAMER [28] and written in Rust. The learning process takes in input the training instances and outputs the trained value function as a neural network. In the learning algorithm we set the following hyperparameters: $\gamma = 0.99$ for R_{bin}, the maximum size of the replay buffer is 50K, the batch size is 1000, the maximum depth for an episode is $\Delta_{RL} = 200$ for MAJSP, 100 for Kitting and 150 for MatchCellar (due to the different expected size of plans), the learning rate of the optimizer is 10^{-4}, $\Delta_h = 100$ for MAJSP, 50 for Kitting and 75 for MatchCellar. In the planning algorithm, we set $w = 0.8$ for A^* algorithm and Algorithm 1, $\Delta_H = 600$ for MAJSP, 300 for Kitting and 450 for MatchCellar.

To measure the effectiveness of our framework, we performed a 5-fold cross validation: for each domain, we generated the set of ground instances and we randomly partitioned it into 5 equally sized subsamples. In turn, we use each subsample as testing data for the planning part, and the remaining 4 subsamples as training data for learning. We repeat the learning process twice, resulting in 10 total runs.

All the experiments have been conducted on a AMD CPU EPYC 7413 2.65GHz; we imposed a 600s/20GB time/memory limit for executing all the planning approaches. The learning algorithm has been executed for 25K, 50K and 100K episodes and the total training time was approximately equivalent to 225 days of computation on a single-core machine. Benchmarks, code and all the plots are available in the additional material of this paper [3].

We experimented with eight learning techniques, corresponding to the combinations of using the binary reward (Section 3.1) or the counting reward (Section 3.2), bootstrapping at episode truncation with a constant value or with the symbolic heuristic (Section 4.1), and learning a residual or learning from scratch (Section 4.2). For each of these eight learning techniques, we experimented with the two planning techniques of A^* with h^{nn} or the multi-queue (Section 4.3). We used h_{ff} as symbolic heuristic for all the three techniques of Section 4. We also compare against the baseline performance of our planner equipped with h_{ff} without learned information.

Figure 2 reports the coverage results for our three domains. The figure is read like a table with eight cells whose rows correspond to the configurations binary reward vs. counting reward, and whose columns are the combinations of the techniques truncation bootstrap with a constant value or with the heuristic, and learning from scratch (denoted with "Full") or learning a residual. In each of the eight cells, the x axis is the number of episodes, the y axis is the coverage. The vertical lines on top of the bars represent the standard deviation, computed by considering the two runs for each set. The blue bars are the coverage obtained with the single queue algorithm and orange bars are obtained with the multi queue algorithm. The horizontal dashed line is the h_{ff} coverage, whose value is also reported in the top left corner of the figure. In the Kitting and MAJSP domains, all combinations of techniques are able to beat the fully symbolic planner. The striped bars represent the coverage of the baseline which is a re-implementation of the approach by [19] (top leftmost cell) and the coverage obtained when all our contributions are applied (bottom rightmost cell). Remarkably, the latter is consistently higher than the former for all number of episodes in the case of MAJSP and MatchCellar, and comparable in the case of Kitting.

We now analyze the single techniques in isolation. The counting reward performs better than the binary reward in Kitting and MAJSP in almost all cases; in the MatchCellar domain the situation is more scattered. The impact of the heuristic bootstrap is not decisive but positive in many cases. The residual techniques perform better than the eager techniques across the spectrum. It is important to note that when learning a residual, the planner is able to reach a good performance with a much lower number of episodes, hence reducing training time. This can be also seen from the learning curves, which plot the fraction of problems solved during training: learning a residual is much faster than learning from scratch. Learning curves for Kitting with counting reward and heuristic bootstrap are reported in Figure 3. All other learning curves can be found in the additional material.

In the MAJSP and MatchCellar domains, the multiqueue approach consistently performs better than the single queue for all com-

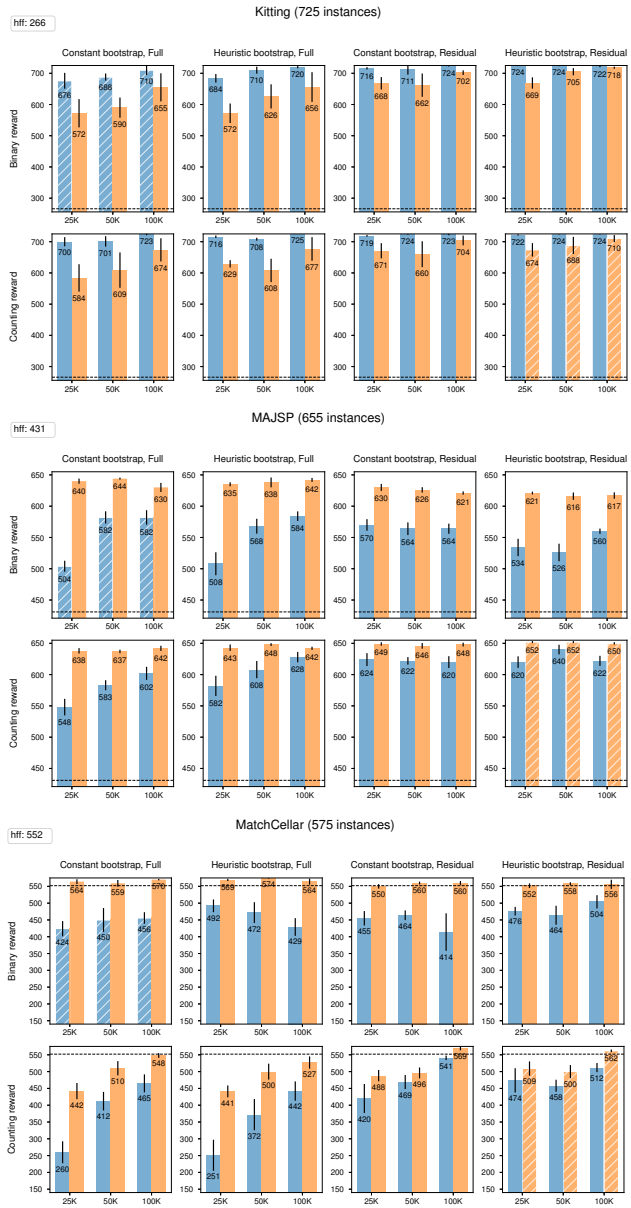

Figure 2. Experimental coverage

binations of techniques. In Kitting, instead, it performs worse; this could be due to the fact that the single queue coverage is already very close to the perfect coverage and the computation of the symbolic heuristic just adds overhead at planning time; furthermore, h_{ff} alone is very weak in the Kitting domain, therefore the combination of learned heuristic with symbolic heuristic brings no benefit.

7 Related Work

Nowadays, combining automated planning with Machine Learning (ML) techniques is a very hot topic. However, with the exception of [19], which we already discussed at length, and [4], all the other works we are aware of focus on classical planning. ML can be used for different purposes in automated planning. Some authors (e.g. [2, 22]) tackle the problem of learning symbolic planning models from data; others (e.g. [27, 25, 23]) aim to learn generalized policies. Furthermore, [21] aims at estimating the cost of a plan from instance

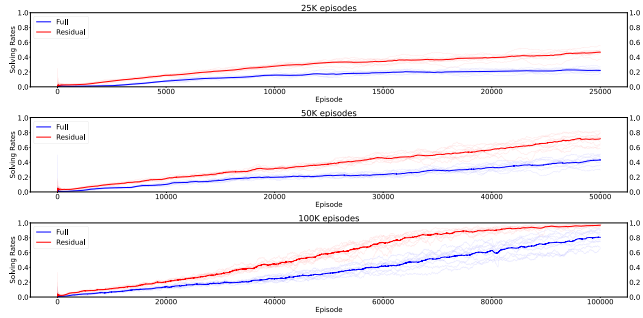

Figure 3. Learning curves for Kitting, counting reward, heuristic bootstrap: residual (in red) vs. full (in blue). Y-axis is the fraction of episodes that reached a goal in the previous 1000 episodes. We plot a semi-transparent line for each cross-validation set and each run; the bold line is their average.

features, while [29] learns to compose multiple symbolic heuristics. In [4] a factored policy is learned with policy gradient RL for solving concurrent probabilistic temporal planning problems.

In this paper, we focus on learning heuristics for search-based temporal planners. In this area, several papers perform supervised learning from a collection of plans to regress a heuristic function. [10] exploit a case-based database to construct planning heuristics, while learning of control policies for classical planning is studied in [30]. In [1], the search spaces generated by a classical planning heuristic are used to learn an incrementally better one. [12] showed a comprehensive hyper-parameter experimentation for supervised learning of a classical planning heuristic. More recently, both Graph Neural Networks [5] and classical machine learning [6] have been used to learn heuristics for classical and numeric planning from a database of example plans. Differently from all these works, which are limited to classical planning, we tackle expressive temporal planning with intermediate conditions and effects and provide a fully-automated technique to learn heuristics from simulations via RL.

Similarly to our work, but differently in terms of expressiveness, [15] propose the use of symbolic heuristics as dense reward generators that improve the sample efficiency of RL for classical planning. Their MDP formulation is similar to our counting reward schema with two key differences: first, it is discounted (because the search space in classical planning is not a tree as in temporal planning); second, they truncate the episode upon reaching a dead end, whereas we appropriately modify the MDP encoding to deal with dead ends.

Finally, [14] learn a classical planning state ranking function for use in a multi-queue planner. Differently from our work, they operate in a supervised setting, learning from a database of plans; instead we use RL to make the learning process fully automatic.

8 Conclusion

In this paper, we discussed how to exploit the information provided by a symbolic heuristic in a RL schema aimed at synthesizing guidance for search-based temporal planners. We use symbolic heuristics during learning, to mitigate the issues caused by episode truncation, and to learn a residual of an existing heuristic instead of learning the whole heuristic function from scratch; moreover, we exploit symbolic heuristics in planning to balance the greedy exploitation of the learned guidance with systematic search, in a multi-queue approach.

For future work, we plan to relax the assumption on the maximum number of objects by adapting recent results using Graph Neural Networks (GNN) [25] to the temporal planning case, and to experiment with multiplicative residuals instead of additive residuals.

Acknowledgements

This work has been supported by the STEP-RL project funded by the European Research Council under GA n. 101115870. This work has been carried out while Irene Brugnara was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with Fondazione Bruno Kessler.

References

[1] S. J. Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[2] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, pages 6094–6101. AAAI Press, 2018.

[3] I. Brugnara, A. Valentini, and A. Micheli. Additional material for "Exploiting Symbolic Heuristics for the Synthesis of Domain-Specific Temporal Planning Guidance using Reinforcement Learning". https://github.com/fbk-pso/step-rl, 2025.

[4] O. Buffet and D. Aberdeen. The factored policy-gradient planner. *Artificial Intelligence*, 173(5-6):722–747, 2009.

[5] D. Z. Chen, S. Thiébaux, and F. W. Trevizan. Learning domain-independent heuristics for grounded and lifted planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20078–20086. AAAI Press, 2024.

[6] D. Z. Chen, F. W. Trevizan, and S. Thiébaux. Return to tradition: Learning reliable heuristics with classical machine learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 68–76. AAAI Press, 2024.

[7] A. Coles and A. Coles. Have I been here before? State memoization in temporal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, pages 97–105. AAAI Press, 2016.

[8] A. J. Coles, A. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, pages 42–49, 2010.

[9] W. Cushing, S. Kambhampati, Mausam, and D. S. Weld. When is temporal planning really temporal? In *Proceedings of the International Joint Conference on Artifical Intelligence*, pages 1852–1859, 2007.

[10] T. de la Rosa, A. G. Olaya, and D. Borrajo. Using cases utility for heuristic planning improvement. In *International Conference on Case-Based Reasoning*, volume 4626 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2007.

[11] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[12] P. Ferber, M. Helmert, and J. Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2346–2353. IOS Press, 2020.

[13] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[14] C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez. Learning to rank for synthesizing planning heuristics. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 3089–3095. IJCAI/AAAI Press, 2016.

[15] C. Gehring, M. Asai, R. Chitnis, T. Silver, L. P. Kaelbling, S. Sohrabi, and M. Katz. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 588–596. AAAI Press, 2022.

[16] M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[17] N. Gigante, A. Micheli, A. Montanari, and E. Scala. Decidability and complexity of action-based temporal planning over dense time. *Artificial Intelligence*, 307:103686, 2022.

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

[19] A. Micheli and A. Valentini. Synthesis of search heuristics for temporal planning via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11895–11902. AAAI Press, 2021.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035. Curran Associates, Inc., 2019.

[21] F. Percassi, A. E. Gerevini, E. Scala, I. Serina, and M. Vallati. Improving domain-independent heuristic state-space planning via plan cost predictions. *Journal of Experimental and Theoretical Artificial Intelligence*, 35(6):849–875, 2023.

[22] I. D. Rodriguez, B. Bonet, J. Romero, and H. Geffner. Learning first-order representations for planning from black box states: New results. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–548, 2021.

[23] N. Rossetti, M. Tummolo, A. E. Gerevini, L. Putelli, I. Serina, M. Chiari, and M. Olivato. Learning general policies for planning through GPT models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 500–508. AAAI Press, 2024.

[24] D. Smith, J. Frank, and W. Cushing. The anml language. In *The ICAPS Workshop on Knowledge Engineering for Planning and Scheduling*, 2008.

[25] S. Ståhlberg, B. Bonet, and H. Geffner. Learning general policies with policy gradient methods. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 647–657, 2023.

[26] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

[27] S. Toyer, F. W. Trevizan, S. Thiébaux, and L. Xie. Action schema networks: Generalised policies with deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, pages 6294–6301. AAAI Press, 2018.

[28] A. Valentini, A. Micheli, and A. Cimatti. Temporal planning with intermediate conditions and effects. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9975–9982. AAAI Press, 2020.

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

[30] S. W. Yoon, A. Fern, and R. Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9: 683–718, 2008.