Reinforcement Learning for Planning Heuristics

Patrick Ferber[1,2] and **Malte Helmert**[1] and **Jörg Hoffmann**[2]

University of Basel[1]
Switzerland
{firstname}.{lastname}@unibas.ch

Saarland Informatics Campus, Saarland University[2]
Saarbrücken, Germany
{lastname}@cs.uni-saarland.de

Abstract

Informed heuristics are essential for the success of heuristic search algorithms. But, it is difficult to develop a new heuristic which is informed on various tasks. Instead, we propose a framework that trains a neural network as heuristic for the tasks it is supposed to solve.

We present two reinforcement learning approaches to learn heuristics for fixed state spaces and fixed goals. Our first approach uses *approximate value iteration*, our second approach uses *searches* to generate training data. We show that in some domains our approaches outperform previous work, and we point out potentials for future improvements.

Introduction

A key component in classical planning is heuristic search (Bonet and Geffner 2001). A search algorithm like $A*$ (Hart, Nilsson, and Raphael 1968) or *greedy best-first search* uses an *heuristic* as guidance to the goal. The heuristic estimates for every visited state its distance to the goal. The closer those estimates are to the true goal distance, the faster we expect the search algorithm to find a solution. Countless researchers designed heuristics for optimal (e.g., Helmert and Domshlak 2009; Helmert et al. 2014; Haslum et al. 2007) and satisficing (e.g., Hoffmann and Nebel 2001; Richter and Westphal 2010; Domshlak, Hoffmann, and Katz 2015) planning or invented ways to combine the power of multiple heuristics (e.g., Röger and Helmert 2010; Seipp 2017).

Instead of designing search algorithms or heuristics independently of the tasks to solve, another line of research develops algorithms that can be adapted for different tasks.

For example, offline portfolio algorithms learn a schedule which describes a planner order and a time limit per planner. To solve a new task, the planners are executed in their order with their time limits (e.g., Helmert et al. 2011; Seipp 2018). Online portfolio algorithms learn a mapping that decides for a given task which planner to use (Sievers et al. 2019; Ma et al. 2020). Gomoluch et al. (2020) apply reinforcement learning to learn how to modify a running search algorithm depending on some statistics. Their algorithm switches among others between best-first search, local search, random walks.

We also use reinforcement learning, but we do not change the behavior of the search algorithm. Instead, we learn a heuristic. Arfaee, Zilles, and Holte (2010) learn to combine multiple feature heuristics into a new heuristic. Iteratively, they use the feature heuristics and their learned heuristics to solve a set of task. From every solved task, the states along the solution together with their estimates of the feature heuristics and their estimated goal distance are saved. These estimates are used to improve the learned heuristic. If they are not able to solve sufficiently many new task, they generate training tasks by regressing from the goal. This is possible, because in their domains regression produce complete assignments and quickly leads to random states.

Agostinelli et al. (2019) learn heuristic functions using reinforcement learning with approximate value iteration. They also generate training states using random walks from the goal. They evaluate training states by minimizing over the heuristic estimates of the states' successors. Like previously, in their domains regression produces complete states and random walks quickly lead to random states (especially in the Rubik's Cube domain).

Ferber, Helmert, and Hoffmann (2020b) take another route. They use progression from some seed task to generate training states. They use an arbitrary heuristic search algorithm to solve sampled states and every state encountered along a plan is stored for training. The authors train their heuristics using supervised learning. Contrary to the previous two approaches they evaluate their technique on domains for which regression does not produce complete assignments.

Like Agostinelli et al. (2019) and Ferber, Helmert, and Hoffmann (2020b), we learn heuristics for fixed state spaces and fixed goals. We present two approaches that use reinforcement learning to train heuristics on tasks of the *International Planning Competition* (IPC). On some domains we already outperform previous approaches, and we identified important future steps that will further improve our performance. The paper is organized as follows. First, we provide some background on planning and reinforcement learning. Next, we present how we train our heuristics. Then, we evaluate our approach on IPC tasks. And finally, we resume our results and present our next steps.

Background

We work on planning tasks in *Finite-Domain Representation* (FDR) (Bäckström and Nebel 1995). An FDR task Π is defined as a quad-tuple $\langle V, A, I, G\rangle$. V is a set of finite-domain variables. Every variable v has a domain $dom(v)$ that contains all values assignable to it. A *state* assigns to every variable exactly one value. A fact is a $\langle var, val\rangle$ pair with $val \in dom(var)$. Two facts can be *mutually exclusive* (mutex), i.e. they cannot be part of the same state. A is a set of actions. Every action $a \in A$ is defined as $\langle pre_a, \mathit{eff}_a\rangle$ and has a cost associated. Both, pre_a and eff_a are partial assignments to V. An action a is applicable in a state s if $pre_a \subseteq s$. Applying action a in state s leads to a new state s' with $s' = \{v \mapsto \mathit{eff}_a[v] \mid v \in V \text{ and } v \in \mathit{eff}_a\} \cup \{v \mapsto s[v] \mid v \in V \text{ and } v \notin \mathit{eff}_a\}$. This is also called progression. I is the initial state and G is a partial assignment which describes the goal of the task. A state s is a goal state if $s \subseteq G$. A *plan* is a sequence of actions $\langle a_1, a_2, \ldots, a_n\rangle$ such that applying one action after another leads from the initial state to a goal state.

In this paper we do not only use progression, but also regression. A partial state p is regressable with an action a if, firstly, $\mathit{eff}_a \cap p \neq \emptyset$, secondly, there is no $v \in V$ such that $v \in \mathit{eff}_a$ and $v \in p$ and $\mathit{eff}_a[v] \neq p[v]$, and thirdly, there is no $v \in V$ such that $v \notin \mathit{eff}_a$ and $v \in pre_a$ and $v \in p$ and $pre_a[v] \neq p[v]$. The result of regressing the partial state p with the action a is defined as $\{var \mapsto val \mid var \mapsto val \in p \text{ and } var \notin \mathit{eff}_a\} \cup \{var \mapsto val \mid var \mapsto val \in pre_a\}$ (Alcázar et al. 2013).

Sometimes planning and machine learning use the same notations, but with different meanings. We annotate variables with ˜, if we use their machine learning meaning. We use reinforcement learning to learn a value function $\tilde{V} : S \mapsto \mathbb{R}$ that assigns every state a value. One technique to learn this function is *approximate value iteration* (AVI) (Bertsekas and Tsitsiklis 1996). We start with an arbitrary function $\tilde{V}_0$ and iteratively improve it using Equation 1.

$$\tilde{V}_{n+1} = \tilde{\mathcal{A}}\mathcal{T}\tilde{V}_n \tag{1}$$

$\mathcal{T}$ denotes the Bellman operator and is defined in Equation 2.

$$\mathcal{T}(s) = \max_a \left[\mathcal{R}_s^a + \delta \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a \tilde{V}(s')\right] \tag{2}$$

For a given state the Bellman operator provides an estimation of the expected total reward given the current value function $\tilde{V}_i$. We use the simplification of Agostinelli et al. (2019) shown in Equation 3. Our rewards $\mathcal{R}$ depend only on the action and can be replaced by the negated action cost. We do not need a weighted sum over possible successors, because our actions produce exactly one successor. We set δ to 1 such that the value function learns to estimate the remaining cost to the goal. All our rewards are negative (assuming planning tasks with non-negative costs). Therefore, we change the maximization to a minimization of negative rewards.

$$\mathcal{T}(s) = \min_a \left[cost(a) + \tilde{V}(s')\right] \tag{3}$$

$\tilde{\mathcal{A}}$ represents an approximation method that incorporates the sampled states and their values estimated by $\mathcal{T}$ and returns a new value function.

Training

We use reinforcement learning to train value functions that approximate the optimal heuristic for an FDR task Π. As approximation method $\tilde{\mathcal{A}}$ we use supervised learning. Our networks are residual network (He et al. 2016) with two dense layers followed by one residual block containing two dense layers and a single output neuron. Each dense layer contains 250 neurons. All neurons use the *ReLU* activation function. The inputs of our networks are states represented as fixed size Boolean vectors. We associate every entry of the input vector with a fact of Π. For all facts that are part of the input state we set their vector entries to 1. All other entries are set to 0. We train the network using the *mean squared error* as loss function and the *adam* optimizer with its default parameters (Kingma and Ba 2015). To prevent performance instabilities during training, we update the model for the sample generation after at least 50 epochs have passed and the mean squared error is below 0.1.

Because generating a training batch of 250 samples takes longer than training on that batch, we use experience replay. The data generating process pushes all samples into a *first-in-first-out* buffer with a maximum size of 25,000. In each training epoch we choose uniformly 250 samples from the buffer. This allows us to train multiple times on the same - recent - samples and to decouple the training from the data generation.

We run the data generation in four independent processes. Algorithm 1 provides an overview of them. Each process calls GENERATE_DATA and samples $\langle state, value\rangle$ pairs until we terminate it. First the process checks if a new value function is available. If yes, it loads the new value function. Then, the process samples a new state from the state space of Π using either SAMPLE_PROGRESSION or SAMPLE_REGRESSION. It evaluates the state using either EVALUATE_LOOKAHEAD or EVALUATE_SEARCH. EVALUATE_SEARCH can return multiple $\langle state, value\rangle$ pairs for each sampled state. Each $\langle state, value\rangle$ pair will be stored for training. Depending on some conditions, the process updates the parameters for the sampling methods.

SAMPLE_PROGRESSION starts at the initial state of the task Π, and performs a random walk for *walk_length* steps using progression. At each step it chooses a random applicable action which does not undo the previous action. SAMPLE_REGRESSION starts at the goal of Π, and performs a random walk for *walk_length* steps using regression. At each step it chooses a random regressable action which again does not undo the previous action. Unlike the progression walk, the regression walk ends with a partial assignment. We randomly complete the partial assignment to a state. Therefore, we assign every unassigned variable a value of its domain. We use the translator of Fast Downward (Helmert 2009) to

Algorithm 1 Generate Training Data

```
function SAMPLE_PROGRESSION(Π, walk_length)
    s, s' ← Π.I, None
    for i = 1..walk_length do
        a ← choose({a | a ∈ A, applicable(s, a) ∧
                        s' ≠ apply(s, a)})
        s, s' ← apply(s, a), s
    return s
function SAMPLE_REGRESSION(Π, walk_length)
    p, p' ← Π.G, None
    for i ← 1..walk_length do
        a ← choose({a | a ∈ A, regressable(p, a) ∧
                        p' ≠ regress(p, a)})
        p, p' ← regress(p, a), p
    return make_complete_assignment(p)
function EVALUATE_LOOKAHEAD(Π, s, Ṽ, lookahead)
    curr, succs ← [⟨s, 0⟩], []
    for i ← 1..lookahead do
        for s', c' ∈ curr do
            if is_goal(Π, s') then
                succs.insert(⟨s', c'⟩)
                continue
            for a ∈ {a | a ∈ A, applicable(Π, s', a)} do
                s'' ← apply(s', a)
                c'' ← c' + cost(Π, a)
                succs.insert(⟨s'', c''⟩)
        curr, succs ← succs, []
    c ← min({c' + (0 if is_goal(s') else Ṽ(s')) |
                s', c' ∈ curr})
    return [⟨s, c⟩]
function EVALUATE_SEARCH(Π, s, Ṽ, search)
    try
        plan ← search(Π, s, Ṽ)
        c ← sum([cost(a) | a ∈ plan])
        evals ← [⟨s, c⟩]
        for a ∈ plan do
            s, c ← apply(s, a), c − cost(Π, a)
            evals.insert(⟨s, c⟩)
        return evals
    exceptTIMEOUT, UNSOLVABLE
        return []
function GENERATE_DATE(Π, min_walk, max_walk, ls)
    while true do
        if value_function_outdated() then
            Ṽ ← load_value_function()
        s ← sample_X(Π, rnd(min_walk, max_walk))
        evals ← evaluate_Y(Π, s, Ṽ, ls)
        if s, v ∈ evals then
            store(s, v)
        if CONDITION(s, v) then
            min_walk ← update_min_walk_length()
            max_walk ← update_max_walk_length()
```

identify some mutexes of the task Π and enforce that none of them are violated.

To label the sampled state, we use either EVALUATE_LOOKAHEAD or EVALUATE_SEARCH. EVALUATE_LOOKAHEAD is an adaption of the simplified Bellman operator in Equation 3. Instead of considering only the direct successors of the current state, the function considers the n-step successors. If it finds a goal state during the n-step successor exploration, then it will not further explore the successors of this state. This is possible, because we want to learn the distance to the *closest* goal. Any successor of a goal state is further afar from us than the goal state itself. Every n-step successor is evaluated by adding up the action cost to reach it with its estimate of the value function. If a successor is a goal state, then the optimal value function would assign it to 0. Thus, we evaluate goal states with their action costs only. EVALUATE_SEARCH evaluates a state by executing a - potentially suboptimal - search on the state. If the search finds a plan, then it stores all states along the plan for training. Their associated values are the summed action costs from their position in the plan to the goal.

We use 2 different configuration to generate training data in our experiments. The first configuration samples all states using SAMPLE_REGRESSION and a random walk length between 0 and 300. It evaluates states using EVALUATE_LOOKAHEAD with a lookahead of 2. We call this configuration *approximate value iteration* (AVI). The second configuration samples for the first 10 hours with SAMPLE_REGRESSION and afterwards uses both SAMPLE_REGRESSION and SAMPLE_PROGRESSION. To evaluate the sampled states it uses EVALUATE_SEARCH. As search engine we use *greedy best-first search* with a timeout of 10s. In the beginning the value function is not good enough to solve sampled states far away from the goal. Therefore, the sampling starts with a walk length between 0 and 5. We double the maximum random walk length, if EVALUATE_SEARCH finds a plan for more than 95% of the sampled states. We observed that at some point further increasing the random walk length does not sample states further away from the goal, but just wastes computational time. Thus, we double at most 8 times the maximum walk length. We call this configuration *sampling search* (SaSe).

We run the training - including data generation - for 28 hours on 4 cores of an Intel Xeon E5-2600 processor with 3.8 GB of memory. For training the neural networks we used the Keras framework (Chollet 2015) with Tensorflow (Abadi et al. 2015) as back-end. We implemented the data generation and all searches in Fast Downward (Helmert 2006), and used Lab (Seipp et al. 2017) to setup our experiments.

Experiments

We train neural networks as heuristics to solve different tasks from the same state space and with the same goal. We evaluate our training procedure on the domains Ferber, Helmert, and Hoffmann (2020b) used. They selected a subset of tasks which they deemed hard enough to be interesting, but also easy enough for them to generate training data. We call their task selection *moderate tasks*. Because the data

Domain	AVI	SaSe	SL	Lama
blocks	0.0	0.0	**98.0**	96.8
depots	17.7	39.7	64.3	**98.7**
grid	51.0	86.0	74.0	**97.0**
npuzzle	1.0	1.5	0.0	**97.8**
pipesworld-notankage	29.8	50.4	92.8	**97.2**
rovers	25.8	35.8	12.5	**98.0**
scanalyzer-opt11-strips	83.3	33.3	77.7	**97.7**
storage	47.5	**71.5**	22.0	37.5
transport-opt14-strips	69.0	70.5	**98.0**	97.5
visitall-opt14-strips	13.0	30.7	0.7	**95.0**
Average	33.8	41.9	54.0	**91.3**

Table 1: Coverage of LAMA and greedy best-first search with heuristics trained using approximate value iteration (AVI), sampling search (SaSe), or supervised learning (SL) on the moderate tasks.

generation is not a bottleneck in our method, we also consider the harder tasks they skipped. We call these *hard tasks*. For every task selected, we have 50 different initial states for testing. We use the test states provided for the moderate tasks by Ferber, Helmert, and Hoffmann (2020b). For the other tasks, we generate new test states in the same way they did. We start at the initial state of the original task and perform a 200 step forward random walk.

For every task, we train a network using the approximate value iteration (AVI) configuration and a network using the sampling search (SaSe) configuration. To solve the test tasks, we use our neural networks as heuristics in an eager greedy best-first search. Exploratory experiments have shown that the eager version of greedy best-first search performs better with our heuristic than the lazy version. We run each search for 10 hours with a memory limit of 3.8 GB on a single core of an Intel Xeon Silver 4114. We use the first iteration of *LAMA* (Richter and Westphal 2010) as baseline. On the moderate tasks we also compare to the networks of Ferber, Helmert, and Hoffmann (2020b) used in a greedy best-first search. Because they used supervised learning, we call this baseline *supervised learning* (SL).

All code, benchmarks, and experimental results are online available (Ferber, Helmert, and Hoffmann 2020a).

Moderate Tasks

Table 1 shows the coverage of our configurations (AVI, SaSe) against the supervised learning (SL) and the LAMA baseline on the moderate tasks. On average LAMA outperforms all other techniques. Given enough time to generate its training data (400 hours) the supervised training approach solves more tasks than our *current* approach. From our two approaches, the sampling search configuration outperforms the approximate value iteration approach.

A more detailed view shows that whether an approach works well or not depends on the domain. There are some domains on which the supervised learning approach works well, but our reinforcement learning approach does not work at all (e.g. Blocksworld) and some domains where the reinforcement learning works better than the supervised learning(e.g. VisitAll). In Storage we outperform not only the supervised learning approach, but also LAMA.

Domain	AVI	SaSe	Lama
depots	15.1	6.9	**80.6**
grid	0.0	0.0	**90.0**
npuzzle	0.0	0.0	**84.0**
pipesworld-notankage	1.4	25.1	**68.7**
rovers	0.1	0.8	**97.7**
scanalyzer-opt11-strips	34.0	3.3	**98.7**
storage	18.8	**26.5**	11.0
visitall-opt14-strips	0.0	36.0	**98.0**
Average	8.7	12.3	**78.6**

Table 2: Coverage of LAMA and greedy best-first search with heuristics trained using approximate value iteration (AVI), sampling search (SaSe) on the hard tasks.

Figure 1 shows for each domain how the coverage increases over time. We see that LAMA quickly reaches its coverage limit. The supervised learning approach takes a bit longer. The two reinforcement learning approaches require the most time until they converge to their final coverage.

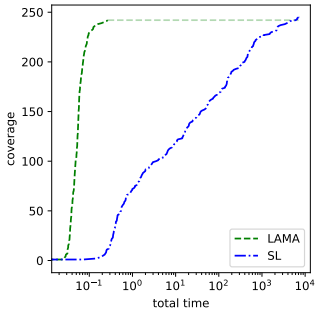
Hard Tasks

Table 2 shows the coverage on the hard tasks. The Blocksworld and Transport domains have no tasks in this category. The hard tasks show a similar picture than the moderate tasks. All techniques solve fewer tasks, but LAMA is still the best technique, and the sampling search configuration is still better than the approximate value iteration technique. Furthermore, in the Storage domain reinforcement learning outperforms LAMA.

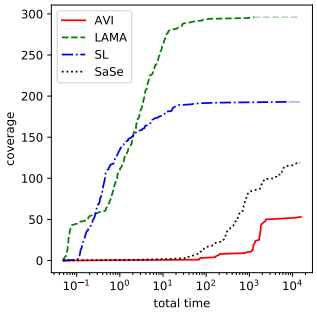
Robustness

We observe that for some tasks within a domain our approaches solve either almost none or almost all states (see Table 3, columns *x1*). We speculate that the randomness during training sometimes produces good and sometimes bad models. To verify this, we train for the domains Depots and Scanalyzer four additional models per task. We run each of our five models on a fifth of the test states. If our assumption is correct, then we expect for the same task some models which solve almost all test states, and some models which solve almost no test states. Column *x5* of Table 3 shows for each of the five models how many test states they solves. As expected, most of the trained models solve either all or none of their test states and for the same task it is possible to obtain good and bad models. We also see that for some tasks it is more likely to train a good model than for other tasks.

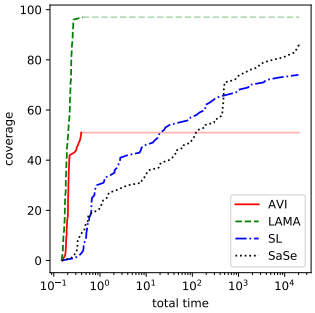
This raises the question of what would be our performance if we could detect which models are good? For every task we select the model with the highest coverage out of the five models in column *x5* and use those on all test states. Columns *x1'* shows that this greatly increases our coverage.



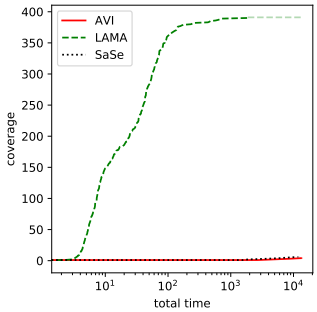
(a) Blocks



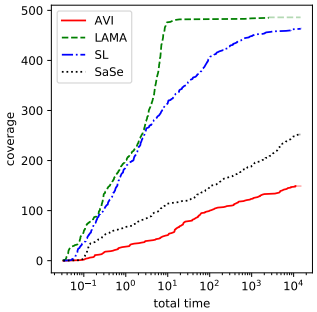
(b) Depots



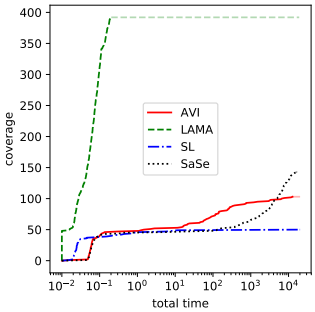
(c) Grid



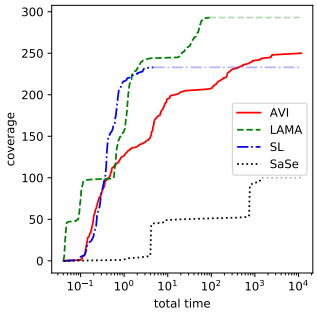
(d) NPuzzle



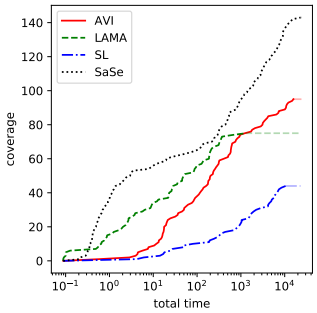
(e) Pipesworld-NoTankage



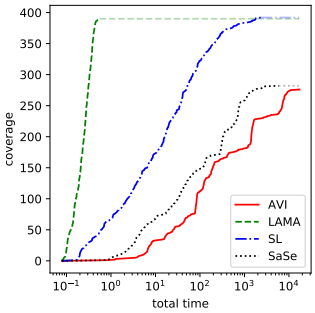
(f) Rovers



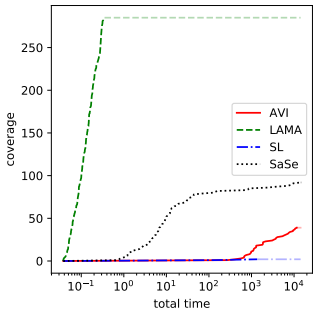
(g) Scanalyzer-opt11-strips



(h) Storage

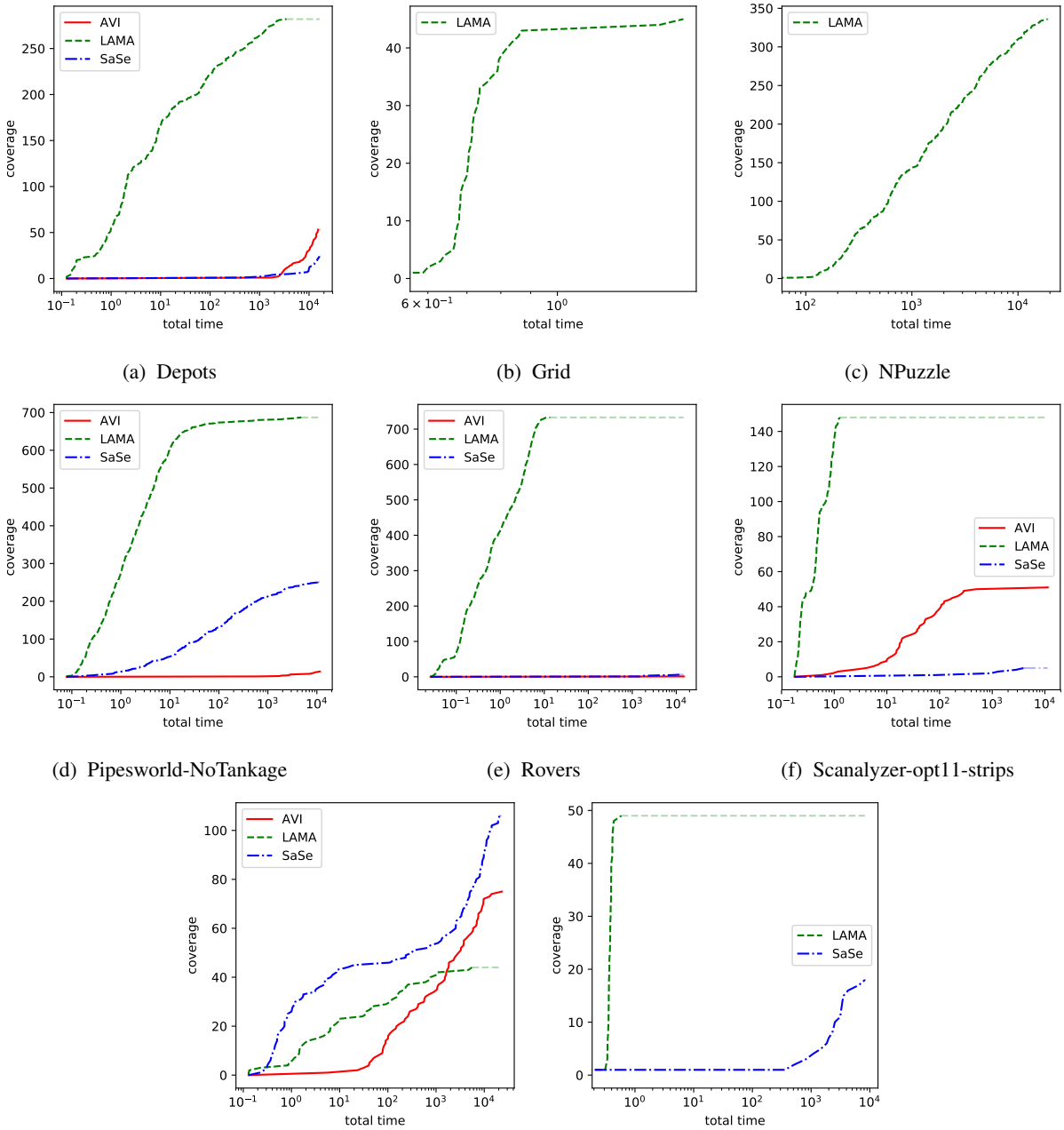


(i) Transport-opt14-strips



(j) Visitall-opt14-strips

Figure 1: Cumulative coverage of LAMA and greedy best-first search with the heuristic trained using approximate value iteration (AVI), sampling search (SaSe), and the supervised learning (SL) baseline on the moderate tasks.



(a) Depots (b) Grid (c) NPuzzle

(d) Pipesworld-NoTankage (e) Rovers (f) Scanalyzer-opt11-strips

(g) Storage (h) Visitall-opt14-strips

Figure 2: Cumulative coverage of LAMA and greedy best-first search with the heuristic trained using approximate value iteration (AVI), sampling search (SaSe), and the supervised learning (SL) baseline on the hard tasks.

	AVI							SaSe						
Depots	x1	x5					x1'	x1	x5					x1'
p05	**50**	10	10	10	9	9	**50**	0	10	10	0	0	0	**41**
p06	0	0	0	0	0	0	0	0	1	1	1	0	0	**10**
p08	1	7	5	2	0	0	**28**	19	10	10	6	6	0	**47**
p09	0	8	0	0	0	0	**32**	**50**	10	1	0	0	0	44
p11	2	10	1	1	0	0	**50**	0	10	7	0	0	0	**49**
p12	0	0	0	0	0	0	0	**5**	0	0	0	0	0	3
p14	11	10	10	9	9	3	**50**	**1**	0	0	0	0	0	0
p15	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p16	0	9	7	0	0	0	**45**	**50**	10	10	10	0	0	**50**
p18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p19	**42**	7	2	2	0	0	**42**	18	6	3	1	0	0	**23**
p20	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p22	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum	106	150					**297**	143	123					**267**

	AVI							SaSe						
Scanalyzer	x1	x5					x1'	x1	x5					x1'
p07	0	2	0	0	0	0	**16**	0	0	0	0	0	0	0
p10	**50**	10	9	3	0	0	**50**	0	0	0	0	0	0	0
p13	**50**	10	9	9	9	9	**50**	**50**	10	10	10	10	10	**50**
p15	**50**	10	10	10	9	8	**50**	**50**	10	10	10	10	10	**50**
p16	**50**	10	10	10	10	10	**50**	0	0	0	0	0	0	0
p17	**50**	10	10	9	8	1	**50**	0	0	0	0	0	0	0
p18	1	8	8	3	0	0	**49**	0	0	0	0	0	0	0
p19	0	0	0	0	0	0	0	5	2	0	0	0	0	**11**
p20	**50**	10	9	9	9	9	**50**	0	10	0	0	0	0	**50**
Sum	301	288					**365**	105	112					**161**

Table 3: Absolute coverage for greedy best-first search using heuristics trained by the approximate value iteration (AVI) configuration and by the sampling search (SaSe) configuration. *x1* uses one model for all test states. *x5* uses five models (10 states per model). *x1'* uses the best model from *x5* on all test states. (Top) Shows the results for Depots. (Bottom) Shows the results for Scanalyzer.

	Moderate Tasks			Hard Tasks	
Domain	AVI	SaSe	SL	AVI	SaSe
depot	68.8	77.0	64.3	26.3	10.3
scanalyzer	88.7	50.0	77.7	66.0	7.3

Table 4: Coverage on Depots and Scanalyzer if the best models from Table 3 column x5 are used to solve all test states.

The performance of a model on a subset of test states approximates well the performance of the model on all test states. Table 4 shows how this increases the coverage fractions. For the moderate tasks of Depots this increases the coverage by 50% (AVI) resp. 40% (SaSe) and our approach would outperform the supervised learning baseline.

A followup question is, how can we detect whether a model will be good on the test states? We saw that a subset of test states approximates well the performance of model on all test states. Thus, a first approach could be to create an additional set of validation states which is independent of the test states. The performance of every trained model is evaluated on the validation states, and we select the model with the best performance on the validation states.

Conclusion

We presented two approaches to learn heuristics for fixed state spaces and goals using reinforcement learning. The first one uses approximate value iteration the other one uses a search in its data generation. We showed that our approaches can outperform the previous state of the art on some domains while requiring only $1/16$ of the time for training. Furthermore, our approach can easily be applied to hard planning tasks.

We observed that most of our trained models are either very good or very bad at solving the test states. We presented a naive test to detect good models and showed that this can drastically improve our coverage. In our next steps, we plan to examine how we can detect during training if a model is on its way to become a good or bad model. We expect to see the performance boost shown for Depots and Scanalyzer also on the other domains.

We also observed that increasing the random walk length to sample states farther away from the goal suffers from diminishing returns. Adding 10 additional steps to the walk does not lead us 10 steps farther away from the goal. We started preliminary experiments which use a known heuristic or the current value function as bias. We believe that biasing the random walk, especially with the trained value function, is an essential step for learning the heuristic estimates of large state spaces.

Acknowledgments

Patrick Ferber was funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science). This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan).

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Man'e, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Fern; a Vi'egas; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence* 1:356–363.

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *Proc. IJCAI 2013*, 2254–2260.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap learning of heuristic functions. In *Proc. SoCS 2010*, 52–60.

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

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

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

Chollet, F. 2015. Keras. https://keras.io.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *AIJ* 221:73–114.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020a. Code, benchmarks and experiment data for the PRL 2020 workshop paper "Reinforcement Learning for Planning Heuristics". https://doi.org/10.5281/zenodo.4049617.

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

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

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, 1007–1012.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 planner abstracts*, 38–45.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM* 61(3):16:1–63.

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

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

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *Proc. ICLR 2015*.

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.

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

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In *Proc. SoCS 2017*, 149–153.

Seipp, J. 2018. Fast Downward Remix. In *IPC-9 planner abstracts*, 74–76.

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.