Learning Heuristic Selection with Dynamic Algorithm Configuration

David Speck[1,*], **André Biedenkapp**[1,*], **Frank Hutter**[1,2],
Robert Mattmüller[1], **Marius Lindauer**[3]
[1]University of Freiburg, [2]Bosch Center for Artificial Intelligence, [3]Leibniz University Hannover
⟨speckd, biedenka, fh, mattmuel⟩@cs.uni-freiburg.de, lindauer@tnt.uni-hannover.de

Abstract

A key challenge in satisficing planning is to use multiple heuristics within one heuristic search. An aggregation of multiple heuristic estimates, for example by taking the maximum, has the disadvantage that bad estimates of a single heuristic can negatively affect the whole search. Since the performance of a heuristic varies from instance to instance, approaches such as algorithm selection can be successfully applied. In addition, alternating between multiple heuristics during the search makes it possible to use all heuristics equally and improve performance. However, all these approaches ignore the internal search dynamics of a planning system, which can help to select the most helpful heuristics for the current expansion step. We show that dynamic algorithm configuration can be used for dynamic heuristic selection which takes into account the internal search dynamics of a planning system. Furthermore, we prove that this approach generalizes over existing approaches and that it can exponentially improve the performance of the heuristic search. To learn dynamic heuristic selection, we propose an approach based on reinforcement learning and show empirically that domain-wise learned policies, which take the internal search dynamics of a planning system into account, can exceed existing approaches in terms of coverage.

Introduction

Heuristic forward search is one of the most popular and successful techniques in classical planning. Although there is a large number of heuristics, it is known that the performance, i.e. the informativeness, of a heuristic varies from instance to instance (Wolpert and Macready 1995; Droste, Jansen, and Wegener 2002). While in optimal planning it is easy to combine multiple admissible heuristic estimates using the maximum, in satisficing planning the estimates of inadmissible heuristics are difficult to combine in general (Röger and Helmert 2010). The reason for this is that highly inaccurate and uninformative estimates of a heuristic can have a negative effect on the entire search process when aggregating all estimates. Therefore, an important task in satisficing planning is to utilize multiple heuristics within one heuristic search.

Röger and Helmert (2010) introduced the idea of a search with multiple heuristics, maintaining a set of heuristics, each associated with a separate open list to allow switching between such heuristics. This bypasses the problem of aggregating different heuristic estimates, while the proposed alternating procedure uses each heuristic to the same extent. Another direction is the selection of the best algorithm or heuristic a priori based on the characteristics of the present planning instance (Seipp et al. 2012; Cenamor, de la Rosa, and Fernández 2016; Sievers et al. 2019). In other words, different search algorithms and heuristics are part of a portfolio from which one is selected to solve a particular problem instance. The automated procedure for performing the former is referred to as algorithm selection (Rice 1976) while optimization of algorithm parameters is referred to as algorithm configuration (Ansótegui, Sellmann, and Tierney 2009; Hutter et al. 2009; López-Ibáñez et al. 2016). Both methodologies have been successfully applied to planning (Fawcett et al. 2011; 2014; Seipp et al. 2015; Sievers et al. 2019) and various other areas of artificial intelligence such as machine learning (Snoek, Larochelle, and Adams 2012) or satisfiability solving (Hutter et al. 2017). However, algorithm selection and configuration ignore the non-stationarity of which configuration performs well. In order to remedy this, Biedenkapp et al. (2020) showed that the problem of selecting and adjusting configurations during the search based on the current solver state and search dynamics can be modelled as contextual Markov decision processes and addressed by standard reinforcement learning methods.

In planning, there is only little work that take into account the search dynamics of a planner to decide which planner to use. Cook and Huber (2016) showed that switching between different heuristic searches (planners) based on the search dynamics obtained during a search leads to better performance than a static selection of a heuristic. However, in this approach, several disjoint searches (planners) are executed, which do not share the search progress (Aine and Likhachev 2016). Ma et al. (2020) showed that a portfolio-based approach that can switch the planner at halftime, depending on the performance of the previously selected one, can improve

[*]Contact Author, Equal contribution

performance over a simple algorithm selection at the beginning. Recent works have investigated switching between different search strategies depending on the internal search dynamics of a planner (Gomoluch, Alrajeh, and Russo 2019; Gomoluch et al. 2020). One approach that shares the search progress is to maintain multiple heuristics as separate open lists (Röger and Helmert 2010). Furthermore, it has been shown that boosting, i.e., giving preference to heuristics that have recently made progress, can improve search performance (Richter and Helmert 2009). While in these works heuristic values are computed for each state, Domshlak, Karpas, and Markovitch (2010) investigated the question, whether the time spent for the computation of the heuristic value for a certain state pays off.

Another avenue of work considers how to "directly" create or learn new heuristic functions. One example is the work of Ferber, Helmert, and Hoffmann (2020), which utilizes supervised learning to learn a heuristic function where the input is the planning (world) state itself. Further, Virseda, Borrajo, and Alcázar (2013) used regression techniques to automatically learn good combinations of heuristics, which can lead to an informative heuristic estimate, but which are also learned a priori and do not adapt to current search dynamics and progress. Finally, Thayer, Dionne, and Ruml (2011) showed that admissible heuristics can be transformed online, into inadmissible heuristics, which makes it possible to tailor a heuristic to a specific planning instance.

In this work we introduce and define dynamic algorithm configuration (Biedenkapp et al. 2020) for planning by learning a policy that dynamically selects a heuristic within a search based on the current search dynamics. We prove that a dynamic adjustment of heuristic selection during the search can exponentially improve the search performance of a heuristic search compared to a static heuristic selection or a *non-adaptive* policy like alternating. Furthermore, we show that such a dynamic control policy is a strict generalization of other already existing approaches to heuristic selection. We also propose a set of state features describing the current search dynamics and a reward function for training a reinforcement learning agent. Finally, an empirical evaluation shows that it is possible to learn a dynamic control policy on a per-domain basis that outperforms approaches that do not involve search dynamics, such as ordinary heuristic search with a single heuristic and alternating between heuristics.

Background

We first introduce classical planning, then discuss greedy best-first search with multiple heuristics, and finally present the concept of dynamic algorithm configuration based on reinforcement learning. Note that the terminology and notation of planning and reinforcement learning are similar, so we use the symbol $\sim$ for all notations directly related to reinforcement learning; e.g. π denotes a plan of a planning task, while $\tilde{\pi}$ is a policy obtained by reinforcement learning.

Classical Planning

A problem instance or task in classical planning, modeled in the SAS$^+$ formalism (Bäckström and Nebel 1995), is a tuple $i = \langle \mathcal{V}, s_0, \mathcal{O}, s_\star \rangle$ consisting of four components. $\mathcal{V}$ is a finite set of state variables, each associated with a finite domain D_v. A fact is a pair (v, d), where $v \in \mathcal{V}$ and $d \in D_v$, and a partial variable assignments over $\mathcal{V}$ is a consistent set of facts, i.e. a set that does not contain two facts for the same variable. If s assigns a value to each $v \in \mathcal{V}$, s is called a state. States and partial variable assignments are functions which map variables to values, i.e. $s(v)$ is the value of variable v in state s (analogous for partial variable assignments). $\mathcal{O}$ is a set of operators, where an operator is a pair $o = \langle pre_o, \mathit{eff}_o \rangle$ of partial variable assignments called preconditions and effects, respectively. Each operator has cost $c_o \in \mathbb{N}_0$. The state s_0 is called the initial state and the partial variable assignment $s_\star$ specifies the goal condition, which defines all possible goal states $S_\star$. With $\mathcal{S}$ we refer to the set of all states defined over $\mathcal{V}$, and with $|i|$ we refer to the size of the planning task i, i.e. the number of operators and facts.

We call an operator $o \in \mathcal{O}$ applicable in state s iff pre_o is satisfied in s, i.e. $s \models pre_o$. Applying operator o in state s results in a state s' where $s'(v) = \mathit{eff}_o(v)$ for all variables $v \in \mathcal{V}$ for which eff_o is defined and $s'(v) = s(v)$ for all other variables. We also write $s[o]$ for s'. The objective of classical planning is to determine a plan, which is defined as follows. A *plan* $\pi = \langle o_0, \ldots, o_{n-1} \rangle$ for planning task i is a sequence of applicable operators which generates a sequence of states $s_0, \ldots, s_n$, where $s_n \in S_\star$ is a goal state and $s_{i+1} = s_i[o_i]$ for all $i = 0, \ldots, n-1$. The cost of plan π is the sum of its operator costs.

Given a planning task, the search for a good plan is called satisficing planning. In practice, heuristic search algorithms such as greedy best-first search have proven to be one of the dominant search strategy for satisficing planning.

Greedy Search with Multiple Heuristics

Greedy best-first search is a pure heuristic search which tries to estimate the distance to a goal state by means of a heuristic function. A heuristic is a function $h : \mathcal{S} \mapsto \mathbb{N}_0 \cup \{\infty\}$, which estimates the cost to reach a goal state from a state $s \in \mathcal{S}$. The perfect heuristic $h^\star$ maps each state s to the cost of the cheapest path from s to any goal state $s_\star \in S_\star$. The general idea of greedy best-first search with a single heuristic h is to start with the initial state and to expand the most promising states based on h until a goal state is found (Pearl 1984). During the search, relevant states are stored in an open list that is sorted by the heuristic values of the contained states in ascending order so that the state with the lowest heuristic values, i.e. the most promising state, is at the top. More precisely, in each step a state s with minimal heuristic value is expanded, i.e. its successors $S' = \{s[o] \mid o \in O\}$ are generated and states $s' \in S'$ not already expanded are added to the open list according to their heuristic values $h(s')$. Within an open list, for states with the same heuristic value (h-value) the tie-breaking rule that is used is according to the first-in-first-out principle.

In satisficing planning it is possible to combine multiple heuristic values for the same state in arbitrary ways. It has been shown, however, that the combination of several heuristic values into one, e.g. by taking the maximum or a (weighted) sum, does not lead to informative heuristic estimates (Röger and Helmert 2010). This can be explained

by the fact that if one or more heuristics provide very inaccurate values, the whole expansion process is affected. Röger and Helmert (2010) introduced the idea to maintain multiple heuristics $H = \{h_0, \dots, h_{n-1}\}$ within one greedy best-first search. More precisely, it is possible to maintain a separate open list for each heuristic $h \in H$ and switch between them at each expansion step while always expanding the most promising state of the currently selected open list. The generated successor states are then evaluated with *each* heuristic and added to the corresponding open lists. This makes it possible to share the search progress (Aine and Likhachev 2016). Especially, an alternation policy, in which all heuristics are selected one after the other in a cycle such that all heuristics are treated and used equally, has proven to be an efficient method. Such equal use of heuristics can help to progress the search space towards a goal state, even if only one heuristic is informative. However, in some cases it is possible to infer that some heuristics are currently, i.e. in the current region of the search space, more informative than others, which is ignored by a strategy like alternation. More precisely, with alternation, the choice of the heuristic depends only on the current time step and not on the current search dynamics or planner state. In general, however, it is possible to dynamically select a heuristic based on internal information provided by the planner. This is the key idea behind our approach described in the following.

Dynamic Algorithm Configuration

Automated algorithm configuration (AC) has proven a powerful approach to leveraging the full potential of algorithms. Standard AC views the algorithms being optimized as black boxes, thereby ignoring an algorithm's temporal behaviour and ignoring that an optimal configuration might be non-stationary (Arfaee, Zilles, and Holte 2011).

Dynamic algorithm configuration (DAC) is a new meta-algorithmic framework that makes it possible to learn to adjust the hyperparameters of an algorithm given a description of the algorithm's behaviour (Biedenkapp et al. 2020). We first describe DAC on a high level. Given a parameterized algorithm A with its configuration space $\tilde{\Theta}$, a set of problem instances $\mathcal{I}$ the algorithm has to solve, a state description $\tilde{s}_t^i$ of the algorithm A solving an instance $i \in \mathcal{I}$ at step $t \in \mathbb{N}_0$, and a reward signal $\tilde{r}$ assessing the reward of using a control policy $\tilde{\pi} \in \tilde{\Pi}$ to control A on an instance $i \in \mathcal{I}$ (e.g. runtime or number of state expansions), the goal is to find a (*dynamic*) control policy $\tilde{\pi}^* : \mathbb{N}_0 \times \tilde{\mathcal{S}} \times \mathcal{I} \to \tilde{\Theta}$, that adaptively chooses a configuration $\tilde{\theta} \in \tilde{\Theta}$ given a state $\tilde{s}_t \in \tilde{\mathcal{S}}$ of A at time $t \in \mathbb{N}_0$ to optimize the reward of A across the set of instances $\mathcal{S}$, i.e. $\tilde{\pi}^* \in \arg\max_{\tilde{\pi} \in \tilde{\Pi}} \mathbb{E}[\tilde{r}(\tilde{\pi}, i)]$. Note that the current time step $t \in \mathbb{N}_0$ and instance $i \in \mathcal{I}$ can be encoded in the state description $\tilde{\mathcal{S}}$ of an algorithm A, which leads to a dynamic control policy, which is defined as $\tilde{\pi}_{\text{dac}} : \tilde{\mathcal{S}} \to \tilde{\Theta}$.

Figure 1 depicts the interaction between a control policy $\tilde{\pi}$ and a planning system A schematically. At each time step t, the planner sends the current internal state $\tilde{s}_t^i$ and the corresponding reward $\tilde{r}_t^i$ to the control policy $\tilde{\pi}$ based on which the controller decides which parameter setting $h_{t+1} \in \tilde{\Theta}$ to use. The planner progresses according to the decision to

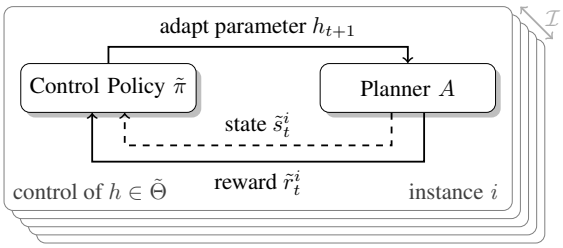

Figure 1: Dynamic configuration of parameter $h \in \tilde{\Theta}$ of algorithm A on an instance $i \in \mathcal{I}$, at time step $t \in \mathbb{N}_0$. Until i is solved or a budget is exhausted, the controller adapts parameter h, based on the internal state $\tilde{s}_t^i$ of A.

the next internal state $\tilde{s}_{t+1}^i$ with reward $\tilde{r}_{t+1}^i$. This formalisation of dynamic algorithm configuration makes it possible to recover prior meta-algorithmic frameworks as special cases which we discuss below.

Dynamic Heuristic Selection

In this section, we will explain how dynamic algorithm configuration can be used in the context of dynamic heuristic selection and how it differs from time-adaptive or in short *adaptive* algorithm configuration and algorithm selection, which have already been used in the context of search with multiple heuristics. Röger and Helmert (2010) introduced the idea of maintaining a set of heuristics H each associated with a separate open list in order to allow the alternation between such heuristics. Considering H as the configuration space $\tilde{\Theta}$ of a heuristic search algorithm A and each state expansion as a time step t, it is possible to classify different dynamic heuristic selection strategies within the framework of algorithm configuration. For example, alternation is an time-adaptive control policy because it maps each time step to a specific heuristic, i.e. configuration, independent of the instance or the state of the planner. The selection of a particular heuristic depending on the current instance before solving the instance, known as "portfolio planner", is an algorithm selection policy that depends only on the instance and not on the current time step or the internal state of the planner. Noteworthy exceptions are policies that compare the heuristic values of states, such as the expansion of the state with the overall minimal heuristic value or according to a Pareto-optimality analysis (Röger and Helmert 2010). Such policies depend on the current state of the planner, but ignore the time step and the current instance being solved. This indicates that all three components — the instance, the time step, and the state of the planner — can be important and helpful in selecting the heuristic for the next state expansion. The following summarizes the existing approaches to heuristic selection within the framework of algorithm configuration.

- *Algorithm Selection*:
  - Policy: $\tilde{\pi}_{\text{as}} : \mathcal{I} \to H$
  - Example: Portfolios (Seipp et al. 2012; Cenamor, de la Rosa, and Fernández 2016; Sievers et al. 2019)

- *Adaptive Algorithm Configuration*:
  - Policy: $\tilde{\pi}_{\text{aac}} : \mathbb{N}_0 \to H$
  - Example: Alternation (Röger and Helmert 2010)
- *Dynamic Algorithm Configuration*:
  - Policy: $\tilde{\pi}_{\text{dac}} : \mathbb{N}_0 \times \tilde{\mathcal{S}} \times \mathcal{I} \to H$
  - Example: Approach proposed in this paper

An Approach based on Reinforcement Learning

In this section, we describe all the parts required to dynamically configure a planning system so that for each individual time step, a dynamic control policy can decide which heuristic to use based on a dynamic control policy. Here, a time step is a single expansion step of the planning system.

State Description Learning dynamic configuration policies requires descriptive state features that inform the policy about the characteristics and the behavior of the planning system in the search space. Preferably, such features are domain-independent, such that the same features can be used for a wide variety of domains. In addition, such state features should be cheap to compute in order to keep the overhead as low as possible. As consequence of both desiderata and the intended learning task we propose to use the following state features computed over the entries contained in the corresponding open list of each heuristic:

$\max_h$**:** maximum h value for each heuristic $h \in H$;

$\min_h$**:** minimum h value for each heuristic $h \in H$;

μ_h**:** average h value for each heuristic $h \in H$;

σ_h^2**:** variance of the h values for each heuristic $h \in H$;

$\#_h$**:** number of entries for each heuristic $h \in H$;

t**:** current time/expansion step $t \in \mathbb{N}_0$.

To measure progress, we do not directly use the values of each state feature, but compute the *difference of each state feature* between successive time steps $t-1$ and t. The configuration space is a finite set of n heuristics to choose from, i.e., $\tilde{\Theta} = H = \{h_0, \ldots, h_{n-1}\}$.

Note that the described set of state features is domain independent, but does not contain any specific context information. In general, however, it is possible to describe an instance or domain with state features that describe, for example, the variables, operators or the causal graph (Sievers et al. 2019). If the goal is to learn robust policies that can handle highly heterogeneous sets of instances, it is possible to add contextual information about the planning instance at hand, such as the problem size or the required preprocessing steps (Fawcett et al. 2014), to the state description. However, in this work, limit ourselves to domain-wise dynamic control policies and show that the concept of DAC can improve heuristic search in theory and practice.

Reward Function Similar to the state description, the reward function we want to optimize should ideally be domain-independent, cheap and quick to compute. Since the goal is usually to quickly solve as many tasks as possible, a good reward feature should reflect this desire.

We use a reward of -1 for each expansion step that the planning system has to perform in order to find a solution. Using this reward function, a configuration policy learns to select heuristics that minimize the expected number of state expansions until a solution is found. This sparse reward function ignores aspects such as the quality of a plan, but its purpose is to reduce the search effort and thus improve search performance. Clearly, it is possible to define other reward functions with, e.g., dense rewards to make learning easier. We nevertheless demonstrate that already with our reward function and state features it is possible to learn dynamic control policies, which dominate algorithm selection and adaptive control policies in theory and practice.

Dynamic Algorithm Configuration in Theory

In optimal planning, where the goal is to find a plan with minimal cost, the performance of heuristic search can be measured by the number of state expansions (Helmert and Röger 2008). This is different for satisficing planning, because plans with different costs can be found and there are generally no "must expand" states that need to be expanded to prove that a solution is optimal. However, the number of state expansion until *any* goal state is found can be used to measure the guidance of a heuristic or heuristic selection (Richter and Helmert 2009; Röger and Helmert 2010).

We want to answer the question of whether it can theoretically be beneficial to use dynamic control policies $\tilde{\pi}_{\text{dac}}$ over algorithm selection policies $\tilde{\pi}_{\text{as}}$ or adaptive control policies $\tilde{\pi}_{aac}$. Proposition 1 proves that for each heuristic search algorithm in combination with each collection of heuristics there is a dynamic control policy $\tilde{\pi}_{\text{dac}}$ which is as good as $\tilde{\pi}_{\text{as}}$ or $\tilde{\pi}_{aac}$ in terms of state expansions. The key insight is that dynamic control policies are a strict generalization of algorithm selection policies and adaptive control policies that always allow the simulation of the former policies.

Proposition 1. *Independent of the heuristic search algorithm and the collection of heuristics, for each algorithm selection policy $\tilde{\pi}_{as}$ and adaptive algorithm configuration policy $\tilde{\pi}_{aac}$ there is a dynamic control policy $\tilde{\pi}_{dac}$ which expands at most as many states as $\tilde{\pi}_{as}$ and $\tilde{\pi}_{aac}$ until a plan is found for a given planning instance.*

Proof. DAC policies generalize algorithm selection and adaptive algorithm configuration policies, thus it is always possible to define $\tilde{\pi}_{\text{dac}}$ as $\tilde{\pi}_{\text{dac}} = \tilde{\pi}_{\text{as}}$ or $\tilde{\pi}_{\text{dac}} = \tilde{\pi}_{\text{aac}}$. □

With Proposition 1 it follows directly that an optimal algorithm configuration policy $\tilde{\pi}^*_{\text{dac}}$ is at least as good as an optimal algorithm selection policy $\tilde{\pi}^*_{\text{as}}$ and an optimal adaptive algorithm configuration policy $\tilde{\pi}^*_{\text{aac}}$.

Corollary 2. *Independent of the heuristic search algorithm and the collection of heuristics, an optimal dynamic control policy $\tilde{\pi}^*_{dac}$ expands at most as many states as an optimal algorithm selection policy $\tilde{\pi}^*_{as}$ and an optimal adaptive algorithm configuration policy $\tilde{\pi}^*_{aac}$ until a plan π is found for a planning task.* □

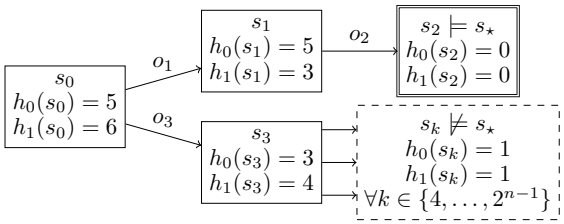

Figure 2: Visualization of the induced transition system of the planning task family i_n.

It is natural to ask the question to what extent the use of a dynamic control policy instead of an algorithm selection or an adaptive control policy can improve the search performance of heuristic search. We will show that for each algorithm selection policy $\tilde{\pi}_{\text{as}}$ and adaptive algorithm configuration policy $\tilde{\pi}_{\text{aac}}$, we can construct a family of planning tasks so that a dynamic control policy $\tilde{\pi}_{\text{dac}}$ will expand exponentially fewer states until a plan is found. For this purpose, we introduce a family of planning instances i_n with $O(n)$ propositional variables and $O(n)$ operators. The induced transition system of i_n is visualized in Figure 2. There is exactly one goal path s_0, s_1, s_2, which is induced by the unique plan $\pi = \langle o_1, o_2 \rangle$. Furthermore, exactly two states are directly reachable from the initial state, s_1 and s_3. While state s_1 leads to the unique goal state s_2, from s_3 onward exponentially many states $s_4, \ldots, s_{2^n-1}$ in $n = |i_n|$, i.e. $\Omega(2^n) = \Omega(2^{|i_n|})$, can be reached by the subsequent application of multiple actions.

Theorem 3. *For each adaptive algorithm configuration policy $\tilde{\pi}_{aac}$ there exists a family of planning instances i_n, a collection of heuristics H and a dynamic control policy $\tilde{\pi}_{dac}$, so that greedy best-first search with H and $\tilde{\pi}_{aac}$ expands exponentially more states in $|i_n|$ than greedy best-first search with H and $\tilde{\pi}_{dac}$ until a plan π is found.*

Proof. Let $\tilde{\pi}_{\text{aac}}$ be an adaptive algorithm configuration policy. Now, we consider the family of planning tasks i_n (Figure 2) with $|i_n| = O(n)$ and a collection of two heuristics $H = \{h_0, h_1\}$. The heuristic estimates of h_0 and h_1 are shown in Figure 2 and the open lists of greedy best-first search at each time step t are visualized in Figure 3. In time step 0, it is irrelevant which heuristic is selected, always leading to time step 1, where state s_3 is the most promising state according to heuristic h_0, while state s_1 is the most promising state according to heuristic h_1. In time step 1, $\tilde{\pi}_{\text{aac}}$ can either select heuristic h_0 or h_1. We first assume that $\tilde{\pi}_{\text{aac}}$ selects h_0 so that state s_3 is expanded, leading to exponentially many states s_k, which are all evaluated with $h_0(s_k) = h_1(s_k) = 1$ and thus are all expanded before s_1. Therefore, the unique goal state s_2 is found after all other states in the state space $\mathcal{S}$ have been expanded.

In comparison, for $\tilde{\pi}_{\text{dac}}$ we can pick the policy that always selects the heuristic with minimum average heuristic value of all states in the corresponding open list, i.e. $\arg\min_{h \in H} \mu_h$. Following $\tilde{\pi}_{\text{dac}}$, first h_0 and then h_1 is selected, generating the goal state s_2 in time step 1. Therefore,

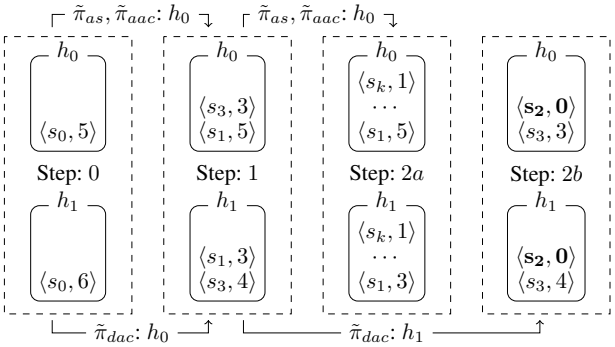

Figure 3: Visualization of two heuristics used to solve an instance of the planning task family i_n.

$\tilde{\pi}_{\text{dac}}$ only expands 2 states, while $\tilde{\pi}_{\text{aac}}$ expands 2^{n-2} states until a goal state is found.

Finally, for a policy $\tilde{\pi}_{\text{aac}}$ that selects h_1 at time step 1, it is possible to swap the heuristic estimates of h_0 and h_1 in the constructed collection of heuristics, resulting in the same number of state extensions. □

Theorem 4. *For each algorithm selection policy $\tilde{\pi}_{as}$ there exists a family of planning instances i_n, a collection of heuristics H and a dynamic control policy $\tilde{\pi}_{dac}$, so that greedy best-first search with H and $\tilde{\pi}_{as}$ expands exponentially more states in $|i_n|$ than greedy best-first search with H and $\tilde{\pi}_{dac}$ until a plan π is found.*

Proof. Let $\tilde{\pi}_{\text{as}}$ be an algorithm selection policy. Now, we consider the family of planning tasks i'_n, which is similar to the family of planning tasks i_n (Figure 2), with one modification: the goal state s_2 is not directly reachable from s_1, but via an additional state s'. In other words, we insert the state s' between s_1 and s_2. Furthermore, we again consider a collection of two heuristics $H = \{h_0, h_1\}$ with the heuristic estimates shown in Figure 2 and $h_0(s') = 2$ and $h_1(s') = 10$. The idea is that both heuristics alone lead to the expansion of exponentially many states, whereas a dynamic switch of the heuristic only leads to constantly many expansions.

Policy $\tilde{\pi}_{\text{as}}$ selects exactly one heuristic, h_0 or h_1, for each planning task. If h_0 is selected, with the same argument used in the proof of Theorem 3, exponentially many states in $|i'_n|$ are expanded. If h_1 is selected, in time step 2, states s_3 and s' are contained in both open lists. According to h_1, state s_3 is more promising than s', which leads again to an expansion of exponentially many states in $|i'_n|$.

In comparison, for $\tilde{\pi}_{\text{dac}}$ we pick again the policy that always selects the heuristic with minimum average heuristic value of all states in the corresponding open list, i.e. $\arg\min_{h \in H} \mu_h$. Policy $\tilde{\pi}_{\text{dac}}$ selects first h_0, followed by h_1 and again h_0, resulting in the generation of the goal state after three state extensions. □

In Theorems 3 and 4 we assume for simplicity that expanded states are directly removed from all open lists. In practice, open lists are usually implemented as min-heaps, and it is costly to search and remove states immediately.

Therefore, states that have already been expanded are kept in the open lists and ignored as soon as they have reached the top. We note that this does not affect the theoretical results.

Finally, it is important to emphasize that Proposition 1, Corollary 2 and Theorems 3 and 4 are theoretical results. All results are based on the assumption that it is possible to learn good dynamic control policies. Next, we show that it is possible in practice to learn such dynamic control policies.

Empirical Evaluation

We conduct experiments[1] to measure the performance of our reinforcement learning (RL) approach on domains of the International Planning Competition (IPC). For each domain, the RL policies are trained on a training set and evaluated on a disjoint prior unseen test set of the same domain. Note that such policies are not domain-independent, although it is generally possible to add instance- and domain-specific information to the state features. We leave the task of learning domain-independent policies for future work.

Setup

All experiments are conducted with FAST DOWNWARD (Helmert 2006) as the underlying planning system. We use ("eager") greedy best-first search (Richter and Helmert 2009) and min-heaps to represent the open lists (Röger and Helmert 2010). Furthermore, we implemented an extension for FAST DOWNWARD, which makes it possible to communicate with a controller (dynamic control policy) via TCP/IP and thus to send relevant information (state features and reward) in each time/expansion step and to receive the selected parameter (heuristic). This architecture allows the planner and controller to be decoupled, making it easy to replace components. We considered four different heuristic estimators as configuration space, i.e. $\tilde{\Theta} = H = \{h_{\text{ff}}, h_{\text{cg}}, h_{\text{cea}}, h_{\text{add}}\}$ which can be changed at each time step:

- h_{ff}: the FF heuristic (Hoffmann and Nebel 2001),
- h_{cg}: the causal graph heuristic (Helmert 2004),
- h_{cea}: the context-enhanced additive heuristic (Helmert and Geffner 2008), and
- h_{add}: the additive heuristic (Bonet and Geffner 2001).

For the evaluation of the test set, i.e. the final planning runs, we used a maximum of 4 GB memory and 5 minutes runtime. All experiments were run on a compute cluster with nodes equipped with two Intel Xeon Gold 6242 32-core CPUs, 20 MB cache and 188GB (shared) RAM running Ubuntu 18.04 LTS 64 bit.

Similar to Biedenkapp et al. (2020), we use ϵ-greedy deep Q-learning in the form of a double DQN (van Hasselt, Guez, and Silver 2016) implemented in CHAINER (Tokui et al. 2019) (CHAINERRL v0.7.0) to learn the dynamic control policies. The networks are trained using ADAM[2] (Kingma and Ba 2014) for 10^6 update steps. In order to avoid bad policies being executed arbitrarily long during training, we use a cutoff of $7\,500$ control/expansion steps. Although some instances are not solved within this cutoff, even with the optimal policy, the underlying assumption is that good policies for smaller instances generalize to larger instances within a domain. Note that it is in general also possible to add a certain time cutoff. To determine the quality of a learned policy, we evaluated it every $30\,000$ steps during training and save the best policy we have seen so far. In total, we performed 5 independent runs of our control policies for each domain, for which we report the average performance. The policies are represented by neural networks for which we determined the hyperparameters in a white-box experiment on a new artifical domain and kept these hyperparameters fixed for all domains in the experiments.

White-Box Experiments. We conducted preliminary experiments on a newly created ARTIFICIAL domain with two artificial heuristics. This domain is designed so that in each step, only one of two heuristics is informative. In other words, similar to the constructed example in the proof of Theorem 4, at each time step, only one heuristic leads to the expansion of a state which is on the shortest path to a goal state. In order to obtain a good control policy that leads to few state expansions, it is necessary to derive a *dynamic* control policy from the state features. We generated 30 training instances on which we performed a small grid search over the following parameters *#layers* $\in \{2, 5\}$, *hidden units* $\in \{50, 75, 150, 200\}$ and *epsilon decay* $\in \{2.5 \times 10^5, 5 \times 10^5\}$. We determined that a 2-layer network with 75 hidden units and a linear decay for ϵ over 5×10^5 steps from 1 to 0.1 worked best.

Interestingly, it was possible to learn policies with a performance close to the optimal policy, see Figure 4. Both individual heuristics perform poorly (even when using an oracle selector). Randomly deciding which heuristic to play performs nearly as good as the alternating strategy that alternates between the heuristics at each step. In the beginning the learned policy needs some time to figure out in which states a heuristic might be preferable. However, it quickly learns to choose the correct heuristic, outperforming all other methods and nearly recovering the optimal policy.

Experiments

We evaluated the performance of our RL approach on six domains of the International Planning Competition (IPC). These domains were chosen because there are instance generators available online[3] that make it possible to create a suitable number of instances of different sizes. Furthermore, instances of these domains usually require a significant number of state expansions in order to find a plan. For this purpose, we generated 200 instances for all domains and randomly divided them into disjoint training and test sets with the same size of 100 instances each. For each domain we trained five dynamic control policies on the training set and compared them with other approaches on the unseen test set. We are mainly interested in comparing different policies for heuristic selection, which is why, here, the planner

[1] Resources: https://github.com/speckdavid/rl-plan

[2] We use CHAINER's v0.7.0 default parameters for ADAM.

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

Algorithm	CONTROL POLICY			SINGLE HEURISTIC				BEST AS (ORACLE)		
Domain (# Inst.)	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}	**RL**	ALT	SINGLE h
BARMAN (100)	**84.4**	83.8	83.3	66.0	17.0	18.0	18.0	**89.0**	84.0	67.0
BLOCKSWORLD (100)	**92.9**	83.6	83.7	75.0	60.0	92.0	92.0	**96.3**	88.0	93.0
CHILDSNACK (100)	**88.0**	86.2	86.7	75.0	86.0	86.0	86.0	**88.0**	**88.0**	86.0
ROVERS (100)	95.2	**96.0**	**96.0**	84.0	72.0	68.0	68.0	**96.0**	**96.0**	91.0
SOKOBAN (100)	87.7	87.1	87.0	88.0	**90.0**	60.0	89.0	88.6	87.0	**92.0**
VISITALL (100)	56.9	51.0	51.5	37.0	**60.0**	**60.0**	**60.0**	**61.4**	52.0	60.0
SUM (600)	**505.1**	487.7	488.2	425.0	385.0	384.0	413.0	**519.3**	495.0	489.0

Table 1: Average coverage of different policies for the selection of a heuristic in each expansion step when evaluating the strategies on the prior unseen *test* set. The first three columns are control policies, the next four are individual heuristic searches, while the last three represent the best algorithm selection of the corresponding strategies, i.e. oracle selector for each instance.

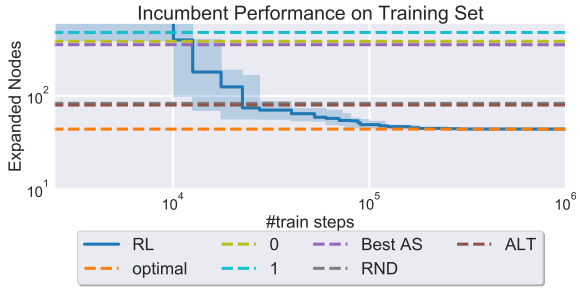

Figure 4: Performance of the best learned policy during training (RL), compared to the the performance of the individual heuristics (0 & 1), the oracle selector (BEST AS), an alternating schedule (ALT), a random policy (RND) and the optimal policy. Dashed lines indicate the performance of our baselines, the solid line the mean performance and the shaded area the standard deviation of our approach.

always maintains all four open lists, even if only one heuristic is used, and the controller, i.e. the dynamic control policy, alone decides which heuristic is selected.

Table 1 shows the percentage of solved instances per domain, i.e. the average coverage, on the test set. Each domain has a score in the range of 0-100, with larger values indicating more solved instances on average. More precisely, it is possible to obtain a score between 0 and 1 for each planning instance. A value of 0 means that the instance was never solved by the approach, 0.5 means that the instance was solved in half the runs, and 1 means that the instance was always solved. These scores are added up to give the average coverage per domain.

The first three columns correspond to control policies. Entry RL is the average coverage of the five trained dynamic control policies based on reinforcement learning, each averaging over 25 runs with different seeds. Entry RND denotes the average coverage of 25 runs, where a random heuristic is selected in each step. Entry ALT stands for the average over all possible permutations of the execution of alternation. Note that there are $4! = 24$ different ways of executing alternation with four different heuristics. The SINGLE HEURISTIC columns show the coverage when only the corresponding heuristic is used. Finally, the columns for selecting the best algorithm selection (BEST AS) stand for the use of an oracle selector, which selects the best configuration of the corresponding technique for each instance. In other words, the best algorithm selection for RL is to choose the best dynamic control policy from the five trained policies for each instance, the best algorithm selection for ALT is to choose the best permutation of alternation for each instance and the best algorithm selection for SINGLE h is to choose the best heuristic for each instance.

The results of Table 1 show that our approach (RL) performs best on average in terms of coverage (individual coverage of the five trained RL policies: 505.4, 500.6, 501.6, 507.4, 510.1). ALT is slightly better than the uniform randomized choice of a heuristic RND, which indicates that the most important advantage of ALT is to use each heuristic equally with frequent switches and not to switch between them systematically. Furthermore, consistent with the results of Röger and Helmert (2010), single heuristics perform worse than the use of multiple heuristics. Interestingly, in the domain VISITALL, single heuristics have the highest coverage and while RND and ALT have a low coverage, RL performs better. This indicates that in this domain, the dynamic control policies of RL were able to infer that a static policy is good or to exclude certain single heuristics. In BLOCKSWORLD, RL has the highest coverage among all approaches. A possible explanation is that a dynamic policy is the key to solving difficult instances in this domain. This assumption is supported by the observation that the best algorithm selection, i.e. the oracle selection of RL, clearly exceeds the other approaches in BLOCKSWORLD. Finally, in ROVER, the use of multiple heuristics seems to be important, and while RL scores better than using single heuristics, the learned policy scores worse than RND and ALT. This may be due to overfitting which we will discuss below.

Considering the columns of best algorithm selection, it is possible to observe that an oracle single heuristic selection or oracle alternating selection would not perform better than the average performance of our learned RL which policies shows that 1) heuristic search with multiple heuristics can in practice benefit from dynamic algorithm configuration and 2) it is possible to learn good dynamic policies domain-wise. Even under the unrealistic circumstances of an *optimal* algorithm selector, our learned policies perform better and thus outperform all possible algorithm selection policies.

Algorithm	CONTROL POLICY			SINGLE HEURISTIC			
Metric	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}
COVERAGE	**84.2**	81.3	81.4	70.8	64.2	64.0	68.8
GUIDANCE	**38.5**	37.4	37.5	30.8	27.6	28.6	30.4
SPEED	**66.6**	62.8	62.8	54.9	50.4	50.3	54.0
QUALITY	**76.2**	76.0	76.0	65.8	57.6	56.2	60.9

(a) Test set

Algorithm	CONTROL POLICY			SINGLE HEURISTIC			
Metric	**RL**	RND	ALT	h_{ff}	h_{cg}	h_{cea}	h_{add}
COVERAGE	**87.0**	83.6	83.0	71.7	64.3	65.0	68.5
GUIDANCE	**39.8**	38.3	38.4	31.4	26.6	28.8	30.2
SPEED	**69.3**	65.3	65.4	56.0	49.1	51.1	54.2
QUALITY	**79.5**	77.9	77.5	66.8	57.3	58.0	61.3

(b) Training set

Table 2: A comparison of different control policies and single heuristic search measuring coverage, guidance, speed and solution quality on the prior unseen *test* set (a) and the *training* set (b). A higher score means better performance for all four metrics.

Table 2 shows four different metrics including the coverage from above. We additionally evaluate the guidance, speed and quality for each approach with a rating scale (Richter and Helmert 2009; Röger and Helmert 2010). For *guidance*, tasks solved within one state expansion get one point, while unsolved tasks or tasks solved with more than 10^6 state expansions get zero points. Between these extremes the scores are interpolated logarithmically. For *speed* the algorithm gets one point for tasks solved within one second, while the algorithm gets zero points for unsolved tasks or tasks solved in 300 seconds. For *quality* the algorithm gets a score of c^*/c for a solved task, where c is the cost of the reported plan and c^* is the cost of the best plan found with any approach. Finally, the sum of each metric is divided by the number of domains to obtain a total score between 0 and 100. Considering those metrics, control policies perform better than single heuristic approaches. Furthermore, dynamic control polices obtained by RL perform best according to all metrics. However, this analysis favors approaches which solve more instances than others. Recall that plan quality is not taken into account when learning a policy, which explains the small advantage of RL in plan quality, even though more instances have been solved by RL.

Next we compare the performance of our approach RL on the *training* set (Table 2b) with the performance of RL on the test set (Table 2a). It is possible to observe that RL performs better on the *training* set which can be attributed to overfitting and can explain why in some instances the performance of RL is worse than other approaches on the *test* set (see column RL of Table 2a and 2b). This issue of RL can be addressed in several ways, such as tuning the hyperparamters, expanding the training set or adding state features.

Finally, we want to mention the computational overhead of our RL approach compared to ALT and SINGLE HEURISTIC search approaches. While the performance of RL still exceeds the SINGLE HEURISTIC search of FAST DOWNWARD for all four heuristics, RL performs slightly worse than the internal heuristic alternation strategy of FAST DOWNWARD. In the future the overhead can be reduced by integrating the reinforcement learning part directly in FAST DOWNWARD instead of communicating via TCP/IP.

Conclusion

We theoretically and empirically evaluated the use of dynamic algorithm configuration for planning. More specifically, we have shown that dynamic algorithm configuration can be used for dynamic heuristic selection that takes into account the internal search dynamics of a planning system. Dynamic policies for heuristic selection generalize policies of existing approaches like algorithm selection and adaptive algorithm control, and their use can improve search performance exponentially. We presented an approach based on dynamic algorithm configuration and showed empirically that it is possible to learn policies capable of outperforming other approaches in terms of coverage.

For future work we will investigate domain-specific state features to learn domain-independent dynamic policies. Further, it is possible to dynamically control several parameters of a planner and to switch dynamically between different search algorithms. This raises the question how the search progress (Aine and Likhachev 2016) can be shared when using different search strategies. In particular, if we want to combine different search techniques, such as heuristic search (Bonet and Geffner 2001), symbolic search (Torralba et al. 2017; Speck, Geißer, and Mattmüller 2018) and planning as satisfiability (Kautz and Selman 1992; Rintanen 2012), it is an open question how to share the search progress.

Acknowledgments David Speck was supported by the German Research Foundation (DFG) as part of the project EPSDAC (MA 7790/1-1). André Biedenkapp, Marius Lindauer and Frank Hutter acknowledge funding by the Robert Bosch GmbH.

References

Aine, S., and Likhachev, M. 2016. Search portfolio with sharing. In *Proc. ICAPS 2016*, 11–19.

Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. of CP'09*, 142–157.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *AIJ* 175:2075–2098.

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

Biedenkapp, A.; Bozkurt, H. F.; Eimer, T.; Hutter, F.; and Lindauer, M. 2020. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In *Proc. of ECAI'20*.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2016. The IBaCoP planning system: Instance-based configured portfolios. *JAIR* 56:657–691.

Cook, B., and Huber, M. 2016. Dynamic heuristic planner selection. In *Proc. SMC 2016*, 2329–2334.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *Proc. AAAI 2010*.

Droste, S.; Jansen, T.; and Wegener, I. 2002. Optimization with randomized search heuristics - the (A)NFL theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science* 287(1):131–144.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Automated configuration of Fast Downward. In *IPC 2011 planner abstracts*, 31–37.

Fawcett, C.; Vallati, M.; Hutter, F.; Hoffmann, J.; Hoos, H.; and Leyton-Brown, K. 2014. Improved features for runtime prediction of domain-independent planners. In *Proc. ICAPS 2014*.

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

Gomoluch, P.; Alrajeh, D.; and Russo, A. 2019. Learning classical planning strategies with policy gradient. In *Proc. ICAPS 2019*, 637–645.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2020. Learning neural search policies for classical planning. In *Proc. ICAPS 2020*, 522–530.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *Proc. ICAPS 2008*, 140–147.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proc. AAAI 2008*, 944–949.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

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

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hutter, F.; Hoos, H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: An automatic algorithm configuration framework. *JAIR* 36:267–306.

Hutter, F.; Lindauer, M.; Balint, A.; Bayless, S.; Hoos, H.; and Leyton-Brown, K. 2017. The configurable SAT solver challenge (CSSC). *AIJ* 243:1–25.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. ECAI 1992*, 359–363.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs.LG]*.

López-Ibáñez, M.; Dubois-Lacoste, J.; Caceres, L. P.; Birattari, M.; and Stützle, T. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3:43–58.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Adaptive planner scheduling with graph neural networks. In *Proc. AAAI 2020*. 5077–5084.

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

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65–118.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, 273–280.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *AIJ* 193:45–86.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, 246–249.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning portfolios of automatically tuned planners. In *Proc. ICAPS 2012*, 368–372.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Proc. AAAI 2015*, 3364–3370.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proc. AAAI 2019*, 7715–7723.

Snoek, J.; Larochelle, H.; and Adams, R. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NeurIPS'12*, 2960–2968.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018. Symbolic planning with edge-valued multi-valued decision diagrams. In *Proc. ICAPS 2018*, 250–258.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proc. ICAPS 2011*, 250–257.

Tokui, S.; Okuta, R.; Akiba, T.; Niitani, Y.; Ogawa, T.; Saito, S.; Suzuki, S.; Uenishi, K.; Vogel, B.; and Yamazaki, H. V. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proc. of KDD'19*, 2002–2011.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *AIJ* 242:52–79.

van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *Proc. of AAAI'16*, 2094–2100.

Virseda, J.; Borrajo, D.; and Alcázar, V. 2013. Learning heuristic functions for cost-based planning. In *ICAPS 2013 Workshop on Planning and Learning*.

Wolpert, D. H., and Macready, W. G. 1995. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.