Learning Heuristic Selection with Dynamic Algorithm Configuration

David Speck\textsuperscript{1,*}, André Biedenkapp\textsuperscript{1,*}, Frank Hutter\textsuperscript{1,2}, Robert Mattmüller\textsuperscript{1}, Marius Lindauer\textsuperscript{3}

\textsuperscript{*}Contact Author, Equal contribution

\textsuperscript{1}University of Freiburg, Freiburg, Germany
\textsuperscript{2}Bosch Center for Artificial Intelligence, Renningen, Germany
\textsuperscript{3}Leibniz University Hannover, Hannover, Germany

\{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 useful 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.

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). 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) showed the promise of searching 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 a priori based on the characteristics of the present planning instance (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. This automated process is referred to as algorithm selection (Rice 1976) while optimization of algorithm parameters is referred to as algorithm configuration (Hutter et al. 2009). 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 a contextual Markov decision process and addressed by standard reinforcement learning methods.

In planning, there is little work that takes 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 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. 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 with multiple open lists (Röger and Helmert 2010) 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 ∼ for all notations directly related to reinforcement learning; e.g. π denotes a plan of a planning task, while ̂π 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 \( i = (\mathcal{V}, s_0, \mathcal{O}, s_*) \) 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 assignment 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 \( o = (pre_o, eff_o) \) 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 assign-
tics 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 (Röger and Helmert 2010). 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, 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 parameters 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 $\Theta$, a set of problem instances $I$ the algorithm has to solve, a state description $s_i$ of the algorithm $A$ solving an instance $i \in I$ at step $t \in \mathbb{N}_0$, and a reward signal $r$ assessing the reward (e.g., runtime or number of state expansions) of using a control policy $\pi \in \Pi$ to control $A$ on an instance $i \in I$, the goal is to find a (dynamic) control policy $\pi^* : \mathbb{N}_0 \times \tilde{S} \times I \rightarrow \tilde{\Theta}$. This policy adaptively chooses a configuration $\theta \in \tilde{\Theta}$ given a state $s_i \in \tilde{S}$ of $A$ at time $t \in \mathbb{N}_0$ to optimize the reward of $A$ across the set of instances $S$, i.e., $\pi^* \in \arg \max_{\pi \in \Pi} \mathbb{E}[r(\pi, i)]$. Note that the current time step $t \in \mathbb{N}_0$ and instance $i \in I$ can be encoded in the state description $\tilde{S}$ of an algorithm $A$, which leads to a dynamic control policy, defined as $\tilde{\pi}_{\text{dac}} : \tilde{S} \rightarrow \tilde{\Theta}$.

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 $s_i$ and the corresponding reward $r_i^t$ 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 the next internal state $s_{i+1}$ with reward $r_{i+1}^t$. 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. Helmert (2006) 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 dynamic algorithm configuration. For example, alternation is a 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. 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 — instance, time step, and state of the planner — can be important and helpful in selecting the heuristic for the next state expansion. The following summarizes existing approaches to heuristic selection within the framework of algorithm configuration.

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

- **Adaptive Algorithm Configuration**:
  - Policy: $\tilde{\pi}_{\text{aac}} : \mathbb{N}_0 \rightarrow H$
  - Example: Alternation (Röger and Helmert 2010; Seipp et al. 2015)

- **Dynamic Algorithm Configuration**:
  - Policy: $\tilde{\pi}_{\text{dac}} : \mathbb{N}_0 \times \tilde{S} \times I \rightarrow H$
  - 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:

\[
\begin{align*}
\max_h &: \text{maximum } h \text{ value for each heuristic } h \in H; \\
\min_h &: \text{minimum } h \text{ value for each heuristic } h \in H; \\
\mu_h &: \text{average } h \text{ value for each heuristic } h \in H; \\
\sigma^2_h &: \text{variance of the } h \text{ values for each heuristic } h \in H; \\
\#_h &: \text{number of entries for each heuristic } h \in H; \\
t &: \text{current time/expansion step } t \in \mathbb{N}_0.
\end{align*}
\]

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., \(\Theta = H = \{h_0, \ldots, h_{n-1}\}\).

The described set of features is a starting point and domain independent, but does not contain any specific context information yet. In general, it is possible to describe an instance or domain with 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, we limit ourselves to domain-wise dynamic control policies and show that the concept of DAC can improve heuristic search for this setting 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 function should reflect this desire.

We use a simple 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 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 this section, we investigate the theoretical properties of using DAC for heuristic search algorithms. 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 expansions 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 \(\pi_{dac}\) over algorithm selection policies \(\pi_{as}\) or adaptive algorithm configuration policies \(\pi_{aac}\). Proposition 1 proves that for each heuristic search algorithm in combination with each collection of heuristics there is a dynamic control policy \(\pi_{dac}\) which is as good as \(\pi_{as}\) or \(\pi_{aac}\) in terms of state expansions.

**Proposition 1.** Independent of the heuristic search algorithm and the collection of heuristics, for each algorithm selection policy \(\pi_{as}\) and adaptive algorithm configuration policy \(\pi_{aac}\) there is a dynamic control policy \(\pi_{dac}\) which expands at most as many states as \(\pi_{as}\) and \(\pi_{aac}\) 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 \(\pi_{dac}\) as \(\pi_{dac} = \pi_{as}\) or \(\pi_{dac} = \pi_{aac}\). □

With Proposition 1 it follows directly that an optimal algorithm configuration policy \(\pi_{dac}^{\ast}\) is at least as good as an optimal algorithm selection policy \(\pi_{as}^{\ast}\) and an optimal adaptive algorithm configuration policy \(\pi_{aac}^{\ast}\).

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

□

It is natural to ask 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 \(\pi_{as}\) and adaptive algorithm configuration policy \(\pi_{aac}\), we can construct a family of planning tasks so that a dynamic control policy \(\pi_{dac}\) will expand exponentially fewer states until a plan is found. For this purpose, we introduce a family of planning instances \(t_n\) with \(O(n)\) propositional variables and \(O(n)\) operators. The induced transition system of \(t_n\) is visualized in Figure 2. There is exactly one goal path \(s_0, s_1, s_2\), which is induced by the unique plan \(\pi = (o_1, o_2)\). 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 \( |i_n| \), 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 \( \pi_{dac} \) there exists a family of planning instances \( i_n \), a collection of heuristics \( H \) and a dynamic control policy \( \pi_{dac} \), so that greedy best-first search with \( H \) and \( \pi_{dac} \) expands exponentially more states in \(|i_n|\) than greedy best-first search with \( H \) and \( \pi_{dac} \), until a plan \( \pi \) is found.

**Proof.** Let \( \pi_{dac} \) 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, \( \pi_{dac} \) can either select heuristic \( h_0 \) or \( h_1 \). We first assume that \( \pi_{dac} \) 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 \( S \) have been expanded.

In comparison, for \( \pi_{dac} \) 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., \( \arg\min_{h \in H} h_b \). Following \( \pi_{dac} \), first \( h_0 \) and then \( h_1 \) is selected, generating the goal state \( s_2 \) in time step 1. Therefore, \( \pi_{dac} \) only expands 2 states, while \( \pi_{dac} \) expands \( 2^{n-2} \) states until a goal state is found.

Finally, for a policy \( \pi_{dac} \) 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 \( \pi_{at} \) there exists a family of planning instances \( i_n \), a collection of heuristics \( H \) and a dynamic control policy \( \pi_{dac} \), so that greedy best-first search with \( H \) and \( \pi_{at} \) expands exponentially more states in \(|i_n|\) than greedy best-first search with \( H \) and \( \pi_{dac} \), until a plan \( \pi \) is found.

For the proof of this theorem, we refer to the longer arXiv version of this paper (Speck et al. 2020).

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. Thus, 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, we want to emphasize that all results presented are theoretical and 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 obtained by a random split. Note that the policies we consider here 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). Although there are many more sophisticated search strategies and components, we choose a vanilla search strategy to reduce the factors that might affect the comparison of the actual research question of whether the learned DAC policies can improve the search performance of heuristic search. Nevertheless, our approach is in principle also capable of handling more complex search strategies and components, such as lazy eager search with preferred operators and simple handcrafted DAC like policies such as boosting heuristics (Richter, Westphal, and Helmert 2011).

We implemented an extension for Fast Downward, which makes it possible to communicate with a controller.

\(^1\)Resources: https://github.com/speckdavid/rl-plan
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.

(We also conducted additional experiments with five heuristics instead of four, including the \( h_{\text{lm-count}} \) heuristic (Richter, Helmert, and Westphal 2008); please see Speck et al. (2020).)

We use CHAINER’s v0.7.0 default parameters for ADAM.

We used earlier iterations of the oracle selector, which selects the best configuration space, i.e., \( H = \{h_{\text{ff}}, h_{\text{cg}}, h_{\text{cea}}, h_{\text{add}}\} \), which can be changed at each time step: (1) the FF heuristic \( h_{\text{ff}} \) (Hoffmann and Nebel 2001), (2) the causal graph heuristic \( h_{\text{cg}} \) (Helmert and Geffner 2008), and (3) the context-enhanced additive heuristic \( h_{\text{cea}} \) (Helmert and Geffner 2008), and (4) the additive heuristic \( h_{\text{add}} \) (Bonet and Geffner 2001). For evaluation (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 188 GB shared RAM running Ubuntu 18.04 LTS 64 bit.

We use \( \epsilon \)-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 (Kingma and Ba 2014) for 10^6 update steps on a single machine of our cluster with two CPU cores and 20 GB RAM. We use a cutoff of 7 500 control/expansion steps in order to avoid policies being executed arbitrarily long during training. Complex instances may not be solved within this cutoff, even with the optimal policy, and thus learning occurs on simpler instances. However, the underlying assumption is that well performing policies for smaller instances generalize to larger instances within a domain. 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 artificial domain (see Speck et al. (2020)) and kept these hyperparameters fixed for all 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\(^4\) 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 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

\(^4\)https://github.com/AI-Planning/pddl-generators
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.

**Coverage.** The results of Table 1 show that our approach (RL) performs best on average in terms of coverage (individual coverage of the five 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 well performing 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. We also compare our approach to the theoretically best possible algorithm selector. Considering the columns BEST AS, we observe that oracle single heuristic selection and oracle alternating selection do not perform better than the average performance of our learned RL policies, which shows that 1) heuristic search with multiple heuristics can in practice benefit from dynamic algorithm configuration and 2) it is possible to learn well performing dynamic policies domain-wise. Even under the unrealistic circumstances of an optimal algorithm selector, our learned policies perform better and therefore outperform all possible algorithm selection policies.

If we increase the configuration space by adding another heuristic ($h_{\text{ff, count}}$) the overall coverage of the control policies increases. However, the results are still qualitatively similar to those presented here, showing that the learned DAC policies perform best overall (see Speck et al. (2020)).

We also want to mention the computational overhead of our RL approach compared to ALT and SINGLE HEURISTIC search approaches. While the performance of RL on the test set still exceeds the SINGLE HEURISTIC search of FAST DOWNWARD for all four heuristics with a single open list (maintaining only the used heuristic), 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.

**Guidance, speed and quality.** 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 policies 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.

**Policy analysis.** We analyzed the resulting policies on the test set considering all successful runs, i.e., runs where a plan was found. Our first finding is that the RL policies often favor one of the heuristics over all others. In BARMAN, CHILDSNACK, ROVERS, and VISITALL, the $h_{\text{ff}}$ heuristic is preferred, while in BLOCKSWORLD the $h_{\text{add}}$ and in VISITALL the $h_{\text{sea}}$ is most often chosen. Interestingly, the percentage of use of the preferred heuristic varies greatly by domain. In SOKOBAN, e.g., $h_{\text{ff}}$ is selected for 98% exploration steps on average, while in BARMAN, the policies only select this heuristic 60% of the time on average.

---

**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). Higher scores are preferable for all metrics.**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CONTROL POLICY</th>
<th>SINGLE HEURISTIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
<td>RL RND ALT h_{ff} h_{sea} h_{add}</td>
<td>h_{ff} h_{sea} h_{add}</td>
</tr>
<tr>
<td>COVERAGE</td>
<td>84.2 81.3 81.4 70.8 64.2 64.0 68.8</td>
<td></td>
</tr>
<tr>
<td>GUIDANCE</td>
<td>38.5 37.4 37.5 30.8 27.6 28.6 30.4</td>
<td></td>
</tr>
<tr>
<td>SPEED</td>
<td>66.6 62.8 62.8 54.9 50.4 50.3 54.0</td>
<td></td>
</tr>
<tr>
<td>QUALITY</td>
<td>76.2 76.0 76.0 65.8 57.6 56.2 60.9</td>
<td></td>
</tr>
</tbody>
</table>

(a) Test set

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CONTROL POLICY</th>
<th>SINGLE HEURISTIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
<td>RL RND ALT h_{ff} h_{sea} h_{add}</td>
<td>h_{ff} h_{sea} h_{add}</td>
</tr>
<tr>
<td>COVERAGE</td>
<td>87.0 83.6 83.0 71.7 64.3 65.0 68.5</td>
<td></td>
</tr>
<tr>
<td>GUIDANCE</td>
<td>39.8 38.3 38.4 31.4 26.6 28.8 30.2</td>
<td></td>
</tr>
<tr>
<td>SPEED</td>
<td>69.3 65.3 65.4 56.0 49.1 51.1 54.2</td>
<td></td>
</tr>
<tr>
<td>QUALITY</td>
<td>79.5 77.9 77.5 66.8 57.3 58.0 61.3</td>
<td></td>
</tr>
</tbody>
</table>

(b) Training set
This raises the question of how similar the learned policies are and whether the policies are dynamic in the sense that they often switch between different heuristics. To visualize the policies, we used t-distributed stochastic neighbor embedding (t-SNE; Hinton and Roweis 2002), which maps higher-dimensional data into a 2-D space for visualization (Figure 4). As input, we used a vector of four numbers for each instance, indicating the percentage of times a heuristic was chosen, and mapped it to a space where, informally, the more similar the policy executions are, the closer the instances are to each other. Figure 4a shows the similarity of the policy executions for each instance of the different domains. It can be observed that the overall heuristic selection of the learned policies is similar domain-wise, given the resulting clustering. Figure 4b additionally visualizes how often a heuristic was switched, i.e., the frequency of switching the heuristic between subsequent expansion steps. Interestingly, in the majority of the instances, the switching frequency is high (sequences with no $h$ switch < 100 steps), while in the instances corresponding, for example, to the domain SOKOBAN, the switching frequency is low (sequences with no $h$ switch > 1000 steps). This correlates with the fact that in this domain a single heuristic was selected with a high percentage.

We conclude that our approach is able to learn highly dynamic policies, but also highly static policies, depending on the problem instance at hand. Intuitively, this makes sense, since there may be domains for which a static policy performs best. However, DAC is a generalization of algorithm selection which allows to learn such static policies as well.

**Training performance.** We compare the performance of our RL approach on the training set (Table 2b) with the performance of RL on the test set (Table 2a). Interestingly, RL performs better on the training set which can be attributed to a certain degree of overfitting and can explain why in some instances the performance of RL is worse than other approaches on the test set. This issue can be addressed by tuning the hyperparameters, expanding the training set or adding further state features. Overall, the improvements DAC yields over the other methods more than outweigh any overfitting, leading to DAC performing best on the test set.

**Conclusion**

We investigated 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 they 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.

In 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**

D. Speck was supported by the German Research Foundation (DFG) as part of the project EPSDAC (MA 7790/1-1). M. Lindauer acknowledges support by the DFG under LI 2801/4-1. A. Biedenkapp, M. Lindauer and F. Hutter acknowledge funding by the Robert Bosch GmbH.

**References**


