Reinforcement Learning for Classical Planning: Viewing Heuristics as Dense Reward Generators

Clement Gehring[*1], **Masataro Asai**[*2], **Rohan Chitnis**[1], **Tom Silver**[1], **Leslie Pack Kaelbling**[1], **Shirin Sohrabi**[3], **Michael Katz**[3]

[1]MIT, [2]MIT-IBM Watson AI Lab, [3]IBM Research

{gehring,lpk}@csail.mit.edu, {ronuchit,tslvr}@mit.edu, {masataro.asai,michael.katz1}@ibm.com, ssohrab@us.ibm.com

Abstract

Recent advances in reinforcement learning (RL) have led to a growing interest in applying RL to classical planning domains or applying classical planning methods to some complex RL domains. However, the long-horizon goal-based problems found in classical planning lead to sparse rewards for RL, making direct application inefficient. In this paper, we propose to leverage domain-independent heuristic functions commonly used in the classical planning literature to improve the sample efficiency of RL. These classical heuristics act as dense reward generators to alleviate the sparse-rewards issue and enable our RL agent to learn domain-specific value functions as residuals on these heuristics, making learning easier. Correct application of this technique requires consolidating the discounted metric used in RL and the non-discounted metric used in heuristics. We implement the value functions using Neural Logic Machines, a neural network architecture designed for grounded first-order logic inputs. We demonstrate on several classical planning domains that using classical heuristics for RL allows for good sample efficiency compared to sparse-reward RL. We further show that our learned value functions generalize to novel problem instances in the same domain. The source code and the appendix are available at github.com/ibm/pddlrl and arxiv.org/abs/2109.14830.

Introduction

Deep reinforcement learning (RL) approaches have several strengths over conventional approaches to decision making problems, including compatibility with complex and unstructured observations, little dependency on hand-crafted models, and some robustness to stochastic environments. However, they are notorious for their poor sample complexity; e.g., it may require 10^{10} environment interactions to successfully learn a policy for Montezuma's Revenge (Badia et al. 2020). This sample inefficiency prevents their applications in environments where such an exhaustive set of interactions is physically or financially infeasible. The issue is amplified in domains with sparse rewards and long horizons, where the reward signals for success are difficult to obtain through random interactions with the environment.

In contrast, research in AI Planning and classical planning has been primarily driven by the identification of tractable fragments of originally PSPACE-complete planning problems (Bylander 1994), and the use of the cost of the tractable *relaxed* problem as *domain-independent* heuristic guidance for searching through the state space of the original problem. Contrary to RL approaches, classical planning has focused on long-horizon problems with solutions well over 1000 steps long (Jonsson 2007; Asai and Fukunaga 2015). Moreover, classical planning problems inherently have sparse rewards — the objective of classical planning is to produce a sequence of actions that achieves a goal. However, although domain-independence is a welcome advantage, domain-independent methods can be vastly outperformed by carefully engineered domain-specific methods such as a specialized solver for Sokoban (Junghanns and Schaeffer 2000) due to the no-free-lunch theorem for search problems (Wolpert, Macready et al. 1995). Developing such domain-specific heuristics can require intensive engineering effort, with payoff only in that single domain. We are thus interested in developing domain-independent methods for *learning* domain-specific heuristics.

In this paper, we draw on the strengths of reinforcement learning and classical planning to propose an RL framework for learning to solve STRIPS planning problems. We propose to leverage classical heuristics, derivable automatically from the STRIPS model, to accelerate RL agents to learn a domain-specific neural network value function. The value function, in turn, improves over existing heuristics and accelerates search algorithms at evaluation time.

To operationalize this idea, we use *potential-based reward shaping* (Ng, Harada, and Russell 1999), a well-known RL technique with guaranteed theoretical properties. A key insight in our approach is to see classical heuristic functions as providing *dense rewards* that greatly accelerate the learning process in three ways. First, they allow for efficient, informative exploration by initializing a good baseline reactive agent that quickly reaches a goal in each episode during training. Second, instead of learning the value function directly, we learn a *residual* on the heuristic value, making learning easier. Third, the learning agent receives a reward by reducing the estimated cost-to-go (heuristic value). This effectively mitigates the issue of sparse rewards by allowing the agent to receive positive rewards more frequently.

We implement our neural network value functions as Neural Logic Machines (Dong et al. 2019, NLM), a recently pro-

[*]These authors contributed equally.

posed neural network architecture that can directly process first-order logic (FOL) inputs, as are used in classical planning problems. NLM takes a dataset expressed in grounded FOL representations and learns a set of (continuous relaxations of) lifted Horn rules. The main advantage of NLMs is that they structurally *generalize* across different numbers of terms, corresponding to objects in a STRIPS encoding. Therefore, we find that our learned value functions are able to generalize effectively to problem instances of arbitrary sizes in the same domain.

We provide experimental results that validate the effectiveness of the proposed approach in 8 domains from past IPC (International Planning Competition) benchmarks, providing detailed considerations on the reproducibility of the experiments. We find that our reward shaping approach achieves good sample efficiency compared to sparse-reward RL, and that the use of NLMs allows for generalization to novel problem instances. For example, our system learns from blocksworld instances with 2-6 objects, and the result enhances the performance of solving instances with up to 50 objects.

Background

We denote a multi-dimensional array in bold. $\boldsymbol{a}; \boldsymbol{b}$ denotes a concatenation of tensors $\boldsymbol{a}$ and $\boldsymbol{b}$ in the last axis. Functions (e.g., $\log, \exp$) are applied to arrays element-wise.

Classical Planning

We consider planning problems in the STRIPS subset of PDDL (Fikes and Nilsson 1972), which for simplicity we refer to as lifted STRIPS. We denote such a planning problem as a 5-tuple $\langle O, P, A, I, G\rangle$. O is a set of objects, P is a set of predicates, and A is a set of actions. We denote the arity of predicates $p \in P$ and action $a \in A$ as $\#p$ and $\#a$, and their parameters as, e.g., $X = (x_1, \cdots, x_{\#a})$. We denote the set of predicates and actions instantiated on O as $P(O)$ and $A(O)$, respectively, which is a union of Cartesian products of predicates/actions and their arguments, i.e., they represent the set of all ground propositions and actions. A state $s \subseteq P(O)$ is a set of propositions that are true in that state. An action is a 4-tuple $\langle \textsc{pre}(a), \textsc{add}(a), \textsc{del}(a), \textsc{cost}(a)\rangle$, where $\textsc{pre}(a), \textsc{add}(a), \textsc{del}(a) \in P(X)$ are preconditions, add-effects, and delete-effects, and $\textsc{cost}(a) \in \mathbb{R}$ is a cost of taking the action a. In this paper, we primarily assume a unit-cost domain where $\textsc{cost}(a) = 1$ for all a. Given a *current state* s, a ground action $a_\dagger \in A(O)$ is *applicable* when $\textsc{pre}(a_\dagger) \subseteq s$, and applying an action $a_\dagger$ to s yields a *successor state* $a_\dagger(s) = (s \setminus \textsc{del}(a_\dagger)) \cup \textsc{add}(a_\dagger)$. Finally, $I, G \subseteq P(O)$ are the initial state and a goal condition, respectively. The task of classical planning is to find a *plan* $(a^1_\dagger, \cdots, a^n_\dagger)$ which satisfies $a^n_\dagger \circ \cdots \circ a^1_\dagger(I) \supseteq G$ and every action $a^i_\dagger$ satisfies its preconditions at the time of using it. The machine representation of a state s and the goal condition G is a bitvector of size $|P(O)| = \sum_{p\in P} O^{\#p}$, i.e., the i-th value of the vector is 1 when the corresponding i-th proposition is in s, or G.

Markov Decision Processes

In general, RL methods address domains modeled as a discounted Markov decision processes (MDP), $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, r, q_0, \gamma\rangle$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $T(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ encodes the probability $\Pr(s'|s, a)$ of transitioning from a state s to a successor state s' by an action a, $r(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function, q_0 is a probability distribution over initial states, and $0 \le \gamma < 1$ is a discount factor. In this paper, we restrict our attention to deterministic models because PDDL domains are deterministic, and we have a deterministic mapping $T' : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Given a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ representing a probability $\Pr(a|s)$ of performing an action a in a state s, we define a sequence of random variables $\{S_t\}_{t=0}^{\infty}$, $\{A_t\}_{t=0}^{\infty}$ and $\{R_t\}_{t=0}^{\infty}$, representing states, actions and rewards over time t.

Our goal is to find a policy maximizing its *long term discounted cumulative rewards*, formally defined as a *value function* $V_{\gamma,\pi}(s) = \mathbb{E}_{A_t\sim\pi(S_t,\cdot)}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s\right]$. We also define an *action-value* function to be the value of executing a given action and subsequently following some policy π, i.e., $Q_{\gamma,\pi}(s, a) = \mathbb{E}_{S_1\sim T(s,a,\cdot)}\left[R_0 + \gamma V_{\gamma,\pi}(S_1) \mid S_0 = s, A_0 = a\right]$. An *optimal policy* π^* is a policy that achieves the *optimal value function* $V^*_\gamma = V_{\gamma,\pi^*}$ that satisfies $V^*_\gamma(s) \ge V_{\gamma,\pi}(s)$ for all states and policies. V^*_γ satisfies *Bellman's equation*:

$$V^*_\gamma(s) = \max_{a\in\mathcal{A}} Q^*_\gamma(s, a) \quad \forall s \in \mathcal{S}, \tag{1}$$

where $Q^*_\gamma = Q_{\gamma,\pi^*}$ is referred to as the *optimal action-value function*. We may omit π in $V_{\gamma,\pi}$, $Q_{\gamma,\pi}$ for clarity.

Finally, we can define a policy by mapping action-values in each state to a probability distribution over actions. For example, given an action-value function, Q, we can define a policy $\pi(s, a) = \textsc{softmax}(Q(s, a)/\tau)$, where $\tau > 0$ is a temperature that controls the greediness of the policy. It returns a greedy policy $\arg\max_a Q(s, a)$ when $\tau \to 0$; and approaches a uniform policy when $\tau \to \infty$.

Formulating Classical Planning as an MDP

There are two typical ways to formulate a classical planning problem as an MDP. In one strategy, given a transition (s, a, s'), one may assign a reward of 1 when $s' \in G$, and 0 otherwise (Rivlin, Hazan, and Karpas 2019). In another strategy, one may assign a reward of 0 when $s \in G$, and -1 otherwise (or, more generally $-\textsc{cost}(a)$ in a non-unit-cost domain). In this paper we use the second, *negative-reward* model because it tends to induce more effective exploration in RL due to optimistic initial values (Sutton and Barto 2018). Both cases are considered sparse reward problems because there is no information about whether one action sequence is better than another until a goal state is reached.

Bridging Deep RL and AI Planning

We consider a multitask learning setting with a training time and a test time (Fern, Khardon, and Tadepalli 2011). During training, classical planning problems from a single domain

are available. At test time, methods are evaluated on held-out problems from the same domain. The transition model (in PDDL form) is known at both training and test time.

Learning to improve planning has been considered in RL. For example, in AlphaGo (Silver et al. 2016), a value function was learned to provide heuristic guidance to Monte Carlo Tree Search (Kocsis and Szepesvári 2006). Applying RL techniques in our classical planning setting, however, presents unique challenges.

(P1): Preconditions and dead-ends. In MDPs, a failure to perform an action is typically handled as a self-cycle to the current state in order to guarantee that the state transition probability T is well-defined for all states. Another formulation augments the state space with an absorbing state with a highly negative reward. In contrast, classical planning does not handle non-deterministic outcomes (success and failure). Instead, actions are forbidden at a state when its preconditions are not satisfied, and a state is called a dead-end when no actions are applicable. In a self-cycle formulation, random interaction with the environment could be inefficient due to repeated attempts to perform inapplicable actions. Also, the second formulation requires assigning an ad-hoc amount of negative reward to an absorbing state, which is not appealing.

(P2): Objective functions. While the MDP framework itself does not necessarily assume discounting, the majority of RL applications aim to maximize the expected cumulative *discounted* rewards of trajectories. In contrast, classical planning tries to minimize the sum of costs (negative rewards) along trajectories, i.e., cumulative *undiscounted* costs, thus carrying the concepts in classical planning over to RL requires caution.

(P3): Input representations. While much of the deep RL literature assumes an unstructured (e.g., images in Atari) or a factored input representation (e.g., location and velocity in cartpole), classical planning deals with structured inputs based on FOL to perform domain- and problem-independent planning. This is problematic for typical neural networks, which assume a fixed-sized input. Recently, several network architectures were proposed to achieve invariance to size and ordering, i.e., neural networks for *set*-like inputs (Zaheer et al. 2017). Graph/Hypergraph Neural Networks (Scarselli et al. 2009; Rivlin, Hazan, and Karpas 2019; Shen, Trevizan, and Thiébaux 2020; Ma et al. 2020, GNNs/HGNs) have also been recently used to encode FOL inputs. While the choice of the architecture is arbitrary, our network should be able to handle FOL inputs.

Value Iteration for Classical Planning

Our main approach will be to learn a value function that can be used as a heuristic to guide planning. To learn estimated value functions, we build on the *value iteration* (VI) algorithm (line 1, Algorithm 1), where a known model of the dynamics is used to incrementally update the estimates $V_{\gamma,\pi}(s)$ of the optimal value function $V_{\gamma,\pi^*}(s)$. The current estimates $V_{\gamma,\pi}(s)$ is updated by the r.h.s. of Eq. 1 until a fixpoint is reached. In classical planning, however, state spaces are too large to enumerate its states (line 3), or to represent the estimates $V_{\gamma,\pi}(s)$ in a tabular form (line 4).

Algorithm 1: VI, RTDP, RTDP for Classical Planning

1: *Value Iteration (VI)*:
2: **while** not converged **do**
3: **for** $s \in \mathcal{S}$ **do**
4: $V_{\gamma,\pi}(s) \leftarrow \max_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a)$
5: *Approximate RTDP with Replay Buffer*:
6: Buffer $B \leftarrow \emptyset$
7: **while** not converged **do**
8: $s \sim q_0$, $t \leftarrow 0$
9: **while** $t < D$ and s is non-terminal **do**
10: $a \leftarrow \arg\max_a Q_{\gamma,\pi}(s,a)$
11: $s \leftarrow T'(s,a)$
12: B.*push*(s)
13: SGD$(\frac{1}{2}(V_{\gamma,\pi}(s) - \mathbb{E}_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a))^2, B)$
14: $t \leftarrow t+1$
15: *Approximate RTDP for Classical Planning*:
16: Buffer $B \leftarrow [\emptyset, \emptyset, \emptyset, \ldots]$
17: **while** not converged **do**
18: $\langle \mathcal{D}, O, I, G\rangle \sim q_0$, $t \leftarrow 0$, $s \leftarrow I$,
19: **while** $t < D$, $s \notin G$, s is not a deadlock **do**
20: $a \leftarrow \arg\max_{a\in\{a|\text{PRE}(a)\subseteq s\}} Q_{\gamma,\pi}(s,a)$
21: $s \leftarrow T'(s,a)$
22: $B[|O|]$.*push*(s)
23: SGD$(\frac{1}{2}(V_{\gamma,\pi}(s) - \mathbb{E}_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a))^2, B)$
24: $t \leftarrow t+1$

To avoid the exhaustive enumeration of states in VI, Real Time Dynamic Programming (Sutton and Barto 2018, RTDP, line 5) samples a subset of the state space based on the current policy. In this work, we use *on-policy* RTDP, which replaces the second $\max_a$ with $\mathbb{E}_a$ (line 13) for the current policy defined by the SOFTMAX of the current action-value estimates. On-policy methods are known to be more stable but can sometimes lead to slower convergence.

Next, to avoid representing the value estimates in an exhaustive table, we encode $V_{\gamma,\pi}$ using a neural network parameterized by weights θ and applying the Bellman updates approximately with Stochastic Gradient Descent (line 13).

As a common practice called *experience replay* (Lin 1993; Mnih et al. 2015), we store the state history into a fixed-sized FIFO buffer B (lines 6-12), and update $V_{\gamma,\pi}(s)$ with mini-batches sampled from B to leverage GPU parallelism. The oldest record retires when $|B|$ reaches a limit.

We modify RTDP to address the assumptions **(P1)** in classical planning, resulting in line 15. First, in our multitask setting, where goals vary between problem instances, we wish to learn a single goal-parameterized value function that generalizes across problems (Schaul et al. 2015). We omitted the goal for notational concision, but all of our value functions are implicitly goal-parameterized, i.e., $V(s) = V(s,G)$.

Next, problem instances with different numbers of objects have state representations (tensors) of varying sizes and dimensions. Such a set of arrays with non-uniform shapes makes it challenging from a mini-batch processing on GPUs. Moreover, since larger problem instances typically require more steps to solve, states from these problems are likely to

dominate the replay buffer. This can make updates to states from smaller problems rare, which can lead to catastrophic forgetting. To address this, we separate the buffer into buckets (line 22), where states in one bucket are from problem instances with the same number of objects. When we sample a mini-batch, we randomly select a bucket and randomly select states from this bucket.

Next, instead of terminating the inner loop and sampling the initial state in the same state space, we redefine q_0 to be a distribution of problem instances, and select a new training instance and start from its initial state (line 18).

Finally, since $\arg\max_a$ in RTDP is not possible at a state with no applicable actions (a.k.a. *deadlock*), we reset the environment upon entering such a state (line 19). We also select actions only from applicable actions and do not treat an inapplicable action as a self-cycle (line 20). Indeed, training a value function along a trajectory that includes self-cycles has no benefit because the test-time agents never execute them due to duplicate detection.

Planning Heuristics as Dense Rewards

The fundamental difficulty of applying RL-based approaches to classical planning is the lack of dense reward to guide exploration. We address this by combining heuristic functions (e.g., $h^{\text{FF}}, h^{\text{add}}$) with a technique called *potential-based reward shaping*. To correctly perform this technique, we should take care of the difference between the discounted and non-discounted objectives **(P2)**.

Potential-based reward shaping (Ng, Harada, and Russell 1999) is a technique that helps RL algorithms by modifying the reward function r. Formally, with a *potential function* $\phi : \mathcal{S} \to \mathbb{R}$, a function of states, we define a shaped reward function on transitions, $\hat{r} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, as follows:

$$\hat{r}(s, a, s') = r(s, a, s') + \gamma\phi(s') - \phi(s). \tag{2}$$

Let $\hat{\mathcal{M}}$ be a MDP with a shaped reward $\hat{r}$, and $\mathcal{M}$ be the original MDP. When the discount factor $\gamma < 1$, or when the MDP is *proper*, i.e., every policy eventually ($t \to \infty$) reaches a terminal state with probability 1 under $\gamma = 1$, any optimal policy $\hat{\pi}^*$ of $\hat{\mathcal{M}}$ is an optimal policy π^* of $\mathcal{M}$ regardless of ϕ, thus RL converges to an policy optimal in the original MDP $\mathcal{M}$. Also, the optimal value function $\hat{V}^*_\gamma$ under $\hat{\mathcal{M}}$ satisfies

$$V^*_\gamma(s) = \hat{V}^*_\gamma(s) + \phi(s). \tag{3}$$

In other words, an agent trained in $\hat{\mathcal{M}}$ is learning an offset of the original optimal value function from the potential function. The potential function thus acts as prior knowledge about the environment, which initializes the value function to non-zero values (Wiewiora 2003).

Building on this theoretical background, we propose to leverage existing domain-independent heuristics to define a potential function that guides the agent while it learns to solve a given domain. A naive approach that implements this idea is to define $\phi(s) = -h(s)$. The h value is negated because the MDP formulation seeks to *maximize* reward and h is an estimate of cost-to-go, which should be minimized. Note that the agent receives an additional reward when $\gamma\phi(s') - \phi(s)$ is positive (Eq. 2). When $\phi = -h$, this means that approaching toward the goal and reducing h is treated as a reward signal. Effectively, this allows us to use a domain-independent planning heuristic to generate dense rewards that aid in the RL algorithm's exploration.

However, this straightforward implementation has two issues: **(1)** First, when the problem contains a dead-end, the function may return ∞, i.e., $h : \mathcal{S} \to \mathbb{R}^{+0} \cup \{\infty\}$. This causes a numerical error in gradient-based optimization. **(2)** Second, the value function still requires a correction even if h is the "perfect" oracle heuristic h^*. Recall that V^*_γ is the optimal discounted value function with -1 rewards per step. Given an optimal unit-cost cost-to-go $h^*(s)$ of a state s, the discounted value function and the non-discounted cost-to-go can be associated as follows:

$$V^*_\gamma(s) = \sum_{t=1}^{h^*(s)} \gamma^t \cdot (-1) = -\frac{1-\gamma^{h^*(s)}}{1-\gamma} \neq -h^*(s). \tag{4}$$

Therefore, the amount of correction needed (i.e., $\hat{V}^*_\gamma(s) = V^*_\gamma(s) - \phi(s)$) is not zero even in the presence of an oracle $\phi = -h^*$. *This is a direct consequence of discounting difference.*

To address these issues, we propose to use the *discounted value of the heuristic function* as a potential function. Recall that a heuristic function $h(s)$ is an estimate of the cost-to-go from the current state s to a goal. Since $h(s)$ does not provide a *concrete* idea of how to reach a goal, we tend to treat it as a black box. An important realization, however, is that it nevertheless represents a sequence of actions; thus its value can be *decomposed into a sum of action costs* (below, left), and we define a corresponding *discounted heuristic function* $h_\gamma(s)$ (below, right):

$$h(s) = \sum_{t=1}^{h(s)} 1, \quad h_\gamma(s) = \sum_{t=1}^{h(s)} \gamma^t \cdot 1 = \frac{1-\gamma^{h(s)}}{1-\gamma}. \tag{5}$$

Notice that $\phi = -h^*_\gamma$ results in $\hat{V}^*_\gamma(s) = 0$. Also, h_γ is bounded within $[0, \frac{1}{1-\gamma}]$, avoiding numerical issues.

Value-Function Generalized over Problems

To learn *domain-dependent, instance-independent* heuristics, the value function used in the reward-shaping framework discussed above must be invariant to the number, the order, and the textual representation of propositions and objects in a PDDL definition **(P3)**. We propose the use of Neural Logic Machines (Dong et al. 2019, NLMs), a differentiable ILP system for a learning task over FOL inputs. Below, we describe how it works and how we encode and pass states s and goal condition G to NLM to obtain $V(s, G)$.

Neural Logic Machines NLMs represent a state in terms of binary arrays representing the truth value of each proposition. Propositions are grouped by the arity N of the predicates they were grounded from. This forms a set of $(N+1)$-d arrays denoted as $\boldsymbol{z}/N$, where the leading dimensions are indexed by objects and the last dimension is indexed by predicates of arity N. For example, when we have objects a,

b, c and four binary predicates on, connected, above and larger, we enumerate all combinations on(a,a), on(a,b) ... larger(c,c), resulting in an array $\boldsymbol{z}/2 \in [0,1]^{3\times3\times4}$. Similarly, we may have $\boldsymbol{z}/1 \in [0,1]^{3\times2}$ for 2 unary predicates, and $\boldsymbol{z}/3 \in [0,1]^{3\times3\times3\times5}$ for 5 ternary predicates. The total number of elements in all arrays combined matches the number of propositions $|P(O)| = \sum_{p\in P} O^{\#p}$. In the following, we call this representation a Multi-Arity Predicate Representation (MAPR).

NLMs are designed to learn a class of FOL rules with the following set of restrictions: Every rule is a Horn rule, no rule contains function terms (such as a function that returns an object), there is no recursion, and all rules are applied between neighboring arities. Due to the lack of recursion, the set of rules can be stratified into layers. Let P_k be a set of intermediate conclusions in the k-th stratum. The following set of rules are sufficient for representing any rules in this class of rules (Dong et al. 2019):

$$\text{(expand)} \quad \forall x_{\#\overline{p_k}}; \overline{p_k}(X; x_{\#\overline{p_k}}) \leftarrow p_k(X),$$

$$\text{(reduce)} \quad \underline{p_k}(X) \leftarrow \exists x_{\#p_k}; p_k(X; x_{\#p_k}),$$

$$\text{(compose)} \quad p_{k+1}(X) \leftarrow \mathcal{F}\left(\bigcup_{\pi}\left(\left(P_k \cup \underline{P_k} \cup \overline{P_k}\right)/\#p_{k+1}\right)(\pi(X))\right)\right).$$

Here, $p_k, \underline{p_k}, \overline{p_k}, p_{k+1} \in P_k, \underline{P_k}, \overline{P_k}, P_{k+1}$ (respectively) are predicates, $X = (x_1, \ldots)$ is a sequence of parameters, and $\mathcal{F}(T)$ is a formula consisting of logical operations $\{\wedge, \vee, \neg\}$ and terms T. Intermediate predicates $\underline{p_k}$ and $\overline{p_k}$ have one less / one more parameters than p_k, e.g., when $\#p_k = 3$, $\#\overline{p_k} = 4$ and $\#\underline{p_k} = 2$. $(P_k \cup \underline{P_k} \cup \overline{P_k})/\#p_{k+1}$ extracts the predicates whose arity is the same as that of p_{k+1}. $\pi(X)$ is a permutation of X, and $\bigcup_\pi$ iterates over π to generate propositional groundings with various argument orders. $\mathcal{F}(\cdot)$ represents a formula that combines a subset of input propositions. By chaining these set of rules from P_k to P_{k+1} for a sufficient number of times (e.g., from P_1 to P_5), it is able to represent any FOL Horn rules without recursions (Dong et al. 2019).

All three operations (expand, reduce, and compose) can be implemented as tensor operations over MAPRs (Figure 1). Given a binary tensor $\boldsymbol{z}/n$ of shape $O^n \times |P/n|$, *expand* copies the n-th axis to $n+1$-th axis resulting in a shape $O^{n+1} \times |P/n|$, and *reduce* takes the $\max$ of n-th axis resulting in a shape $O^{n-1} \times |P/n|$, representing $\exists$.

Finally, the COMPOSE operation combines the information between the neighboring tensors $\boldsymbol{z}/n, \boldsymbol{z}/_{n-1}, \boldsymbol{z}/_{n+1}$. In order to use the information in the neighboring arities (P, $\underline{P}$ and $\overline{P}$), the input concatenates $\boldsymbol{z}/n$ with $\text{EXPAND}(\boldsymbol{z}/_{n-1})$ and $\text{REDUCE}(\boldsymbol{z}/_{n+1})$, resulting in a shape $O^n \times C$ where $C = |P/n| + |P/_{n-1}| + |P/_{n+1}|$. Next, a PERM function enumerates and concatenates the results of permuting the first n axes in the tensor, resulting in a shape $O^n \times (!n \cdot C)$. It then applies a n-D pointwise convolutional filter f_n with Q output features, resulting in $O^n \times Q$, i.e., applying a fully connected layer to each vector of length $!n \cdot C$ while sharing the weights. It is activated by any nonlinearity σ to obtain the final result, which is a sigmoid activation function in our implementation. We denote the result as

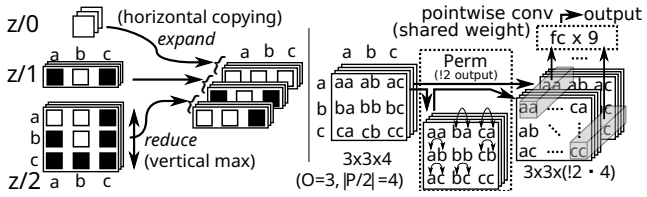


Figure 1: (Left) EXPAND and REDUCE operations performed on a boolean MAPR containing nullary, unary, and binary predicates and three objects, a, b, and c. Each white / black square represents a boolean value (true / false). (Right) COMPOSE operation performed on 4 binary predicates. Each predicate is represented as a 3×3 matrix, resulting in a $\mathbb{R}^{3\times3\times4}$ tensor. For a matrix, PERM is equivalent to concatenating the matrix with its transposition, resulting in a $\mathbb{R}^{3\times3\times(!2\cdot4)}$ tensor. After PERM, a shared fully-connected layer is applied to each combination of arguments (such an operation is sometimes called a pointwise convolution).

$\text{COMPOSE}(\boldsymbol{z}, n, Q, \sigma)$. Formally, $\forall j \in 1..n, \forall o_j \in 1..O$,

$$\Pi(\boldsymbol{z}) = \text{PERM}\Big(\text{EXPAND}(\boldsymbol{z}/_{n-1});\ \boldsymbol{z}/n;\ \text{REDUCE}(\boldsymbol{z}/_{n+1})\Big),$$

$$\text{COMPOSE}(\boldsymbol{z}, n, Q, \sigma)_{o_1\cdots o_n} = \sigma(f_n(\Pi(\boldsymbol{z})_{o_1\cdots o_n})) \in \mathbb{R}^Q.$$

An NLM contains N (the maximum arity) COMPOSE operation for the neighboring arities, with appropriately omitting both ends (0 and $N+1$) from the concatenation. We denote the result as $\text{NLM}_{Q,\sigma}(\boldsymbol{z}) = (\text{COMPOSE}(\boldsymbol{z}, 1, Q, \sigma), \cdots, \text{COMPOSE}(\boldsymbol{z}, N, Q, \sigma))$. These horizontal arity-wise compositions can be layered vertically, allowing the composition of predicates whose arities differ more than 1 (e.g., two layers of NLM can combine unary and quaternary predicates). Since f_n is applied in a convolutional manner over O^n object tuples, the number of weights in an NLM layer does not depend on the number of objects in the input. However, it is still affected by the number of predicates in the input, which alters C.

NLMs/GNNs/HGNs We prefer NLMs over GNNs/HGNs for two reasons. First, unlike GNNs/HGNs, operations in NLMs have *clear logical interpretations*: $\forall/\exists$ (REDUCE / EXPAND) and combining input formula with different arguments (COMPOSE). Next, the arity in NLMs' hidden layers can be EXPAND-ed arbitrarily large. GNNs are limited to binary/unary relations (edge/node-embeddings). The arity of a hidden HGN layer can be higher, but must match the input. FactorGNN (Zhang, Wu, and Lee 2020) is similar.

Value Function as NLMs To represent a goal-generalized value function $V(s, G)$ with NLMs, we concatenate each element of two sets of binary arrays: One set representing the current state and another the goal conditions. The last dimension of each array in the resulting set is twice larger.

When the predicates in the input PDDL domain have a maximum arity N, we specify the maximum intermediate arity M and the depth of NLM layers L as a hyperparameter. The intermediate NLM layers expand the arity up to M using EXPAND operation, and shrink the arity near the output because a value function has a scalar (arity 0) output.

For example, with $N = 2$, $M = 3$, $L = 7$, the arity of each layer follows $(2, 3, 3, 3, 2, 1, 0)$. Higher arities are not necessary near the output because the information in each layer propagates only to the neighboring arities. Since each expand/reduce operation only increments/decrements the arity by one, L, N, M must satisfy $N \leq M \leq L$. Intermediate conclusions in NLM is fixed to $Q = 8$.

The output of this NLM is unactivated, similar to a regression task, because we use its raw value as the predicted correction to the heuristic function. In addition, we implement NLM with a *skip connection* that was popularized in ResNet image classification network (He et al. 2016): The input of l-th layer is a concatenation of the outputs of all previous layers. Due to the direct connections between the layers in various depths, the layers near the input receive more gradient information from the output, preventing the gradient vanishing problem in deep neural networks.

Experimental Evaluation

Our objective is to see whether our RL agent can improve the efficiency of a Greedy Best-First Search (GBFS), a standard algorithm for solving satisficing planning problems, over a domain-independent heuristic. The efficiency is measured by the number of node-evaluations performed during search. We also place an emphasis on generalization: We hope that NLMs are able to generalize from smaller training instances with fewer objects to instances with more objects.

We train our RL agent with rewards shaped by h^{FF} and h^{add} heuristics obtained from `pyperplan` library. We write blind heuristic $\forall s; h^{\text{bl}}(s) = 1$ to denote a baseline without shaping. While our program is compatible with a wide range of unit-cost IPC domains (see the list of 25 domains in the appendix), we focus on extensively testing its selected subset with a large enough number of independently trained models with different random seeds (20), to produce high-confidence results. This is because RL algorithms tend to have a large variance in their outcomes (Henderson et al. 2018), induced by sensitivity to initialization, randomization in exploration, and randomization in experience replay.

We trained our system on five domains in (Rivlin, Hazan, and Karpas 2019): 4-ops blocksworld, ferry, gripper, logistics, satellite, and three additional IPC domains: miconic, parking, and visitall. In all domains, we generated problem instances using existing parameterized generators (Fawcett et al. 2011). For each domain, we provided between 195 and 500 instances for training, and between 250 and 700 instances for testing. The generator parameters for test instances contain the ranges used for IPC instances. We remove trivial instances whose initial states satisfy the goals, which are produced by the generators occasionally, especially for small parameter values used for training instances. Each agent is trained for 50000 steps, which takes about 4 to 6 hours on Xeon E5-2600 v4 and Tesla K80. The domain generator parameters and the training hyperparameters can be found in the appendix.

We ran GBFS on the test instances using $-V(s, G) = -\hat{V}(s, G) + h_\gamma(s)$ as a heuristics. (Discounting does not affect the expansion order in GBFS and unit cost domains.) Instead of setting time or memory limits, we limited the maximum node evaluations in GBFS to 100,000. If a problem was solved within the limit, the configuration gets the score 1 for that instance, otherwise it gets 0. The sum of the scores for each domain is called the coverage in that domain. Table 1 shows the coverage in each of the tested domains, comparing our configurations to the baselines, as well as to the prior work (Section). The baselines are denoted by their heuristic (e.g., h^{FF} is the GBFS with h^{FF}), while our heuristics, obtained by a training with reward shaping $\phi = -h_\gamma$, are denoted with a capital H (e.g., H^{FF}). Additionally, Figure 2 goes beyond the pure coverage and compares the node evaluations. These results answer the following questions:

(Q1) Do our agents learn heuristic functions at all, i.e., is $\text{GBFS}(h^{\text{bl}}) < \text{GBFS}(H^{\text{bl}})$ (**green** dots in Figure 2), where $\text{GBFS}(h^{\text{bl}})$ is similar to breadth-first search with duplicate detection, and $\text{GBFS}(H^{\text{bl}})$ is baseline RL without reward shaping? With the exception of visitall and miconic, $\text{GBFS}(h^{\text{bl}})$ could not solve any instances in the test set, while using the heuristics learned without shaping ($\text{GBFS}(H^{\text{bl}})$) significantly improved coverage in 5 of the 6 domains.

(Q2) Do they improve over the baselines they were initialized with, i.e., is $\text{GBFS}(h) < \text{GBFS}(H)$? In domains where they did not solve every instances (blocks, ferry, logistics, parking, satellite), Table 1 suggests that the reward-shaping-based training has successfully improved the coverage in blocks, ferry, parking. Since the number of solved instances is not a useful metric in domains where both configurations solved nearly all instances (lack of coverage improvement does not imply lack of improvement in efficiency), we next compare the number of node evaluations, which directly evaluates the search efficiency. Figure 2 shows that the search effort tends to be reduced, especially on the best seed. However, this is sensitive to the random seed, and the improvement is weak on logistics. These results suggest that while RL can improve the planning efficiency, we need several iterations of random experiments to achieve improvements due to the high randomness of RL.

(Q3) Do our agents with reward shaping outperform our agents without shaping? According to Table 1, H^{FF} and H^{add} outperforms H^{bl}. Notice that h^{FF} and h^{add} also outperform h^{bl}. This suggest that the informativeness of the base heuristic used for reward shaping affects the quality of the learned heuristic. This matches the theoretical expectation: The potential function plays the role of domain knowledge that initializes the policy.

(Q4) Did heuristics accelerate exploration during training and contribute to the improvement? Table 2 shows the number of goals reached during training, indicating that reward shaping helps the agent receive real rewards at goals more often. See the appendix figures for cumulative plots.

(Q5) Do the learned heuristics maintain their improvement in larger problem instances, i.e., do they generalize to more objects? Figure 2 (Right) plots the number of objects (x-axis) and the ratio of success (y-axis) over blocks instances. The agents are trained on 2-6 objects while evaluated on 10-50 objects. It shows that the heuristic accuracy is improved in instances whose size far exceeds the training instances for $h^{\text{bl}}, h^{\text{FF}}, h^{\text{add}}$. Due to space limitations, plots for

	Baselines			Ours (mean±stderr (max) of 20 runs)			GBFS	GBFS		GBFLS (incomparable)	
domain (total)	h^{bl}	h^{add}	h^{FF}	H^{bl}	H^{add}	H^{FF}	-HGN	-H	-V	(-H	-V)
blocks (250)	0	126	87	**73.1±2.8(94)**	**186.6±7.5(229)**	**104±1.5(114)**	3	208	0	250	250
ferry (250)	0	138	250	**40.4±3.2(62)**	**233.9±4(249)**	⚠ 250±0(250)	27	240	0	250	250
gripper (250)	0	250	250	**47.5±5(85)**	⚠ 250±0(250)	⚠ 250±0(250)	63	139	0	250	250
logistics (250)	0	106	243	0±0(0)	54.1±6.8(**115**)	79.8±12.9(189)	-	0	0	30	33
miconic (442)	171	442	442	143.3±6.8(**246**)	⚠ 442±0(442)	⚠ 440.8±1.3(442)	-	0	0	442	442
parking (700)	0	607	700	**0.9±0.2(3)**	**619±32.4(689)**	⚠ 696.9±0.5(700)	-	333	0	403	357
satellite (250)	0	249	222	**26.5±5(99)**	⚠ 233.3±5.2(250)	163.2±11(205)	-	9	0	137	135
visitall (252)	252	252	252	207.6±5.3(238)	⚠ 251.9±0.1(252)	⚠ 252±0(252)	-	101	0	249	249

Table 1: Coverage of GBFS with 100,000 node evaluations limit. Our scores are highlighted in bold and underline when our average score is significantly better/worse than the baseline ($|\text{ours} - \text{baseline}| > \text{stderr}$, where $\text{stderr} = \frac{\text{stdev}}{\sqrt{20}}$). Some scores are ⚠ marked for caution when both ours and baselines solved nearly all instances. They were too easy to measure coverage differences, and thus the lack thereof does not imply the lack of improvement in the heuristics. Instead, node evaluation plots (Figure 2) show reduction in the search effort. In conclusion, our approach generally improves the performance in many domains except logistics, where only the best seed of h^{add} managed to improve upon the baseline ($106 \rightarrow 115$).

domain	h^{bl}	h^{add}	h^{FF}
blocks	362±42	527±58	**621±31**
ferry	516±52	**976±19**	949±21
gripper	275±33	**673±18**	600±20
logistics	93±21	**502±33**	**488±32**
miconic	542±18	708±9	**722±9**
parking	400±51	**809±24**	**814±27**
satellite	212±37	**658±34**	**654±26**
visitall	211±22	198±20	**350±17**

Table 2: The cumulative number of goal states the agent has reached during training. The numbers are average and standard deviation over 20 seeds. Best numbers among heuristics are highlighted in bold, with ties equally highlighted when there are no statistically significant differences between them under Wilcoxon's rank-sum test ($p \geq 0.05$). The results indicate that reward shaping significantly accelerates the exploration compared to no shaping (h^{bl}).

the remaining domains are in the appendix.

Comparison with Previous Work Next, we compared our learned heuristics with two recent state-of-the-art learned heuristics. The first approach, STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020), is a supervised learning method that can learn domain-dependent or domain-independent heuristics depending on the dataset. It uses hypergraph networks (HGN), a generalization of Graph Neural Networks (GNNs) (Scarselli et al. 2009). The authors have provided us with pre-trained weights for three domains: gripper, ferry, and blocksworld for the domain-dependent setting. STRIPS-HGN was originally developed and evaluated for use with A^*, for obtaining near-optimal plans. Since we do not optimize plan quality in this work, we instead use it with GBFS to find goals more quickly at the expense of plan quality. We recognize it may not have been designed for this scenario and that therefore may not best demonstrate its strengths. It remains useful as a baseline point of comparison for our work. We denote this variant GBFS-HGN.

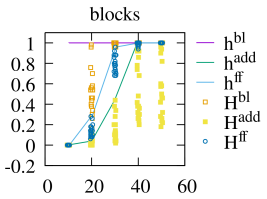
The second approach we compare to is GBFS-GNN (Rivlin, Hazan, and Karpas 2019), an RL-based heuristic learning method that trains a GNN-based value function. The authors use Proximal Policy Optimization (Schulman et al. 2017), a state of the art RL method that stabilizes the training by limiting the amount of policy change in each step (the updated policy stays in the proximity of the previous policy). The value function $V(s)$ is a GNN optionally equipped with attentions (Veličković et al. 2018; Vaswani et al. 2017). In addition, the authors proposed to adjust $V(s)$ by the policy $\pi(a|s)$ and its entropy $H_\pi = \sum_a \pi(a|s) \log \pi(a|s)$. The heuristic value of the successor state $s' = a(s)$ is given by $h(s') = \frac{\pi(a|s)V(s)}{1+H_\pi}$. We call it an *entropy-adjusted value function*.

The authors also proposed a variant of GBFS which launches a greedy informed local search after each expansion. We distinguish their algorithmic improvement and the heuristics improvement by naming their search algorithm as Greedy Best First Lookahead Search (GBFLS). Our formal rendition of GBFLS can be found in the appendix.

We counted the number of test instances that are solved by these approaches within 100,000 node evaluations. In the case of GBFLS, the evaluations also include the nodes that appear during the lookahead. We evaluated GBFS-HGN on the domains where pretrained weights are available. For GBFS-GNN, we obtained the source code from the authors (private communication) and minimally modified it to train on the same training instances that we used for our approach. We evaluated 4 variants of GBFS-GNN: GBFS-H, GBFS-V, GBFLS-H, and GBFLS-V, where "H" denotes entropy-adjusted value function, and "V" denotes the original value function. Note that fair evaluation should compare our method with GBFS-H/V, not GBFLS-H/V.

Table 1 shows the results. We first observe that large part of the success of GBFS-GNN should be attributed to the lookahead extension of GBFS. This is because the score

	H^{bl}	h^{bl}	H^{add}	h^{add}	H^{FF}	h^{FF}
blocks	94	0	224	5	109	9
ferry	62	0	249	0	250	0
gripper	85	0	250	0	200	50
logistics	0	0	108	14	167	77
miconic	234	17	381	61	442	0
parking	2	0	508	105	484	173
satellite	93	0	115	110	155	63
visitall	192	60	216	36	192	59



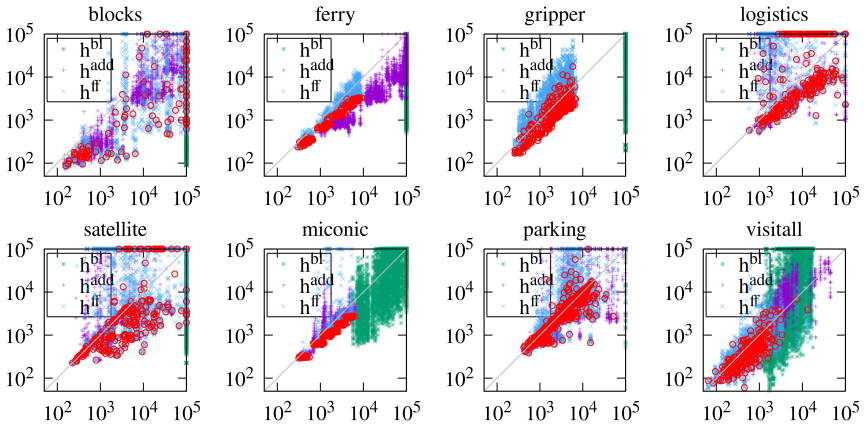


Figure 2: Best on computer screens. Remaining domains are in the appendix. (Left, Top) Number of instances where GBFS(H^x) had less evaluations than GBFS(h^x) did (and vice versa), excluding ties and instances failed by both. GBFS(H^x) is a result of the best training seed with the least sum of evaluations across the instances in a domain. They show that RL improves the performance both with (H^{FF}, H^{add}) and without (H^{bl}) reward shaping in the best case. (Right) Scatter plot showing the number of node evaluations on 8 domains, where x-axis is for GBFS with $h^{\text{bl}}, h^{\text{FF}}, h^{\text{add}}$ and y-axis is for $H^{\text{bl}}, H^{\text{FF}}, H^{\text{add}}$. Each point corresponds to a single test problem instance. Results of 20 random seeds are plotted against a single deterministic baseline. Failed instances are plotted on the border. Red points highlight the best seed from H^{FF}. (Left, Bottom) The rate of finding a solution (y-axis) for blocks instances and the number of objects (x-axis), demonstrating that the improvements generalize to larger instances. The agents are trained on 2-6 objects while the test instances contain 10-50 objects.

is GBFLS-V $\gg$ GBFS-H $\gg$ GBFS-V, i.e., GBFLS-V performs very well even with a bad heuristics ($V(s)$). While we report the coverage for both GBFLS-H/V and GBFS-H/V, the configurations that are comparable to our setting are GBFS-H/V. First, note that GBFS-HGN is significantly outperformed by all other methods. Comparing to the other two, both H^{add} and H^{FF} outperform GBFS-H in 7 out of the 8 domains, losing only on blocks. It is worth noting that H^{bl} outperforms GBFS-H in miconic, satellite, and visitall. Since both H^{bl} and GBFS-H are trained without reward shaping, the difference is due to the network shape (NLM vs GNN) and the training (Modified RTDP vs PPO).

Related Work

Early attempts to learn heuristics include shallow, fully connected neural networks (Arfaee, Zilles, and Holte 2011), its online version (Thayer, Dionne, and Ruml 2011), combining SVMs (Cortes and Vapnik 1995) and NNs (Satzger and Kramer 2013), learning a residual from heuristics (Yoon, Fern, and Givan 2008), or learning a relative ranking between states (Garrett, Kaelbling, and Lozano-Pérez 2016). More recently, Ferber, Helmert, and Hoffmann (2020) tested fully-connected layers in modern frameworks. ASNet (Toyer et al. 2018) learns domain-dependent heuristics using a GNN-like network. They are based on supervised learning methods that require the high-quality training dataset (accurate goal distance estimates of states) that are prepared separately. Our RL-based approaches explore the environment by itself to collect data, which is automated (pros) but could be sample-inefficient (cons).

A large body of work utilize ILP techniques to learn a value function (Gretton and Thiébaux 2004) features (Wu and Givan 2010), pruning rules (Krajnanský et al. 2014), or a policy function by classifying the best action (Fern, Yoon, and Givan 2006). NLM is a differentiable ILP system that subsumes first-order decision lists / trees used in ILP.

Other RL-based approaches include Policy Gradient with FF to accelerate exploration for probabilistic PDDL (Buffet, Aberdeen et al. 2007), and PPO-based Meta-RL (Duan et al. 2016) for PDDL3.1 discrete-continuous hybrid domains (Gutierrez and Leonetti 2021). They do not use reward shaping, thus our contributions are orthogonal.

Grounds and Kudenko (2005) combined RL and STRIPS planning with reward shaping, but in a significantly different setting: They treat a 2D navigation as a two-tier hierarchical problem where unmodified FF (Hoffmann and Nebel 2001) or Fast Downward (Helmert 2006) are used as high-level planner, then their plans are used to shape the rewards for the low-level RL agent. They do not train the high-level planner.

Conclusion

In this paper, we proposed a domain-independent reinforcement learning framework for learning domain-specific heuristic functions. We addressed the difficulty of training an RL agent with sparse rewards using a novel reward-shaping technique that leverages existing heuristic functions. We showed that our framework not only learns a heuristic function from scratch (H^{bl}), but also learns better if aided by heuristic functions (reward shaping). Furthermore, the learned heuristics outperform the baseline across a wide range of problem sizes, demonstrating its generalization over the number of objects in the environment.

Acknowledgments

We thank Shen, Trevizan, and Thiébaux and Rivlin, Hazan, and Karpas for kindly providing us their code.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artif. Intel.*, 175(16-17).

Asai, M.; and Fukunaga, A. 2015. Solving Large-Scale Planning Problems by Decomposition and Macro Generation. In *ICAPS*.

Badia, A. P.; Piot, B.; Kapturowski, S.; Sprechmann, P.; Vitvitskyi, A.; Guo, Z. D.; and Blundell, C. 2020. Agent57: Outperforming the Atari Human Benchmark. In *ICML*. PMLR.

Buffet, O.; Aberdeen, D.; et al. 2007. FF+FPG: Guiding a Policy-Gradient Planner. In *ICAPS*.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artif. Intel.*, 69(1).

Cortes, C.; and Vapnik, V. 1995. Support-Vector Networks. *Mach. Learn.*, 20(3).

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines. In *ICLR*.

Duan, Y.; Schulman, J.; Chen, X.; Bartlett, P. L.; Sutskever, I.; and Abbeel, P. 2016. RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning. *CoRR*, abs/1611.02779.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Domain-Specific Configuration using Fast Downward. In *PAL*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *ECAI*.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The First Learning Track of the International Planning Competition. *Mach. Learn.*, 84(1-2).

Fern, A.; Yoon, S.; and Givan, R. 2006. Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes. *J. Artif. Intell. Res.*, 25.

Fikes, R. E.; and Nilsson, N. J. 1972. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intel.*, 2(3).

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *IJCAI*.

Gretton, C.; and Thiébaux, S. 2004. Exploiting First-Order Regression in Inductive Policy Selection. In *UAI*.

Grounds, M.; and Kudenko, D. 2005. Combining Reinforcement Learning with Symbolic Planning. In *AAMAS*. Springer.

Gutierrez, R. L.; and Leonetti, M. 2021. Meta Reinforcement Learning for Heuristic Planing. In *ICAPS*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26.

Henderson, P.; Islam, R.; Bachman, P.; Pineau, J.; Precup, D.; and Meger, D. 2018. Deep Reinforcement Learning that Matters. In *AAAI*, 1.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res.*, 14.

Jonsson, A. 2007. The Role of Macros in Tractable Planning over Causal Graphs. In *IJCAI*.

Junghanns, A.; and Schaeffer, J. 2000. Sokoban: A Case-Study in the Application of Domain Knowledge in General Search Enhancements to Increase Efficiency in Single-Agent Search. *Artif. Intel.*

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *ECML*. Springer.

Krajnanský, M.; Hoffmann, J.; Buffet, O.; and Fern, A. 2014. Learning Pruning Rules for Heuristic Search Planning. In *ECAI*.

Lin, L.-J. 1993. Reinforcement Learning for Robots using Neural Networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In *AAAI*, 04.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. 2015. Human-Level Control through Deep Reinforcement Learning. *Nat.*, 518(7540).

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy Invariance under Reward Transformations: Theory and Application to Reward Shaping. In *ICML*.

Rivlin, O.; Hazan, T.; and Karpas, E. 2019. Generalized Planning With Deep Reinforcement Learning. In *PRL*.

Satzger, B.; and Kramer, O. 2013. Goal Distance Estimation for Automated Planning using Neural Networks and Support Vector Machines. *Natural Computing*, 12(1).

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *IEEE T. Neural Networ.*, 20(1).

Schaul, T.; Horgan, D.; Gregor, K.; and Silver, D. 2015. Universal Value Function Approximators. In *ICML*. PMLR.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *ICAPS*.

Silver, D.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nat.*, 529(7587).

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Thayer, J.; Dionne, A.; and Ruml, W. 2011. Learning Inadmissible Heuristics during Search. In *ICAPS*, 1.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *AAAI*, 1.

Vaswani, A.; Shazeer, N.; Parmar, N.; et al. 2017. Attention is All You Need. In *Neurips*.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2018. Graph Attention Networks. In *ICLR*.

Wiewiora, E. 2003. Potential-Based Shaping and Q-value Initialization are Equivalent. *J. Artif. Intell. Res.*, 19.

Wolpert, D. H.; Macready, W. G.; et al. 1995. No Free Lunch Theorems for Search. Technical report, SFI-TR-95-02-010.

Wu, J.-H.; and Givan, R. 2010. Automatic Induction of Bellman-Error Features for Probabilistic Planning. *J. Artif. Intell. Res.*, 38.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *J. Mach. Learn. Res.*, 9(4).

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep Sets. In *Neurips*.

Zhang, Z.; Wu, F.; and Lee, W. S. 2020. Factor Graph Neural Networks. *Neurips*, 33.