Learning Domain-Independent Policies for Open List Selection

André Biedenkapp,[1] **David Speck,**[1,2] **Silvan Sievers,**[3]
Frank Hutter,[1,4] **Marius Lindauer,**[5] **Jendrik Seipp**[2]
[1]University of Freiburg, [2]Linköping University, [3]University of Basel,
[4]Bosch Center for Artificial Intelligence, [5]Leibniz University Hannover
biedenka@cs.uni-freiburg.de

Abstract

Since its proposal over a decade ago, LAMA has been considered one of the best-performing satisficing classical planners. Its key component is heuristic search with multiple open lists, each using a different heuristic function to order states. Even with a very simple, ad-hoc policy for open list selection, LAMA achieves state-of-the-art results. In this paper, we propose to use dynamic algorithm configuration to learn such policies in a principled and data-driven manner. On the learning side, we show how to train a reinforcement learning agent over several heterogeneous environments, aiming at zero-shot generalization to new related domains. On the planning side, our experimental results show that the trained policies often reach the performance of LAMA, and sometimes even perform better. Furthermore, our analysis of different policies shows that prioritizing states reached via preferred operators is crucial, explaining the strong performance of LAMA.

Introduction

Domain-independent classical planning is the problem of finding a sequence of (deterministic) actions that lead from a given initial situation of the world to a state satisfying the given goal specification. LAMA (Richter and Westphal 2010) is one of the most successful planning systems for *satisficing* planning, where finding a solution quickly is prioritized over finding a cost-optimal solution. LAMA uses (i) *greedy best-first search* (Pearl 1984) with *deferred heuristic evaluation*, also called lazy greedy search (Helmert 2006; Richter and Helmert 2009), and with (ii) *multiple heuristics* in an alternating scheme (Röger and Helmert 2010). Additionally, it prioritizes expanding states with (iii) *preferred operators* of the heuristics using *boosting* (Richter and Helmert 2009). To do so, LAMA follows a simple ad-hoc policy for selecting an open list for expansion: alternate between all open lists, unless the expansion of a state reached via a preferred operator improved the heuristic value, in which case alternating is restricted to preferred operator open lists for the next 1000 expansions. While this policy for open list selection works well for many planning instances from the International Planning Competitions (IPCs), its strengths and weaknesses are not well understood.

In this paper, we shed light on why LAMA's policy works well and whether it can be learned in a principled and data-driven way using machine learning techniques. Previous application of machine learning techniques in planning mainly used *meta-algorithmic* approaches, such as algorithm selection (Rice 1976; Xu et al. 2008) and algorithm configuration (Hutter et al. 2009; Kadioglu et al. 2010), for improving planners (e.g., Gerevini, Saetti, and Vallati 2009; Fawcett et al. 2011; Seipp et al. 2012; Fawcett et al. 2014; Seipp et al. 2015; Cenamor, de la Rosa, and Fernández 2016; Sievers et al. 2019; Ma et al. 2020). These approaches, however, do not allow interleaving learning and algorithm execution. Gomoluch et al. (2020) instead used the cross-entropy method to train search policies which *dynamically* decide which type of search algorithm to use in a given state. Biedenkapp et al. (2020) introduced a general meta-algorithmic framework for interleaving learning and execution called *dynamic algorithm configuration (DAC)*. Most recently, Speck et al. (2021) used DAC to train *domain-dependent* policies for selecting which heuristic to use when, deciding which state to expand next in an eager greedy best-first search.

In this work, we generalize the DAC framework to learn *domain-independent* policies for open list selection in lazy greedy search with preferred operators, thus allowing our policies to generalize to new unseen domains. To this end, we extend the framework to incorporate task-specific features which allow training a reinforcement learning agent over several heterogeneous environments (i.e., planning tasks from different domains), aiming at zero-shot generalization to related domains. In an experimental study using six IPC domains, we show that our learned policies reduce the required number of node expansions compared to baseline policies and LAMA's boosting policy in several domains and that they often approach the performance of domain-dependent policies. We analyze the learned policies in detail and show that prioritizing states reached via preferred operators (in particular those of the FF heuristic; Hoffmann and Nebel, 2001) is crucial, thus explaining the strong performance of LAMA's policy for combining multiple open lists.

Background

We begin by providing the necessary concepts and background on both DAC and classical planning.

Dynamic Algorithm Configuration

Dynamic algorithm configuration (DAC; Biedenkapp et al. 2020) is a meta-algorithmic approach which uses information about the internal behaviour of an algorithm and information about the instance it is run on to change the configuration of the algorithm *during* its execution. Both types of information can be expressed through so-called features. As the information about the internal behaviour of an algorithm typically changes at every iteration, we refer to such features as *dynamic features*. For example, the minimal heuristic value changes frequently throughout the execution of a planner and it can give information about the progress of the planner. In contrast, information about the instance at hand, such as the instance size, typically remains unchanged throughout execution of an algorithm. We refer to features encoding this type of information as *instance features*.

Let $\mathcal{A}$ be an algorithm and $\mathcal{I}$ be a set of instances on which $\mathcal{A}$ should be configured using DAC. The *DAC state* $\tilde{s}_t^i$ of $\mathcal{A}$ at step $t \in \mathbb{N}_0$ when solving instance $i \in \mathcal{I}$ is described by numerical values consisting of both types of features. Let $\tilde{\mathcal{S}}$ be the set of DAC states and $\tilde{\Theta}$ be the *configuration space* of $\mathcal{A}$. A *DAC policy* $\tilde{\pi}\colon \tilde{\mathcal{S}} \to \tilde{\Theta}$ maps each DAC state $\tilde{s}_t^i \in \tilde{\mathcal{S}}$ to a configuration $\tilde{\theta} \in \tilde{\Theta}$. Let $\tilde{r}$ be a reward function which maps a DAC policy $\tilde{\pi}$ and an instance $i \in \mathcal{I}$ to the performance of $\mathcal{A}$ using $\tilde{\pi}$ on i. The goal of DAC is to find a DAC policy $\tilde{\pi}^*$ which optimizes the reward on $\mathcal{I}$: $\tilde{\pi}^* \in \arg\max_{\tilde{\pi}\in\tilde{\Pi}} \mathbb{E}_{i\sim\mathcal{I}}\left[\tilde{r}(\tilde{\pi}, i)\right]$.

Classical Planning and Greedy Search

A *planning task* (also called planning *instance*) provides a description of the world via an *initial state*, a *goal specification* and a set of *actions* (also called operators) describing the dynamics of the world. Solving a planning task implies finding a sequence of actions, called *plan* or *solution*, that lead from the initial state to a state satisfying the goal specification, called a *goal state*. We assume that all actions cost 1. The cost of a plan is therefore equal to its length. We consider *satisficing* planning, where the objective is to find a short plan, but not necessarily a shortest plan.

A commonly used solution technique for satisficing planning is heuristic search, in particular variants of *greedy best-first search* (Pearl 1984). We consider greedy best-first search with multiple heuristics and deferred heuristic evaluation (Richter and Helmert 2009; Röger and Helmert 2010), which we refer to as *greedy search* in the following. Greedy search maintains a finite set of priority queues, called *open lists*, $\mathcal{O}_n = \{O_1, \ldots, O_n\}$. Each open list O_i is associated with a *heuristic* h_i which estimates the cheapest cost of reaching a goal state from a given state. The *expansion* of a state s results in a set of successor states which are added to all open lists together with the heuristic estimate of their *parent* state s. Only when a state is removed from open list O_i, it is evaluated with heuristic h_i. This *deferred heuristic evaluation* attempts to reduce the number of heuristic evaluations, which can be beneficial if the branching factor is large (Helmert 2006). Greedy search starts by expanding the initial state and then repeatedly selects an open list O_i for expanding the next state until finding a goal state.

Clearly, there are two design choices here: which heuristics (respectively open lists) to use and which one to select in each search iteration. Resolving the first choice, we consider the two heuristics which form the backbone of the state-of-the-art satisficing planner LAMA (Richter and Westphal 2010): the FF heuristic h_{ff} (Hoffmann and Nebel 2001) and the landmark count heuristic h_{lmc} (Richter, Helmert, and Westphal 2008). Both heuristics can determine *preferred operators* (Hoffmann and Nebel 2001; Richter and Helmert 2009) as a side-product of their computation. Preferred operators are operators deemed to make progress towards a goal state. For each heuristic, greedy search using preferred operators maintains an additional preferred operator open list which contains the subset of states reached via preferred operators of the heuristic. Thus, LAMA uses four open lists $\mathcal{O}_4$, associated with the heuristics $\mathcal{H} = \{h_{\mathrm{ff}}, h_{\mathrm{ff}}^{\mathrm{p}}, h_{\mathrm{lmc}}, h_{\mathrm{lmc}}^{\mathrm{p}}\}$.

It remains to resolve the second choice, i.e., determining a *policy for open list selection* which defines the next state to expand in each iteration of greedy search. Röger and Helmert (2010) introduced the idea of switching between open lists in a round-robin manner (`rr`) to use each heuristic function equally often. LAMA improved upon `rr` by deviating from it whenever evaluating a state reached via a preferred operator leads to an improved heuristic value: in this case, all preferred operator open lists are *boosted*, i.e., they receive priority over the others for the next 1000 expansions. Priorities are accumulated if during the priority phase, another preferred state leads to heuristic improvement. Recently, Speck et al. (2021) categorized different types of policies for heuristic selection and showed that using DAC to learn policies per domain generalizes these policies. Since they only investigated heuristic selection rather than open list selection, their design space did not include preferred operator queues and thus cannot be used to learn LAMA-like policies.

Learning Domain-Independent Policies

In this work, we use DAC to train *domain-independent* policies for open list selection of a planner using lazy greedy search with multiple heuristics and preferred operators. Prior approaches to learning DAC policies are typically based on deep reinforcement learning (RL). For example, Speck et al. (2021) use deep RL to train domain-dependent policies for heuristic selection of a planner using lazy greedy search. The common assumption in deep RL is that an RL agent only interacts with a single environment. However, in our case, this assumption does not hold: to obtain domain independence, we need to train across many planning instances from different domains, i.e., related but heterogeneous environments. Even though interest in RL agents that are capable of generalizing and solving multiple related environments is increasing (see, e.g., a survey by Kirk et al. 2021), to the best of our knowledge there exist only few approaches which tackle the problem of consolidating instance features with dynamic features. For example, Cobbe et al. (2020) encode instance-specific information directly in an image state (such as a different background for a new level). Zhang et al. (2021) use the instance ID and Eimer et al. (2021a) use a flattened rep-

resentation of a maze which they simply concatenate with dynamic DAC state features. The few features considered in prior works provide perfect information, i.e., they allow to perfectly distinguish between instances. We are the first to tackle the task of learning general RL policies in settings where perfect instance features are not readily available and study how to incorporate many imperfect features.

We follow the idea of combining instance with dynamic features. We extend the approach by Speck et al. (2021) by incorporating instance features and show how to use deep RL to learn domain-independent policies. In this section, we describe how we instantiate the DAC framework described in the previous section: we define the configuration space, the reward function, dynamic features as well as instance features which make up the DAC state. Finally, we describe the planner and the instances we use for training.

Configuration Space. We use the four open lists of LAMA as configuration space: $\tilde{\Theta} = \mathcal{O}_4 = \{O_1, \dots, O_4\}$ associated with the heuristics $\mathcal{H} = \{h_{\text{ff}}, h^{\text{p}}_{\text{ff}}, h_{\text{lmc}}, h^{\text{p}}_{\text{lmc}}\}$. Since planning states are always pushed into all regular (and possibly preferred) open lists, an open list can contain a closed state s when s was already popped from another queue and expanded. Thus, a DAC agent might have to repeatedly communicate with the planning algorithm until it finally finds a valid planning state to expand. This could lead to large communication overheads without any actual progress in the search. To mitigate this issue we pop planning states from the selected open list until finding a non-closed state.

Reward Function. The reward function is a crucial part of the learning pipeline as it defines which policies are preferable over others. In our scenario, we aim at learning policies that minimize the number of node expansions when finding a plan. We define the reward function $\tilde{r}$ of a policy $\tilde{\pi}$ on instance i as the sum of rewards $\tilde{r}^i_t(\tilde{s}^i_t)$ obtained in each DAC state $\tilde{s}^i_t$ during execution of $\tilde{\pi}$ on i, i.e., $\tilde{r} = \sum_t \tilde{r}^i_t(\tilde{s}^i_t)$. Since the DAC agent maximizes the reward and we want to minimize the number of steps (i.e., expansions), we define $\tilde{r}^i_t(\tilde{s}^i_t) = -1$ for all i and all t; this rewards reaching a goal state faster.

Dynamic Features. State features define the DAC state of the planner. We use the same dynamic state features as Speck et al. (2021):

$\max_h$**:** maximum h value for each open list;

$\min_h$**:** minimum h value for each open list;

μ_h**:** average h value for each open list;

σ^2_h**:** variance of the h values for each open list;

$\#_h$**:** number of entries for each open list

Instead of using the raw feature values, we compute the difference between the last and current time step ($t-1$ and t) for each feature. This allows us to encode the progress that has been made by expanding a node from one of the open lists.

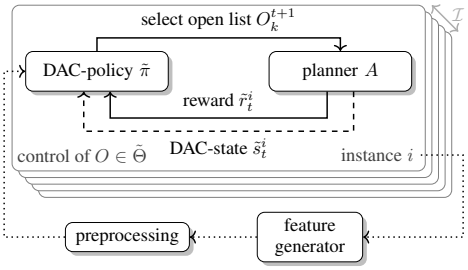


Figure 1: Schematic representation of the interaction of a DAC policy $\tilde{\pi}$ with an algorithm $\mathcal{A}$ when adapting its parameter O at every time step t while solving instance i. We explicitly include instance features to enable learning domain-independent policies.

Instance Features. We also include instance features, as information about the task or the domain at hand can help to distinguish between domains and could prove useful in learning domain-independent DAC policies. To this end we use the handcrafted planning features collected by Fawcett et al. (2014). These features include information about the PDDL description of the task (e.g., the number of action schemas), information about the translation to a SAS$^+$ planning task (Bäckström and Nebel 1995) (e.g., the number of state variables), and information from short probing runs of the Fast Downward Planning System (Helmert 2006) (e.g., the number of evaluated states). Note that instance features remain constant throughout the run of the planning system, thus we do not compute the difference between time steps for these features. Since it is an open question in RL how to best combine instance and dynamic features, we discuss multiple ways to do so in the next section.

Overall Learning Pipeline To conclude this section, Figure 1 depicts the interaction of a learned policy for open list selection $\tilde{\pi}$ with a planner $\mathcal{A}$. Before running $\mathcal{A}$ on instance i, we generate instance features for instance i, possibly preprocessing them further before passing them to the DAC policy. During execution, $\mathcal{A}$ follows $\tilde{\pi}$ by selecting the k-th open list O^t_k for expansion at time step t. After expansion, it informs the DAC policy about the resulting DAC state $\tilde{s}^i_t$ and the incurred reward $\tilde{r}^i_t(\tilde{s}^i_t) = -1$. This feedback loop allows the agent to learn which action in which state results in the largest reward and to adapt the policy accordingly.

Combining Instance and Dynamic Features

With little prior work on combining instance and dynamic features in DAC (which we try to solve with deep RL) we need to carefully consider how to incorporate them to be able to learn domain-independent open list selection policies. This is especially important as the planning community has focused mostly on adapting parameters once for a given instance and not repeatedly while solving it. As a result, in the planning literature knows many instance features and only a few dynamic features. In our work, we consider 305 instance features (Fawcett et al. 2014) and 5 dynamic

features per open list (Speck et al. 2021). Working with such a large set of instance features poses a challenge that has not been tackled before in DAC or similar deep RL settings. We discuss four different methods for combining instance and dynamic features for learning DAC policies in our setting.

Using Only Dynamic Features. The simplest approach to learning DAC policies across domains is to only use the dynamic features while training across a set of different planning domains. Using only dynamic features is already sufficient to learn DAC policies for individual domains that can outperform the theoretical best algorithm selector for this domain (Speck et al. 2021). This intra-domain generalization gives evidence that dynamic features allow us to distinguish between instances from the same domain. Still, it is not clear whether using dynamic features to learn policies is sufficient to enable generalization across multiple domains.

Concatenating Instance and Dynamic Features. When using instance features while training across instances, it is straightforward to concatenate instance and dynamic features. However, this way of combining both types of features abstracts away the type of feature an RL agent is presented with. For different types of learning processes this could pose a challenge, because by design deep RL agents expect that every feature is dynamic and changes frequently between DAC states. The concatenation of both types of features provides more information to the learner than the exclusive use of dynamic features, but may lead to overfitting based on the overproportionally large number of instance features part of the DAC state.

Learning Separate Representations. To mitigate the shortcomings of the previous two approaches, we propose to use neural network architectures that have separate input layers for the two sources of features but combine them further on in the second to last layer. The underlying idea of separating the inputs of the dynamic and instance features is to reduce overfitting to only one feature type and to allow learning representations based only on the dynamic or instance features. Further, this separation allows us to embed the high-dimensional inputs in a lower-dimensional space. These embeddings can then be concatenated further down in the architecture to learn the policy based on these representations. By learning the embeddings and policies at the same time, it is possible to balance these components better.

Decoupling Instance and Dynamic Features. While instance features can be helpful for distinguishing between domains, by design they are not helpful to understand the internal dynamics of the algorithm. Thus, it is reasonable to decouple learning either type of feature. In classical meta-algorithmic approaches, such as algorithm selection, principle component analysis (PCA) is commonly used (see, e.g., Lindauer et al. 2015). With the representation learning capability of deep neural networks we can learn embeddings that are useful for specific tasks, rather than using a hand-designed approach such as PCA. To this end, we propose to treat learning the embedding of instance features as a separate task. The instance features can be used to train a separate small neural network to classify the domain or predict which heuristic is most likely to perform well. Thus, the output of such a network contains a lot of information about the instance at hand and can be used together with dynamic information. This decoupling approach has the benefit of simplifying the learning approach. This is because the instance feature embeddings are learned in a simple supervised learning manner based only on static information, whereas the dynamic policy is trained with dynamic information and uses the instance embedding without having to learn it online.

Experimental Design

Our experiments are divided into several parts. We begin by examining scenarios with a single heuristic $h \in \{h_{\text{ff}}, h_{\text{lmc}}\}$ and two open lists, one associated with h and one with h^{P}. This allows us to assess if our approach based on DAC is capable of learning meaningful boosting policies without having to additionally learn which heuristic is more capable of guiding the search at each step. We further use these scenarios to study the effect of learning domain-independent compared to domain-dependent DAC policies. In a final experiment, we use the full configuration space which also allows us to cover LAMA's policy for open list selection. We analyze the learned policies and shed some light onto the strengths of LAMA's policy.

Training Setup

We use the Fast Downward planning system (Helmert 2006) which contains an implementation of LAMA and thus all components we need. We implement the DAC framework for learning policies in Python and use the socket communication protocol by Speck et al. (2021) for communicating between the planning system (implemented in C++) and the DAC policy. For training DAC policies, we use DACBench (Eimer et al. 2021b) which provides a simple DAC interface.

To avoid following particularly bad policies that might require many node expansions during training we impose a cutoff on the number of node expansions. If the cutoff value is too small, policies might overfit to easier instances, whereas a too large cutoff value might result in very slow training. As a compromise, we use a cutoff of 50 000 node expansions during training. In all experiments, we use the described set of dynamic features (consisting of min, max, average, variance, number of entries) for each open list. In the domain-independent case, we further consider the set of 305 instance features (Fawcett et al. 2014) that allow us to distinguish between the individual instances and domains. Finally, we evaluate the four proposed methods for combining dynamic and instance features in our experiments.

Training Pipelines

Following Biedenkapp et al. (2020), we use ϵ-greedy deep Q-learning. Similarly, we linearly decay ϵ over 5×10^5 steps from 1.0 to 0.1. All networks are trained using ADAM (Kingma and Ba 2015) with the default hyperparameters of

CHAINER v7.7.0 (Tokui et al. 2019) for 2×10^6 training steps. We repeat all training runs five times and report the average performance.

When learning DAC policies purely form dynamic features, we use the same double DQN (van Hasselt, Guez, and Silver 2016) architecture as Speck et al. (2021). This architecture consists of 2 hidden layers with 75 hidden units each. In the domain-independent setting we dub this method `no-F`. When using instance features by concatenating them to the dynamic feature vector, we keep the same architecture. Only the input layer is widened to use both dynamic and instance features. We dub this method `raw-F`.

When learning an embedding online, we adapt the base architecture to allow for two separate input streams, before merging them further down in the architecture. We performed a small search over a set of architectures to determine a suitable architecture. This search considered three different architectures that differed only slightly in the number of hidden units and layers for the input stream of the instance features. Our resulting architecture first processes the instance features over two hidden layers. The dynamic features are processed in a single hidden layer. The outputs of both streams are then concatenated and processed in a final hidden layer. All hidden layers have 75 units each. In the following, we refer to this method as `embed`.

To decouple learning of instance and dynamic features, we first train a 2-layer neural network with $[100, 50]$ hidden units to classify which of two heuristics ($h_{\text{ff}}, h_{\text{lmc}}$) results in a lower number of node expansions on the training instances. We use the soft-max output of the final layer of this network and concatenate it with the dynamic feature input of the DAC policy net, which again uses the same architecture as when only considering dynamic features with a slightly wider input layer. Thus, we get a cheap-to-train, highly-informative but low-dimensional embedding. Note that the classifier network is not further trained while learning the dynamic policy. Thus, counter to the previous method, the DAC policy is trained with a fixed embedding for the instance features. We dub this method `dc` (domain classifier).

Evaluation Protocol

In our evaluation protocol we consider training both in a domain-dependent as well as a domain-independent fashion. For the former, we train policies on a set of instances from only a single domain and in the latter we train policies across instances from multiple domains. To keep CPU computation manageable, we consider instances from six diverse IPC domains (BARMAN, BLOCKSWORLD, CHILDSNACK, DRIVERLOG, FLOORTILE, and VISITALL) generated by Autoscale (Torralba, Seipp, and Sievers 2021). The considered set of instances is split into disjoint training, validation and test sets. We use the training sets exclusively to update the weights of the policy network, while we use the validation sets to estimate generalization to unseen instances from the same domains. The test is used only to determine the final performance. When evaluating test performance, we report the performance of the best policy found on the validation set. This choice is motivated by the fact that we are interested in the quality of found policies. We also consider a leave-one-domain-out setting where we train on five domains and report the performance on the unseen sixth domain. For the final evaluation, we remove the cutoff used during training and use a runtime limit of 5 minutes a 4 GiB memory limit. We use Lab (Seipp et al. 2017) for running all experiments 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 20.04 LTS 64 bit. As our goal is to minimize the number of node expansions, we report the expansion score (Richter and Helmert 2009)

$$\text{score}(ne) = \frac{\ln(\min(\max(ne, 10^2), 10^6)) - \ln(10^6)}{\ln(10^2) - \ln(10^6)}$$

where ne is the number of node expansions required to solve a particular instance by a given policy. Intuitively, unsolved instances or solved instances with more than 10^6 node expansions give zero points, while instances solved with less than or equal to 10^2 node expansion give one point. Solved instances with node expansions between these extremes give a score that is logarithmically interpolated between 0 and 1. We sum up the scores for instances from the same domain to get per-domain scores.

Baselines

We consider four *static* baselines that perform a greedy search with a single open list associated with one of the heuristics in $\mathcal{H} = \{h_{\text{ff}}, h^{\text{p}}_{\text{ff}}, h_{\text{lmc}}, h^{\text{p}}_{\text{lmc}}\}$. We consider two baselines that follow simple *dynamic policies* for open list selection: `rand` randomly selects one of the open lists from $\mathcal{O}$ and `rr` switches between the open lists of $\mathcal{O}$ in a round-robin manner. Finally, we consider the *boosting policy* (`boost`) of LAMA as a baseline that boosts the open lists of preferred operators. We use this baseline when we consider only one heuristic with and without preferred operators, which yields two open lists, and when we consider all four open lists, which then corresponds to the LAMA planner. Note that LAMA is considered one of the most successful satisficing planners, making it a very strong baseline.

Experiments and Analyses

We report results of our experiments in three scenarios: using a single heuristic, using both heuristics, and using the leave-one-domain-out setting. Finally, we analyze LAMA's policy in comparison to those trained with our approaches.

Single Heuristic Scenarios

In the simplest scenario, we consider learning policies for open list selection between two open lists associated with one heuristic. Domain-dependent policies are trained per domain, and domain-independent ones are trained across all six domains. We report only the training results as the validation and test results closely resemble those of the training set. In the *LMC scenario*, the policies learned by the DAC agents select between the two open lists associated with the landmark count heuristic without and with preferred operators. Table 1 shows the expansion scores. In this scenario, mostly static policies perform strongest, as the open list using preferred operators should rarely be boosted. In fact, in

	Static		Dynamic			Domain-		Domain-Independent DAC							
Domain	h_{lmc}	$h^{\text{p}}_{\text{lmc}}$	rand	rr	boost	Dependent DAC		no-F		raw-F		embed		dc	
BARMAN (18)	2.70	0.00	3.21	3.59	3.59	3.60	±0.05	3.60	±0.05	2.70	±0.00	2.77	±0.00	**3.62**	±0.06
BLOCKSWORLD (20)	10.64	0.00	6.79	6.80	6.82	**11.02**	±0.69	9.63	±0.23	10.63	±0.04	10.64	±0.00	7.14	±0.25
CHILDSNACK (13)	2.80	0.00	**3.00**	2.98	2.77	**3.00**	±0.22	2.85	±0.17	2.81	±0.04	2.81	±0.03	2.79	±0.17
DRIVERLOG (21)	16.27	0.00	17.30	17.48	10.62	**17.77**	±0.34	16.29	±1.25	16.18	±0.21	16.27	±0.00	14.62	±2.40
FLOORTILE (3)	0.87	0.00	**0.97**	0.92	**0.97**	0.93	±0.03	0.96	±0.03	0.87	±0.00	0.87	±0.00	0.96	±0.03
VISITALL (30)	24.66	0.00	23.35	23.15	14.03	**24.91**	±0.39	23.25	±1.94	24.02	±1.48	24.67	±0.00	22.24	±2.53

Table 1: Training results for the LMC scenario. For all DAC methods we report the mean and standard deviation.

	Static		Dynamic			Domain-		Domain-Independent DAC							
Domain	h_{ff}	h^{p}_{ff}	rand	rr	boost	Dependent DAC		no-F		raw-F		embed		dc	
BARMAN (18)	4.97	**14.58**	6.32	6.32	14.31	12.20	±2.85	14.04	±0.57	**14.58**	±0.00	14.57	±0.03	14.32	±0.42
BLOCKSWORLD (20)	6.08	**16.99**	7.67	7.55	16.95	16.97	±0.04	16.55	±0.82	**16.99**	±0.00	16.95	±0.09	16.89	±0.20
CHILDSNACK (13)	4.40	10.66	5.50	5.45	10.42	**10.67**	±0.13	10.22	±1.07	10.57	±0.20	10.36	±0.74	9.95	±1.23
DRIVERLOG (21)	16.00	12.28	17.53	17.64	16.03	**18.21**	±0.66	17.78	±0.97	13.17	±0.48	14.04	±0.01	17.00	±2.00
FLOORTILE (3)	1.46	**1.85**	1.60	1.60	1.77	1.82	±0.07	1.68	±0.19	1.75	±0.14	1.80	±0.11	1.73	±0.14
VISITALL (30)	15.15	**15.94**	15.28	15.32	15.93	15.81	±0.20	15.85	±0.15	15.92	±0.04	**15.94**	±0.00	15.81	±0.28

Table 2: Training results for the FF scenario. For all DAC methods we report the mean and standard deviation.

this scenario, the classical boosting approach with a boost value of 1000 performs much worse than only choosing the open list not using preferred operators. In domains such as BARMAN, CHILDSNACK and FLOORTILE, roughly equal usage of both open lists results in well performing policies, slightly outperforming the static selection of only h_{lmc}.

Learning policies in a domain-dependent fashion, i.e., tailored to only the domain at hand, our DAC approach is able to find better policies in most domains. Note that in the domain-dependent case we train policies for each domain and report the score in the same domain without evaluating transfer here. Thus, the domain-dependent results in essence show the results of an oracle selector over DAC policies, to give an indication if larger improvements could be gained over the baselines. In contrast, learning domain-independent policies has the advantage that we can find well-performing policies that can adapt to the domain at hand and thus potentially result in good performance in all domains.

All DAC policies outperform the boosting baseline in nearly all domains and get close to, or outperform the best baseline per domain. Surprisingly, using no features at all already performs well in this scenario. This can be attributed to the similarity of the well-performing policies, with emphasis on the open list without preferred operators and only sporadic usage of the preferred-operator open list.

In the FF scenario, we consider the FF heuristic rather than the landmark count heuristic. Table 2 shows results. This scenario is nearly the perfect opposite to the LMC scenario as it is nearly always better to use the preferred operator open list. In contrast to the LMC scenario though, only in the DRIVERLOG domain does frequent switching (as done by `rand` and `rr`) between the open lists work well. Interestingly, in this scenario, always choosing h^{p}_{ff} works so well that even in the domain-dependent case we rarely find better policies. This confirms the common assumption that preferred operators are crucial for the performance of the FF heuristic. Additionally, these results show that considering *only* states reached via operators that FF considers preferred is enough for some domains. This is quite surprising since pruning non-preferred successors renders the search incomplete in general. When the domain-independent DAC approaches result in equal performance to the baseline, DAC has learned policies that almost always choose h^{p}_{ff}.

Multi-Heuristic Scenario

We extend the previous scenarios to contain both h_{ff} and h_{lmc} along with their preferred operator open lists, resulting in four open lists. Thus we cover and can potentially learn LAMA's hand-crafted policy for open list selection. This scenario is much more varied than the first two as individual static policies are not dominant (see Tables 3 and 4). In BARMAN and CHILDSNACK, it is sufficient to always select h^{p}_{ff}, but LAMA's policy comes close to the same performance. Similarly in VISITALL, only selecting h_{lmc} is a strong baseline but LAMA's policy comes close. By design, LAMA's policy makes use of both heuristics and all four open lists. Thus, in cases where static selection already performs well LAMA does not fall far behind as its boosting mechanism will still select the well performing open list predominantly.

On the training set (Table 3), domain-independent DAC can find better policies on BLOCKSWORLD and FLOORTILE, whereas domain-dependent DAC finds better policies on DRIVERLOG and VISITALL. As shown by the generalization to the test set (Table 4), DAC policies do not generalize well in the DRIVERLOG domain, likely due to individual instances in DRIVERLOG requiring various different policies to be solved best. Furthermore, training domain-independent policies tends to perform better when adding instance specific information, confirming our previously discussed approach. Only on the DRIVERLOG domain learning policies without using instance specific information is better. While concatenating instance features directly with dynamic features (`raw-F`) works well enough on the training set, it has the worst generalization to the test set out of the three domain-independent methods using instance features (`raw-F`, `dc` & `embed`). Using a small classifier network (`dc`) has a similar performance to learning an embedding during training (`embed`) both on the training ant test sets.

	Static				Dynamic			Domain-		Domain-Independent DAC							
Domain	h_{ff}	h^{p}_{ff}	h_{lmc}	$h^{\text{p}}_{\text{lmc}}$	rand	rr	LAMA	Dependent DAC		no-F		raw-F		embed		dc	
BARMAN (18)	4.97	**14.58**	2.70	0.00	6.30	6.27	14.42	14.31	±0.05	12.57	±2.47	12.51	±4.44	14.00	±1.39	13.39	±1.68
BLOCKSWORLD (20)	6.08	16.99	10.64	0.00	13.14	12.91	16.96	16.49	±0.04	15.82	±1.85	16.99	±0.77	**17.04**	±0.78	16.33	±1.68
CHILDSNACK (13)	4.40	**10.66**	2.80	0.00	7.28	7.33	10.37	10.50	±0.06	9.23	±1.87	9.67	±1.54	9.57	±1.99	10.02	±1.45
DRIVERLOG (21)	16.00	12.28	16.27	0.00	18.15	18.27	16.10	**18.53**	±0.03	18.34	±0.75	14.80	±0.88	15.98	±1.16	17.25	±1.90
FLOORTILE (3)	1.46	1.85	0.87	0.00	1.74	1.63	1.78	1.83	±0.01	1.66	±0.15	**1.88**	±0.11	1.78	±0.18	1.82	±0.23
VISITALL (30)	15.15	15.94	24.66	0.00	24.32	24.20	24.81	**25.38**	±0.02	23.55	±3.24	20.37	±5.14	23.61	±3.42	24.56	±1.42

Table 3: Training results for the LAMA scenario. For all DAC methods we report the mean and standard deviation.

	Static				Dynamic			Domain-	Domain-Independent DAC			
Domain	h_{ff}	h^{p}_{ff}	h_{lmc}	$h^{\text{p}}_{\text{lmc}}$	rand	rr	LAMA	Dependent DAC	no-F	raw-F	embed	dc
BARMAN (18)	1.93	9.98	0.85	0.00	4.08	3.95	**10.20**	9.98	9.26	8.35	8.39	9.54
BLOCKSWORLD (18)	3.36	10.70	4.47	0.00	6.20	6.42	**10.90**	9.17	10.33	10.08	9.81	10.08
CHILDSNACK (18)	1.66	**9.21**	0.74	0.00	3.79	3.82	9.08	7.41	8.19	8.56	2.66	3.31
DRIVERLOG (18)	7.41	5.62	7.41	0.00	**10.56**	**10.56**	8.01	9.43	6.59	6.42	8.81	5.63
FLOORTILE (18)	0.85	0.86	0.00	0.00	1.01	0.55	0.57	**1.39**	0.91	**1.39**	0.74	0.44
VISITALL (18)	6.50	6.94	12.24	0.00	11.63	11.64	11.86	**12.37**	12.20	12.02	12.31	11.88

Table 4: Test results for the LAMA scenario. For all DAC methods we report the score of the best seed on the validation set.

However, by separately learning the embedding with a classifier, learning the policy can be done with a simpler and thus more efficient architecture.

Generalizing to Unseen Domains

A question that arises is how well do the policies learned with the presented methods generalize to other domains. To this end, in the domain-dependent case we transfer the policies trained on a single domain to all other domains. We report the score normalized with respect to the performance of the domain-dependent policies in a confusion matrix (see Figure 2). This allows us to see if we can recover the same performance with a policy trained on a different domain as when training and evaluating in the same domain. Overall, policies that yield more robust performance are considered better than policies that yield high scores in single domains they were trained on but are not transferable to other domains that were not seen during training.

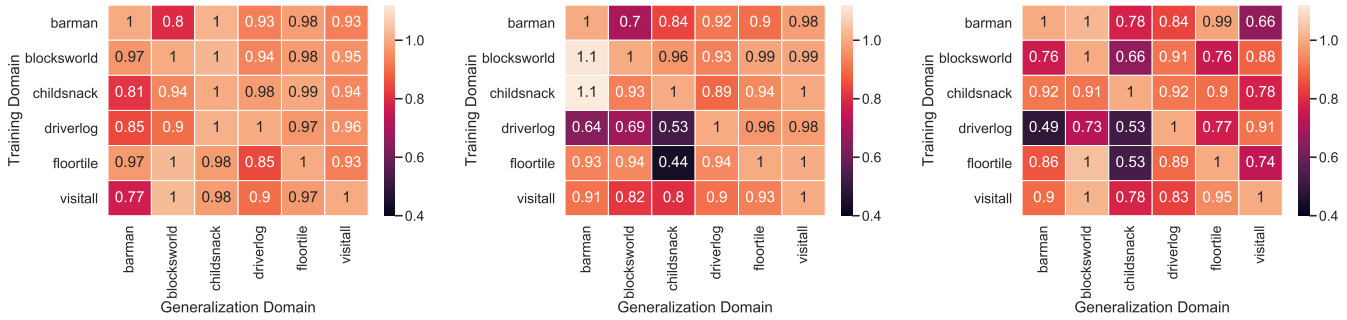
For the LMC scenario (see Figure 2a), we see that generalization to the BARMAN domain is most difficult. In general, however, the found policies in this scenario are so similar (as all successful policies need to mostly ignore $h^{\text{p}}_{\text{lmc}}$) that they are easily transferable to other domains. Policies are far less transferable in the more complex FF scenario (see Figure 2b), as not all domains require mostly static behaviour. While policies that are not trained on VISITALL result in nearly the same performance as when training on VISITALL, policies are far less transferable to BLOCKSWORLD or CHILDSNACK. Interestingly, policies trained on the DRIVERLOG domain are least transferable, resulting in strong performance drops on BARMAN, BLOCKSWORLD and CHILDSNACK. With the increased complexity of the LAMA scenario (see Figure 2c), the transfer of policies yields much worse results than policies trained on the respective domain. Still, CHILDSNACK is most difficult to transfer to, whereas policies trained on DRIVERLOG are least likely to transfer well. These results indicate that well-performing domain-dependent policies are tailored to the domain at hand and are not always directly transferable.

To analyze how transferable domain-independent policies are, we train each method in a *leave-one-domain-out* manner. We normalize the performance of the so found policies with respect to those trained on instances from all six domains. This indicates how well we can recover the performance on domains that were not observed during training. Figure 3 shows a confusion matrix analogously to Figure 2.

As in the domain-dependent case, policies for the LMC scenario are highly transferable as most often the preferred operator open list should be ignored (see Figure 3a). In the FF scenario, policies can be transferred much better than with the domain-dependent counterparts (see Figure 3b). On CHILDSNACK, we observe a drop in performance for all but the `embed` approach. In general both `dc` and `embed` approaches generalize better than `raw-F` and `no-F`. Finally, in the most complex scenario, the LAMA scenario, we again see slightly more pronounced drops in performance, but not as drastic as in the domain-dependent case (see Figure 3c). For example, the performance in VISITALL drops when it is excluded from the training set. This can be explained due to this being the only domain requiring strong usage of h_{lmc}. Excluding it from the training set does not enable methods to learn when this open list is useful. Interestingly, CHILDSNACK again proves to be a domain that is difficult to generalize to. Most notably though, in this scenario the performance drops most strongly when no instance features are used. These results gives further evidence that DAC policies trained in a domain-independent fashion are more transferable than their domain-dependent counterparts. Further, it seems to be important to train policies across domains to achieve domain-independence. Still, the use of instance features can help in learning general policies, especially true when used to learn low-dimensional embeddings.

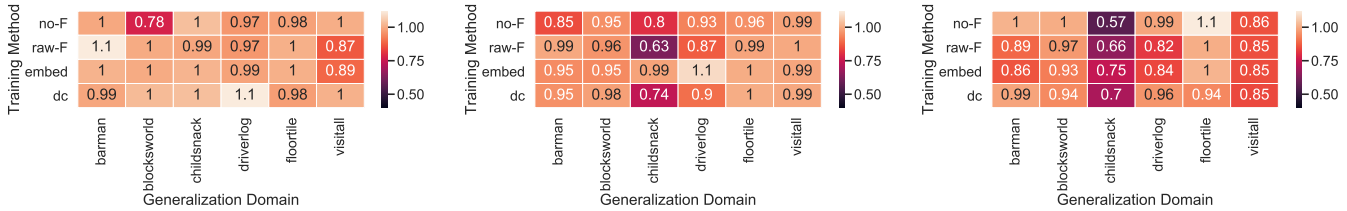
Understanding LAMA

From the data presented in the prior sections we can see that LAMA's policy exhibits some form of adaptation to the domain at hand as it can recover similar performance to the



(a) LMC scenario (b) FF scenario (c) LAMA scenario

Figure 2: Transferability of policies learned on a single domain. Values are normalized with respect to the performance gained on the same domain. Values closer to zero indicate worse performance of the learned policy on that domain. Values larger or equal to 1 indicates better or equal performance compared to the policy trained and evaluated in the same domain.



(a) LMC scenario (b) FF scenario (c) LAMA scenario

Figure 3: Transferability of policies learned on five domains to the left out domain. Values are normalized with respect to the performance achieved when training on all domains. Values closer to zero indicate worse performance. Values larger or equal to 1 indicates better or equal performance.

single best static policies in domains where they are especially strong. Moreover, we have seen that the exclusive use of the open list associated with $h^{\mathrm{p}}_{\mathrm{ff}}$ is already a strong policy. Based on these observations, it is likely that LAMA's boosting policy, which frequently boosts the $h^{\mathrm{p}}_{\mathrm{ff}}$ open list, is largely responsible for LAMA's success.

The use of DAC allows us to find DAC policies that are capable of outperforming LAMA on some instances and some domains. Comparing the found policies with that of LAMA provides insights that are not possible with classical meta-algorithmic approaches. As DAC allows to traverse the possible space of policies it can potentially find many differently behaving but well-performing policies which we can compare to that of LAMA. For all policies, we compare the frequency of usage of the individual open lists with those of the LAMA policy. Our findings show that, whenever a DAC policy performs worse than LAMA on a particular instance, then the DAC policy under-uses the $h^{\mathrm{p}}_{\mathrm{lmc}}$ open list. Out of the remaining three open lists, the DAC policy tends to use h_{lmc} slightly more on average than LAMA. For the two FF-related open lists, there is no clear trend. In cases where DAC finds policies that outperforms LAMA, LAMA over-uses the $h^{\mathrm{p}}_{\mathrm{lmc}}$ open list. Out of the remaining three open lists, successful DAC policies on average put more emphasize on $h^{\mathrm{p}}_{\mathrm{ff}}$ than LAMA. Thus, while LAMA's open list selection policy is robust and somewhat adapts to the domain at hand through boosting, it is not an optimal policy for all domains and instances, and equal boosting of both preferred operator open lists seems to not be the best strategy. Notably, while the frequency of $h^{\mathrm{p}}_{\mathrm{lmc}}$ varies, all successful policies make strong use of $h^{\mathrm{p}}_{\mathrm{ff}}$.

Conclusions

We investigated the use of DAC for learning policies for open list selection in a configuration space that includes LAMA, one of the most successful satisficing planners. If one is interested in finding the best policy (in terms of performance) for a specific domain, domain-dependent DAC is able to find policies that outperform even LAMA. However, these policies are tailored to this domain and fail to generalize to others. Whenever more robust policies are required, domain-independent DAC is capable of generalizing even to unseen domains to a certain degree, but might not produce policies being perfectly tailored to individual domains. Furthermore, DAC not only allows one to find strong policies, but also aids as a tool for gaining insights into AI planning algorithms, as the found policies highlight which open lists are important. Finally, when using RL as a solution approach to DAC, it is not trivial to include imperfect instance information during learning. Simply concatenating state features (as previously done) does not perform as well as using more specialized methods such as `dc` or `embed`.

Acknowledgments

This research was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215. David Speck was supported by the German Research Foundation (DFG) as part of the EPSDAC project (MA 7790/1-1). Marius Lindauer acknowledges funding by the German Research Foundation (DFG) under LI 2801/4-1. Jendrik Seipp was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

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. ECAI 2020*, 427–434.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2016. The IBaCoP Planning System: Instance-Based Configured Portfolios. *JAIR* 56: 657–691.

Cobbe, K.; Hesse, C.; Hilton, J.; and Schulman, J. 2020. Leveraging Procedural Generation to Benchmark Reinforcement Learning. In *Proc. ICML 2020*, 2048–2056.

Eimer, T.; Biedenkapp, A.; Hutter, F.; and Lindauer, M. 2021a. Self-Paced Context Evaluation for Contextual Reinforcement Learning. In *Proc. ICML 2021*, 2948–2958.

Eimer, T.; Biedenkapp, A.; Reimer, M.; Adriaensen, S.; Hutter, F.; and Lindauer, M. 2021b. DACBench: A Benchmark Library for Dynamic Algorithm Configuration. In *Proc. IJCAI 2021*, 1668–1674.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Domain-Specific Configuration using Fast Downward. In *ICAPS 2011 Workshop on Planning and Learning*, 13–17.

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*, 355–359.

Gerevini, A. E.; Saetti, A.; and Vallati, M. 2009. An Automatically Configurable Portfolio-Based Planner with Macro-Actions: PbP. In *Proc. ICAPS 2009*, 350–353.

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. 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.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – Instance-Specific Algorithm Configuration. In *Proc. ECAI 2010*, 751–756.

Kingma, D.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proc. ICLR 2015*.

Kirk, R.; Zhang, A.; Grefenstette, E.; and Rocktäschel, T. 2021. A Survey of Generalisation in Deep Reinforcement Learning. *arXiv:2111.09794 [cs.LG]* .

Lindauer, M.; Hoos, H.; Hutter, F.; and Schaub, T. 2015. AutoFolio: An automatically configured Algorithm Selector. *JAIR* 53: 745–778.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In *Proc. AAAI 2020*, 5077–5084.

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

Rice, J. 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.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks Revisited. In *Proc. AAAI 2008*, 975–982.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR* 39: 127–177.

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.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

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.

Speck, D.; Biedenkapp, A.; Hutter, F.; Mattmüller, R.; and Lindauer, M. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In *Proc. ICAPS 2021*, 597–605.

Tokui, S.; Okuta, R.; Akiba, T.; Niitani, Y.; Ogawa, T.; Saito, S.; Suzuki, S.; Uenishi, K.; Vogel, B.; and Vincent, H. Y. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proc. KDD 2019*, 2002–2011.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proc. ICAPS 2021*, 376–384.

van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proc. AAAI 2016*, 2094–2100.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR* 32: 565–606.

Zhang, A.; Sodhani, S.; Khetarpal, K.; and Pineau, J. 2021. Learning Robust State Abstractions for Hidden-Parameter Block MDPs. In *Proc. ICLR 2021*.