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]

*: equal contributions. [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 and vise versa. 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 our RL agent learns domain-specific value functions as residuals on these heuristics, making learning easier. Proper application of this technique requires consolidating the discounted metric in RL and non-discounted metric 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.

1 Introduction

In the last two decades, research in AI Planning and classical planning has been primarily driven by the advancement of sophisticated heuristic forward search techniques, especially by the identification of tractable fragments of originally PSPACE-complete planning problems (Bäckström and Klein 1991; Bylander 1994; Erol, Nau, and Subrahmanian 1995; Jonsson and Bäckström 1998a,b; Brafman and Domshlak 2003; Katz and Domshlak 2008a,b; Katz and Keyder 2012), and the use of the cost of the tractable relaxed problem as heuristic guidance for searching through the state space of the original problem (Hoffmann and Nebel 2001; Domshlak, Hoffmann, and Katz 2015; Keyder, Hoffmann, and Haslum 2012).

Meanwhile, the broader AI community in the last decade has seen a major surge in deep-learning-based approaches, driven by remarkable successes in computer vision, natural language processing, and reinforcement-based policy and value-function learning in Markov decision processes (Mnih et al. 2015) and adversarial games (Silver et al. 2016).

Deep reinforcement learning (RL) approaches, in particular, have several strengths, 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 a particular environment (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.

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, heuristic search methods in classical planning are not perfect. A great deal of effort has been spent on finding domain-independent heuristics, which provide substantial leverage for forward search. Although domain-independence is a welcome advantage, these heuristics can easily be vastly outperformed by carefully engineered domain-specific methods, such as a specialized solver for Sokoban (Junghanns and Schaeffer 2000). Developing such domain-specific heuristics can require intensive engineering effort, with payoff only in that single domain. We will be 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 quickly learn a domain-specific neural network value function. This value function improves over the domain-independent classical heuristics, and therefore can be used to plan more efficiently 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 improving the 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 proposed 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 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.

2 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 where the rest of the dimensions are same between $\boldsymbol{a}$ and $\boldsymbol{b}$. Functions (e.g., $\log, \exp$) are applied to arrays element-wise. Finally, we let $\mathbb{B}$ denote $[0, 1]$.

2.1 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 $s \subseteq P(O)$ represents truth assignments to the propositions, which can be represented as a bitvector of size $\sum_p O^{\#p}$. An action is a 3-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 $a^n_\dagger \circ \cdots \circ a^1_\dagger(I) \subseteq G$ and every action $a^i_\dagger$ satisfies its preconditions at the time of using it.

2.2 Markov Decision Processes

In general, RL methods address domains modeled as Markov decision processes (MDP), $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, r, q_0, \gamma)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $T(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{B}$ 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 $0 \leq \gamma < 1$ 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* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{B}$ 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.

Given an MDP, we are interested in finding a policy maximizing its *long term discounted cumulative rewards*, formally defined as a *value function*

$$V_{\gamma,\pi}(s) = \mathbb{E}_{A_t \sim \pi(S_t, \cdot)} \left[\sum_{t=0}^{\infty} \gamma^t R_t \,\middle|\, S_0 = s \right].$$

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 satisfy $V^*_\gamma(s) \geq V_{\gamma,\pi}(s)$ 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*.

Finally, we can define a policy by mapping action-values in each state to a probability distribution over actions. For example, using SOFTMAX gives us

$$\pi(s, a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q(s,a')/\tau}},$$

where $\tau \geq 0$ is a temperature that controls the greediness of the policy. It returns a greedy policy $\arg\max_a Q(s, a)$ when $\tau = 0$; and approaches a uniform policy when $\tau \to \infty$.

2.3 Formulating Classical Planning as an MDP

There are two typical ways to formulate a classical planning problem as an MDP. In one strategy, for any transition (s, a, s'), we assign a reward of 1 when $s' \in G$, and 0 otherwise. In the other, we 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. 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.

3 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 (e.g., due to unsatisfied preconditions) 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. Alternative formulations of preconditions in MDPs include one that augments the state space with an absorbing state with a highly negative reward.

(P2): Objective functions. Classical planning tries to minimize the sum of costs along trajectories, while the RL and the MDP frameworks may try to maximize the expected cumulative discounted rewards of trajectories. While the MDP framework does not necessarily assume discounting, the majority of RL applications use the discounted formulation (Schulman et al. 2015; Mnih et al. 2015, 2016; Lillicrap et al. 2016). Moreover, while costs could be treated as negative rewards in MDPs, discounting is unnatural in typical applications of AI Planning. Unlike those of modern RL where the target is the survival of an agent in a stochastic environment, applications of AI Planning are mainly cost minimization and pathfinding problems in fixed-budget scenarios with deterministic, fully-observable environments, where delaying a costly action does not benefit the agent.

(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 the size and the ordering, i.e., neural networks for a *set*-like input representation (Ravanbakhsh, Schneider, and Poczos 2016; Zaheer et al. 2017). Graph Neural Networks (Battaglia et al. 2018) have also been recently used to encode FOL inputs (Rivlin, Hazan, and Karpas 2020; Shen, Trevizan, and Thiébaux 2020; Ma et al. 2020). While the choice of the architecture is arbitrary, our network should be able to handle FOL inputs.

4 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. 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 omit the goal for notational concision in this discussion, but note that all of our value functions are goal-parameterized.

To learn estimated value functions, we build on the *value iteration* algorithm, where a known model of the dynamics is used to incrementally update estimates of the optimal value function and a corresponding optimal policy. In this section, we begin with a review of value iteration and describe a number of important details for scaling to problems with large state and action spaces. We then describe modifications to the generic method for application in our classical planning setting.

4.1 Backgrounds

Value iteration incrementally updates estimates of the optimal value function V^*_γ by updating its current estimates with the r.h.s. of Eq. 1 until a fixpoint:

1: **while** not converged **do**
2: **for** $s \in \mathcal{S}$ **do**
3: $V_{\gamma,\pi}(s) \leftarrow \max_{a \in \mathcal{A}} Q_{\gamma,\pi}(s,a)$. (*Bellman Update*)

where $V_{\gamma,\pi}$ and $Q_{\gamma,\pi}$ correspond to the estimated state and action values. In domains with small state spaces, the estimated value function can be represented with a table. In the classical planning domains that we consider in this work, state spaces are typically far too large for tables. The state spaces are so large, in fact, that enumerating states in an inner loop (Line 2) is impractical.

Real Time Dynamic Programming (RTDP) avoids the exhaustive enumeration of states in value iteration by sampling a subset of the state space based on the current policy. RTDP is a subclass of Asynchronous Dynamic Programming algorithm (Sutton and Barto 2018), which do not require every state to be updated after each iteration of the outer loop. RTDP can be summarized with the following pseudo-code:

1: **while** not converged **do**
2: $s \sim q_0, t \leftarrow 0$
3: **while** $t < D$ and s is a non-terminal state **do**
4: $a \leftarrow \arg\max_a Q_{\gamma,\pi}(s,a)$
5: $s \leftarrow T'(s,a)$
6: $V_{\gamma,\pi}(s) \leftarrow \max_{a \in \mathcal{A}} Q_{\gamma,\pi}(s,a)$. (*Bellman Update*)
7: $t \leftarrow t+1$

In this work, we use *on-policy* RTDP, which replaces the second $\max_a$ with $\mathbb{E}_a$ 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.

The other issue presented by large state spaces is that value estimates cannot be stored in an exhaustive table. We avoid this issue by encoding $V_{\gamma,\pi}$ using a neural network and applying the Bellman updates approximately. A single Bellman update is equivalent to a single Gradient Ascent step and can be made differentiable so that it is compatible with

neural networks. Let $\Delta V_{\gamma,\pi}(s)$ be the amount of the update made by a Bellman update. By substituting $x = V_{\gamma,\pi}(s)$ and treating $\max_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a)$ as a constant C,

$$\Delta V_{\gamma,\pi}(s) = \max_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a) - V_{\gamma,\pi}(s) = C - x.$$

Notice that $C - x = \frac{\partial}{\partial x}(-\frac{1}{2})(x-C)^2$. Therefore, a Bellman update is ascending a partial gradient of the maximization objective with regard to $V_{\gamma,\pi}(s)$:

$$-\frac{1}{2}\left(V_{\gamma,\pi}(s) - \max_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a)\right)^2.$$

This way of treating C as a constant is called *semi-gradient* or *bootstrapping* (Sutton and Barto 2018). In an implementation of V based on neural network libraries such as Tensorflow, we can simply pass a negation of this objective to a gradient-descent optimizer to implicitly perform an update in a "table" approximated by a neural network. It is important to stop the gradient for $\max_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a)$, which treats it as a constant during differentiation.

We use *experience replay* to help smooth out changes in the policy and reduce the correlation between updated states. We store the history of states into a large FIFO buffer B, and update using mini-batches sampled from the replay buffer in order to make use of efficient GPU-based parallelism. This technique has long been used to stabilize and accelerate neural net based RL methods (Lin 1993) and was popularized by Deep Q-learning (Mnih et al. 2015).

4.2 Modified RTDP for Classical Planning

We modify the training algorithm to address the assumptions **(P1)** in classical planning. First, 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, 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. This is also a requirement in the light of implementation since the input shape must be at least the same within a single mini-batch.

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

Third, since $\arg\max_a$ in RTDP is not possible at a state with no applicable actions (a.k.a. *deadlock*), the agent should reset the environment upon entering such a state. We also select actions only from applicable actions and do not treat an inapplicable action as a self-cycle. Indeed, training a value function along a trajectory that includes such a self-cycle has no benefit because the test-time agent (GBFS) never attempts to execute them. These modifications result in a variant of RTDP as follows:

Buffer $B \leftarrow [\emptyset, \emptyset, \emptyset, \ldots]$
while not converged **do**
  $\langle P, A, O, I, G\rangle \sim q_0$, $t \leftarrow 0$, $s \leftarrow I$,
  while $t < D$, $s \notin G$, s is not a deadlock **do**
    $a \leftarrow \arg\max_{a\in\{a|\textsc{pre}(a)\subseteq s\}} Q_{\gamma,\pi}(s,a)$
    $s \leftarrow T'(s,a)$
    $B[|O|].\textit{push}(s)$
    $\mathrm{SGD}(\frac{1}{2}(V_{\gamma,\pi}(s) - \mathbb{E}_{a\in\mathcal{A}} Q_{\gamma,\pi}(s,a))^2, B)$
    $t \leftarrow t+1$

5 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., $h^{\mathrm{FF}}, h^{\mathrm{add}}$) with a technique called *potential-based reward shaping*. To correctly perform this technique, we should take care of **(P2)** the difference between the discounted and non-discounted objectives.

Reward shaping (Ng, Harada, and Russell 1999) is a technique that helps the training of 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:

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

Under some mild assumptions, any function can be used as a potential function without affecting the optimal policies under the original reward function (Ng, Harada, and Russell 1999). Furthermore, the optimal value function $\hat{V}^*_\gamma$ of the modified MDP $\hat{\mathcal{M}} = (\mathcal{S}, \mathcal{A}, T, \hat{r}, q_0)$ satisfies

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

In other words, an agent trained in $\hat{\mathcal{M}}$ is learning the 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 (Wiewiora 2003).

Building on these theoretical backgrounds, 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 the improvement of the heuristic value 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., $h : \mathcal{S} \to \mathbb{R}^{+0} \cup \{\infty\}$. In such cases, gradient-based optimization no longer works due to numerical issues. **(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:

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

This indicates that the amount of correction learned (i.e., $\hat{V}^*_\gamma(s) = V^*_\gamma(s) - \phi(s)$) is not zero even in an ideal scenario of $\phi = -h = -h^*$. This issue is a direct consequence of not properly accounting for discounting.

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 achieve a goal, we tend to treat the value as an opaque number. 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*. In unit-cost domains, we regard a heuristic value $h(s)$ as a non-discounted cost-to-go, and thus define a corresponding *discounted heuristic function* $h_\gamma(s)$ as:

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

Notice that $\phi = -h_\gamma = -h^*_\gamma$ results in $\hat{V}^*_\gamma(s) = 0$. This is also beneficial from the practical standpoint: The weights of neural networks, including those used for representing $\hat{V}_\gamma$, are typically randomly initialized so that the expected value of the output is zero (Glorot and Bengio 2010). Also, the resulting function is bounded by $0 \leq h_\gamma(s) \leq 1/(1-\gamma)$, avoiding issues resulting from infinite heuristic values.

6 Value-function Generalized over Size

To achieve the goal of learning domain-dependent heuristics specialized over the tasks of the same domain, the neural value function used in the reward-shaping framework discussed above must be invariant to the number and the order of propositions and objects in a PDDL definition (point **(P3)**). To address this issue, we propose the use of Neural Logic Machine (Dong et al. 2019, NLM), an architecture originally designed for a supervised learning task over FOL inputs. We first discuss Multi-Arity Predicate Representation (MAPR), an array-based representation of grounded FOL inputs that NLM uses.

6.1 Multi-Arity Predicate Representation

Assume that we need to represent FOL statements combining predicates of different arities. We denote a set of predicates of arity n as P/n (Prolog notation), its propositions as $P/n(O)$, and the Boolean tensor representation of $P/n(O)$ as $\boldsymbol{z}/n \in \mathbb{B}^{O^n \times |P/n|}$. A MAPR is a tuple of N tensors $\boldsymbol{z} = (\boldsymbol{z}/1, \ldots, \boldsymbol{z}/N)$ where N is the largest arity. 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 $\boldsymbol{z}/2 \in \mathbb{B}^{3\times3\times4}$. Similarly, we may have $\boldsymbol{z}/1 \in \mathbb{B}^{3\times2}$ for 2 unary predicates, and $\boldsymbol{z}/3 \in \mathbb{B}^{3\times3\times3\times5}$ for 5 ternary predicates.

6.2 Neural Logic Machines

The NLM (Dong et al. 2019) is a neural Inductive Logic Programming (ILP) (Muggleton 1991) system based on FOL and the Closed-World Assumption (Reiter 1981). NLM represents a set of continuous relaxations of Horn rules as a set of weights in a neural network and is able to infer the truthfulness of some target formulae as a probability. For example, in Blocksworld, based on an input such as on(a, b) for blocks a, b, NLMs may be trained to predict clear(b) is true by learning a quantified formula $\neg\exists x; \mathsf{on}(x, \mathsf{b})$.

NLM takes a boolean MAPR of propositional groundings of FOL statements. NLM is 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. Under these assumptions, the following set of rules are sufficient for representing any rules (Dong et al. 2019):

$$\begin{aligned}
&\text{(expand)} && \forall x_{\#\overline{p_k}}; \overline{p_k}(X; x_{\#\overline{p_k}}) \leftarrow p_k(X), \\
&\text{(reduce)} && \underline{p_k}(X) \leftarrow \exists x_{\#p_k}; p_k(X; x_{\#p_k}), \\
&\text{(compose)} && p_{k+1}(X) \leftarrow \\
& && \mathcal{F}\left(\bigcup_\pi \left(\left(P_k \cup \underline{P_k} \cup \overline{P_k}\right)/\#p_{k+1}\right)(\pi(X))\right).
\end{aligned}$$

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 these 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 (Fig. 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 $O^{n-1} \times |P/n|$. The reduce operation can also use $\min$, in which case $\exists$ becomes $\forall$.

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

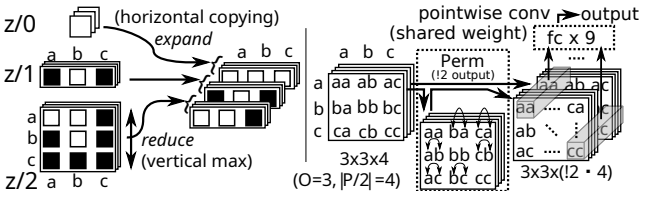

Figure 1: **(Left)** EXPAND and REDUCE operations performed on a boolean MAPR containing nullary, unary, and binary predicates and three objects, a, b, and c. Each white / black square represents a boolean value (true / false). **(Right)** PERM tensor operation performed on binary predicates. They are generated by performing the same operations shown on the left side of the figure on $\boldsymbol{z}/1$, $\boldsymbol{z}/2$, and $\boldsymbol{z}/3$. Each predicate is represented as a matrix. For a matrix, PERM is equivalent to concatenating the matrix with its transposition. When PERM is applied to ternary predicates, it concatenates $!3 = 6$ tensors. After PERM, a single, shared fully-connected layer is applied to each combination of arguments (such an operation is sometimes called a pointwise convolution).

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 we denote as $\text{COMPOSE}(\boldsymbol{z}, n, Q, \sigma)$. 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.

6.3 Value Functions as Neural Logic Machines

The network that represents a value function consists of NLM layers. 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 the value function is a scalar (arity 0). 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 $N \le M \le L$.

Intermediate layers have a sigmoid activation function, while the output is linear, since 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. In the experiments, we typically use $L = 4$ or $L = 6$ layers.

7 Searching with a Learned Value Function

Finally, we describe the test-time agent based on greedy best first search (GBFS) (Hoffmann and Nebel 2001). Since the learned value function $\hat{V}_\gamma(s)$ is a correction to the potential function (Eq. 3), the true value function which represents a discounted cumulative expected reward is

$$V_\gamma(s) = \hat{V}_\gamma(s) + (-h_\gamma(s)). \tag{6}$$

Since GBFS deals with costs rather than rewards, our "heuristic function" is $-V_\gamma(s) = h_\gamma(s) - \hat{V}_\gamma(s)$.

Practically speaking, we could use this $-V_\gamma(s)$ directly as a heuristic function in GBFS. However, this "heuristic function" is theoretically unappealing, especially if we consider using it in algorithms such as Weighted A^* (Pohl 1973). Recall that $-V_\gamma(s)$ is an approximation of optimal discounted cumulative cost. Due to its discounted nature, it is hard to justify adding its value to other distance-based metrics such as g-values in Weighted A^*, which are not discounted. To address this issue, we propose the following *undiscount* operation, which obtains an undiscounted heuristic value from a discounted value function:

$$V_\gamma(s) = -\frac{1 - \gamma^{h(s)}}{1 - \gamma} \tag{7}$$

$$\therefore h(s) = \log_\gamma\left((1-\gamma)V_\gamma(s) + 1\right) \in \mathbb{R}. \tag{8}$$

The idea behind this transformation is to smoothly interpolate a step-wise summation $\sum_{t=1}^{h(s)} \gamma^t \cdot (-1)$ as if $h(s)$ is a continuous value.

Unfortunately, this operation is valid only in a unit-cost domain. Also, in GBFS we evaluate later, whether using $-V_\gamma(s)$ or $h(s)$ does not affect the order of expansions because this operation is monotonic. Future work includes extending this formulation to a non-unit cost domain and evaluating it in Weighted A^*.

8 Experimental Evaluation

All experiments are performed on a distributed compute cluster equipped with Xeon E5-2600 v4 and Tesla K80, which is three generations older than the flagship accelerators at the time of writing. Our implementation combines the `jax` auto-differentiation framework for neural networks (Bradbury et al. 2018), the PDDLGym library (Silver and Chitnis 2020) for parsing, and `pyperplan` to obtain the heuristic value of h^{FF} and h^{add}. The resulting program supports pure STRIPS with unit-cost actions. We verified

that this program works on 25 domains from past IPCs without raising errors by removing `:action-cost` specifications from the domain files. We include the list of domains in the appendix (Sec. A.3).

While our program is compatible with this wide range of domains, we focus on extensively testing a selected subset of domains with a large enough number of independently trained models (20), to feel confident in comparing the results, due to the fact that RL algorithms tend to have a large amount of variance in their outcomes (Henderson et al. 2018), induced by sensitivity to initialization, randomization in exploration, and randomization in experience replay.

Our objective is to see whether the our RL agent can improve the efficiency of GBFS, over a standard domain-independent heuristic, measured in terms of the number of node-evaluations performed during the search. In addition, we place an emphasis on generalization ability: we hope that by using NLMs to represent the value function, we will be able to generalize from training on problem instances with a small number of objects (which makes computation time more feasible) to executing on domains with much larger numbers of objects.

We trained our system on five classical planning domains used in (Rivlin, Hazan, and Karpas 2020): 4-ops blocksworld, ferry, gripper, logistics, satellite, as well as three more domains from past IPCs, miconic, parking, visitall, that are supported by our implementation. In all domains, we generated a number of problem instances with parameterized generators used in the past IPCs [1]. Table 2 in the appendix shows the list of parameters given to the generator. For each domain, we provided from 195 to 500 instances for training, and from 250 to 700 instances for testing. Each agent is trained for 50000 steps, which takes about 4 to 6 hours. The list of remaining hyperparameters are available on the appendix (Sec. A.2).

After the training, we ran a GBFS-based planner which uses the learned heuristics on the test instances. We limited the maximum node evaluations to 100000 without applying a time limit or a memory limit. Let us denote the baseline GBFS with heuristics $h \in \{h^{\text{blind}}, h^{\text{FF}}, h^{\text{add}}\}$ as $\text{GBFS}(h)$. We denote a heuristic function obtained by training an RL agent with reward shaping $\phi = -h_\gamma$ with a capital letter, e.g., H^{FF} is a heuristic function obtained by h^{FF}_γ-based reward shaping.

In this experiment, we aim to answer the following questions: **(Q1)** Do our RL agents learn heuristic functions at all, i.e., $\text{GBFS}(h^{\text{blind}}) < \text{GBFS}(H^{\text{blind}})$? **(Q2.1)** Do our RL agents with reward shaping outperform our RL agents without shaping, i.e., $\text{GBFS}(H^{\text{blind}}) < \text{GBFS}(H^{\text{FF}})$? **(Q2.2)** Can they improve the performance over the baseline heuristics it was initialized to, i.e., $\text{GBFS}(h^{\text{FF}}) < \text{GBFS}(H^{\text{FF}})$? **(Q3)** Does the heuristics obtained by our framework maintain its improvement in larger problem instances, i.e., is it generalized over the number of objects? **(Q4)** Can the improvement be explained by accelerated exploration?

We compared the node evaluations between the baseline $(h^{\text{blind}}, h^{\text{FF}}, h^{\text{add}})$ and the learned heuristics $(H^{\text{blind}}, H^{\text{FF}}, H^{\text{add}})$ for instances solved by either one. Fig. 2 answers the first question **(Q1)**. We compared the result of $\text{GBFS}(h^{\text{blind}})$ and $\text{GBFS}(H^{\text{blind}})$, i.e., Breadth-first search, which is equivalent to GBFS with blind heuristics (heuristic value is constantly 0), against GBFS with a heuristic function without reward shaping. The aim of this experiment is to test if a baseline RL (without reward shaping) learns a useful heuristic function at all. The result was positive: Aside from visitall and miconic, Breadths-first search failed to solve any instance in the test set we provided, while a GBFS using the learned heuristics managed to solve some instances depending on the domain.

We next similarly compare the node evaluations between $\text{GBFS}(h^{\text{FF}})$ and $\text{GBFS}(H^{\text{FF}})$, as well as $\text{GBFS}(h^{\text{add}})$ and $\text{GBFS}(H^{\text{add}})$ **(Q2.2)**. The plot suggests that the reward-shaping-based training has successfully improved upon the baseline heuristics. However, the effect is negative or neutral on gripper and logistics where heuristics are presumably already accurate. In such cases, there is little room to improve upon the baseline, and thus the high randomness in the reinforcement learning may potentially harm the performance.

To show the effectiveness of NLM in generalizing the policy with regard to the number of objects in the environment **(Q3)**, we checked the improvements with regard to the problem size. Fig. 2 (Right) plots the number of objects in the x-axis and the ratio of success over problem instances in the y-axis. It shows that the heuristic accuracy is improved in instances whose size far exceeds the training instances for $h^{\text{blind}}, h^{\text{FF}}, h^{\text{add}}$. Due to space, full results covering the eight domains tested are included in the appendix.

Fig. 2 also answers **(Q2.1)**: $\text{GBFS}(H^{\text{FF}})$ and $\text{GBFS}(H^{\text{add}})$ outperforms $\text{GBFS}(H^{\text{blind}})$. The plots seem to suggest that the performance of the base heuristics used for reward shaping affects the quality of resulting learned heuristics. This matches the theoretical expectation that the potential function is a domain knowledge that initializes the policy.

Finally, we evaluated the effect of reward shaping on the exploration during the training **(Q4)**. Table 1 shows the total number of goals reached by the agent, which indicates that reward shaping indeed helps the agent reach goals more often. (See Appendix Fig. 4-5 for cumulative plots.)

9 Related Work

Approaches that try to improve classical planning performance through learning has been primarily focused on learning domain control knowledge including macro actions (Korf 1985; Coles and Smith 2004; Botea et al. 2005; García-Durán, Fernández, and Borrajo 2006; Newton et al. 2007; Jonsson 2007; Asai and Fukunaga 2015; Chrpa and Siddiqui 2015), HTN methods (Nejati, Langley, and Konik 2006; Hogg, Muñoz-Avila, and Kuter 2008; Hogg, Kuter, and Munoz-Avila 2010), temporal rules (Bacchus and Kabanza 2000), as well as and portfolio optimization (Cenamor, de la Rosa, and Fernández 2014; Sievers et al. 2019).

Early attempts to learn heuristic functions include applying shallow, fully connected neural networks to puzzle domains (Arfaee, Zilles, and Holte 2010, 2011), its online version (Thayer, Dionne, and Ruml 2011), combining SVMs (Cortes and Vapnik 1995) and NNs (Satzger and

[1] `github.com/AI-Planning/pddl-generators`

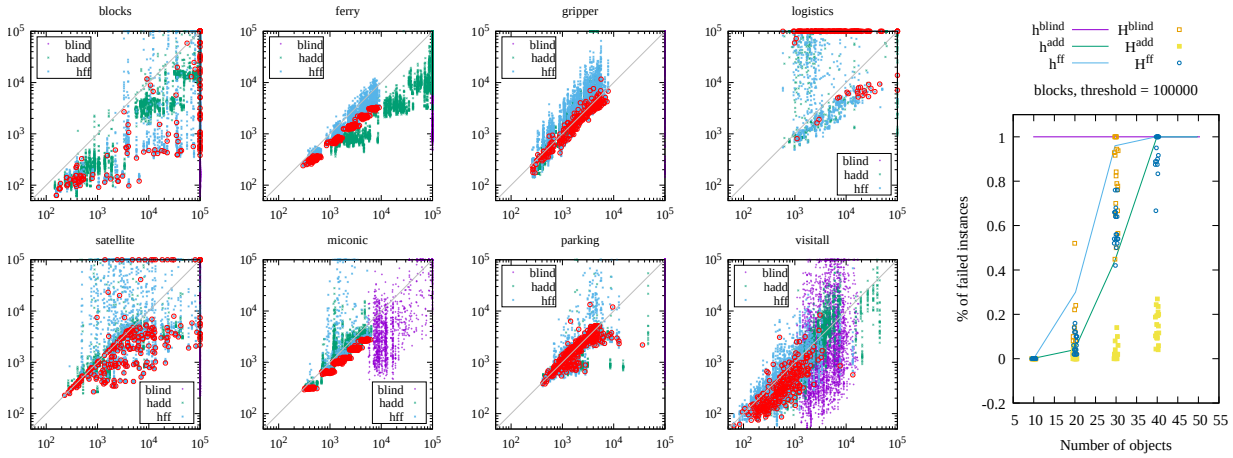

Figure 2: **(Left)** Scatter plot showing the number of node evaluations on 8 domains, where x-axis is for GBFS with $h^{\text{blind}}, h^{\text{FF}}, h^{\text{add}}$ and y-axis is for $H^{\text{blind}}, H^{\text{FF}}, H^{\text{add}}$. Each point corresponds to a single test problem instance. For the learned heuristics, results of 20 seeds are plotted against a single baseline. Failed instances are plotted on the border. Points below the diagonal indicate the instances which were improved by reinforcement learning. In addition, **red circles** highlight the best seed from H^{FF} whose sum of evaluations across instances is the smallest. **(Right)** The rate of successfully finding a solution for a set of problems (y-axis) with a given number of objects (x-axis). Full results are available in the appendix.

domain	h^{blind}	h^{add}	h^{FF}
blocks	4691 ± 100	4808 ± 90	**5089** ± 70
ferry	4981 ± 178	**5598** ± 45	5530 ± 68
gripper	2456 ± 185	**3856** ± 39	3482 ± 122
logistics	3475 ± 214	**5059** ± 151	**5046** ± 131
miconic	3568 ± 25	**3794** ± 21	**3808** ± 24
parking	3469 ± 496	**4763** ± 78	**4716** ± 54
satellite	3292 ± 195	**4388** ± 78	**4387** ± 52
visitall	1512 ± 88	1360 ± 73	**2063** ± 51

Table 1: The cumulative number of goal states the RTDP has reached during training. The numbers are average and standard deviation over 20 seeds. Best heuristics are highlighted in bold. In logistics, miconic, parking, and satellite, we highlight both h^{add} and h^{FF} because the difference between h^{add} and h^{FF} are not statistically significant under Wilcoxon's rank-sum test ($p = 0.37, 0.02, 0.007, 0.43$), while both significantly outperforms h^{blind} ($p < 0.0005$).

Kramer 2013), learning a residual from planning heuristics similar to ours (Yoon, Fern, and Givan 2006, 2008), or a relative ranking between states instead of absolute values (Garrett, Kaelbling, and Lozano-Pérez 2016). Ferber, Helmert, and Hoffmann (2020) tested fully-connected layers in modern frameworks. ASNet (Toyer et al. 2018) learns domain-dependent heuristics using a network that is similar to Graph Neural Networks (GNN) (Battaglia et al. 2018). STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020) learns domain-independent heuristics using hypergraph networks which generalizes GNN and is capable of encoding delete-relaxation. Their disadvantage is their supervised nature, which requires optimal costs of each state in the dataset.

Grounds and Kudenko (2005) combined reinforcement learning and STRIPS planning with reward shaping, but their setting is different from our scenario: They treat a 2D navigation problem as a two-tier hierarchical planning 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. Unlike their approach, our study focuses on training an RL agent for solving the high-level PDDL task. The separate low-level task as considered by their paper does not exist in our scenario.

Rivlin, Hazan, and Karpas (2020) implements an RL agent based on Proximal Policy Optimization (Schulman et al. 2017) which represents its policy function as a Graph Neural Networks (Scarselli et al. 2009, GNNs). While their method includes a technique to derive a value function from a policy function, they modify it in an ad-hoc manner with an entropy to use it as a heuristic function for GBFS.

10 Conclusion

In this paper, we proposed a domain-independent reinforcement learning framework for learning domain-specific heuristic functions. Unlike existing work on applying policy gradient to planning (Rivlin, Hazan, and Karpas 2020), we based our algorithm on value iteration whose training results become a proper heuristic function whose unit of measure is a usual, non-discounted cost in planning. We addressed the difficulty of training an RL agent with sparse rewards using a novel reward-shaping technique which leverages existing heuristics developed in the literature. We showed that our framework not only learns a heuristic function from scratch, but also learns better if aided by heuristic functions (reward shaping). Furthermore, the learned heuristics keeps outperforming the baseline over a wide range of problem sizes, demonstrating its generalization over the number of objects in the environment.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap Learning of Heuristic Functions. In Felner, A.; and Sturtevant, N. R., eds., *Proc. of Annual Symposium on Combinatorial Search*. AAAI Press.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artificial Intelligence* 175(16-17): 2075–2098. doi:10.1016/j.artint.2011.08.001.

Asai, M.; and Fukunaga, A. 2015. Solving Large-Scale Planning Problems by Decomposition and Macro Generation. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*. Jerusalem, Israel.

Bacchus, F.; and Kabanza, F. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116(1): 123–191.

Bäckström, C.; and Klein, I. 1991. Planning in polynomial time: the SAS-PUBS class. *Computational Intelligence* 7(3): 181–197.

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 *Proc. of the International Conference on Machine Learning*, 507–517. PMLR.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* .

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *J. Artif. Intell. Res.(JAIR)* 24: 581–621.

Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M. J.; Leary, C.; Maclaurin, D.; Necula, G.; Paszke, A.; VanderPlas, J.; Wanderman-Milne, S.; and Zhang, Q. 2018. JAX: composable transformations of Python+NumPy programs.

Brafman, R. I.; and Domshlak, C. 2003. Structure and Complexity in Planning with Unary Operators. *J. Artif. Intell. Res.(JAIR)* 18: 315–349.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69(1–2): 165–204.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2014. IBACOP and IBACOP2 planner. In *Proc. of the International Planning Competition*.

Chrpa, L.; and Siddiqui, F. H. 2015. Exploiting Block Deordering for Improving Planners Efficiency. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*.

Coles, A.; and Smith, A. 2004. Marvin: Macro Actions from Reduced Versions of the Instance. In *Proc. of the International Planning Competition*. Http://www.tzi.de/ edelkamp/ipc-4/IPC-4.pdf.

Cortes, C.; and Vapnik, V. 1995. Support-Vector Networks. *Machine learning* 20(3): 273–297.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-Black Planning: A New Systematic Approach to Partial Delete Relaxation. *Artificial Intelligence* 221: 73–114.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines. In *Proc. of the International Conference on Learning Representations*.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, Decidability and Undecidability Results for Domain-Independent Planning. *Artificial Intelligence* 76(1–2): 75–88.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Proc. of European Conference on Artificial Intelligence*, 2346–2353.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* 84(1-2): 81–107.

Fikes, R. E.; and Nilsson, N. J. 1972. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3): 189–208.

García-Durán, R.; Fernández, F.; and Borrajo, D. 2006. Combining Macro-operators with Control Knowledge. In *Proc. of International Conference on Inductive Logic Programming (ILP)*. Santiago de Compostela, Spain. doi:10.1007/978-3-540-73847-3
25.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 3089–3095.

Glorot, X.; and Bengio, Y. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249–256. JMLR Workshop and Conference Proceedings.

Grounds, M.; and Kudenko, D. 2005. Combining Reinforcement Learning with Symbolic Planning. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, 75–86. Springer.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.(JAIR)* 26: 191–246.

Henderson, P.; Islam, R.; Bachman, P.; Pineau, J.; Precup, D.; and Meger, D. 2018. Deep reinforcement learning that matters. In *Proc. of AAAI Conference on Artificial Intelligence*, volume 32.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res.(JAIR)* 14: 253–302. doi:10.1613/jair.855.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *Proc. of AAAI Conference on Artificial Intelligence*, volume 24.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proc. of AAAI Conference on Artificial Intelligence*.

Jonsson, A. 2007. The Role of Macros in Tractable Planning over Causal Graphs. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*.

Jonsson, P.; and Bäckström, C. 1998a. State-Variable Planning under Structural Restrictions: Algorithms and Complexity. *Artificial Intelligence* 100(1–2): 125–176.

Jonsson, P.; and Bäckström, C. 1998b. Tractable Plan Existence Does Not Imply Tractable Plan Generation 22(3,4): 281–296.

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. *Artificial Intelligence* .

Katz, M.; and Domshlak, C. 2008a. New Islands of Tractability of Cost-Optimal Planning. *J. Artif. Intell. Res.(JAIR)* 32: 203–288.

Katz, M.; and Domshlak, C. 2008b. Structural Patterns Heuristics via Fork Decomposition. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, 182–189.

Katz, M.; and Keyder, E. 2012. Structural Patterns Beyond Forks: Extending the Complexity Boundaries of Classical Planning. In *Proc. of AAAI Conference on Artificial Intelligence*, 1779–1785.

Keyder, E.; Hoffmann, J.; and Haslum, P. 2012. Semi-Relaxed Plan Heuristics. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, 128–136.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. 282–293. Springer.

Korf, R. E. 1985. Macro-Operators: A Weak Method for Learning. *J. Artif. Intell. Res.(JAIR)* 26(1): 35–77. doi:10.1016/0004-3702(85)90012-8.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous Control with Deep Reinforcement Learning. In *Proc. of the International Conference on Learning Representations*.

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 *Proc. of AAAI Conference on Artificial Intelligence*, volume 34, 5077–5084.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proc. of the International Conference on Machine Learning*, 1928–1937. PMLR.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-Level Control through Deep Reinforcement Learning. *Nature* 518(7540): 529–533.

Muggleton, S. 1991. Inductive Logic Programming. *New generation computing* 8(4): 295–318.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning Hierarchical Task Networks by Observation. In *Proc. of the International Conference on Machine Learning*.

Newton, M. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proc. of the International Conference on Machine Learning*, volume 99, 278–287.

Pohl, I. 1973. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*.

Ravanbakhsh, S.; Schneider, J.; and Poczos, B. 2016. Deep Learning with Sets and Point Clouds. *arXiv preprint arXiv:1611.04500* .

Reiter, R. 1981. On Closed World Data Bases. In *Readings in Artificial Intelligence*, 119–140. Elsevier.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning.

Satzger, B.; and Kramer, O. 2013. Goal Distance Estimation for Automated Planning using Neural Networks and Support Vector Machines. *Natural Computing* 12(1): 87–100. doi:10.1007/s11047-012-9332-y.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20(1): 61–80.

Schaul, T.; Horgan, D.; Gregor, K.; and Silver, D. 2015. Universal value function approximators. In *International conference on machine learning*, 1312–1320. PMLR.

Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.; and Moritz, P. 2015. Trust Region Policy Optimization. In *Proc. of the International Conference on Machine Learning*, 1889–1897. PMLR.

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

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, 574–584.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *Proc. of AAAI Conference on Artificial Intelligence*, volume 33, 7715–7723.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529(7587): 484–489.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems.

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 *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 21.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proc. of AAAI Conference on Artificial Intelligence*, volume 32.

Wiewiora, E. 2003. Potential-based shaping and Q-value initialization are equivalent. *J. Artif. Intell. Res.(JAIR)* 19: 205–208.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *Journal of Machine Learning Research* 9(4).

Yoon, S. W.; Fern, A.; and Givan, R. 2006. Learning Heuristic Functions from Relaxed Plans. In *ICAPS*, volume 2, 3.

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep Sets. In *Advances in Neural Information Processing Systems*, 3391–3401.

A Appendix

A.1 Generator Parameters

Table 2 contains a list of parameters used to generate the training and testing instances. Since generators have a tendency to create an identical instance especially in smaller parameters, we removed the duplicates by checking the md5 hash value of each file.

Domain	Parameters	$\|O\|$
blocks/train/	2-6 blocks x 50 seeds	2-6
blocks/test/	10,20,..,50 blocks x 50 seeds	10-50
ferry/train/	2-6 locations x 2-6 cars x 50 seeds	4-7
ferry/test/	10,15,...30 locations and cars x 50 seeds	20-60
gripper/train/	2,4...,10 balls x 50 seeds (initial/goal locations are randomized)	6-14
gripper/test/	20,40,...,60 balls x 50 seeds (initial/goal locations are randomized)	24-64
logistics/train/	1-3 airplanes x 1-3 cities x 1-3 city size x 1-3 packages x 10 seeds	5-13
logistics/test/	4-8 airplanes/cities/city size/packages x 50 seeds	32-96
satellite/train/	1-3 satellites x 1-3 instruments x 1-3 modes x 1-3 targets x 1-3 observations	15-39
satellite/test/	4-8 satellites/instruments/modes/targets/observations x 50 seeds	69-246
miconic/train/	2-4 floors x 2-4 passengers x 50 seeds	8-12
miconic/test/	10,20,30 floors x 10,20,30 passengers x 50 seeds	24-64
parking/train/	2-6 curbs x 2-6 cars x 50 seeds	8-16
parking/test/	10,15,..,25 curbs x 10,15,..25 cars x 50 seeds	24-54
visitall/train/	For $n \in 3..5$, nxn grids, 0.5 or 1.0 goal ratio, n blocked locations, 50 seeds	8-22
visitall/test/	For $n \in 6..8$, nxn grids, 0.5 or 1.0 goal ratio, n blocked locations, 50 seeds	32-58

Table 2: List of parameters used for generating the training and testing instances.

A.2 Hyperparameters

We trained our network with a following set of hyperparameters: Maximum episode length $D = 40$, Learning rate 0.001, discount rate $\gamma = 0.999999$, maximum intermediate arity $M = 3$, number of layers $L = 4$ in satellite and logistics, while $L = 6$ in all other domains, the number of features in each NLM layer $Q = 8$, batch size 25, temperature $\tau = 1.0$ for a policy function (Sec. 2.2), and the total number of SGD steps to 50000, which determines the length of the training. We used $L = 4$ for those two domains to address GPU memory usage: Due to the size of the intermediate layer $O^n \times (n! \cdot C)$, NLM sometimes requires a large amount of GPU memory. Each training takes about 4 to 6 hours, depending on the domain.

A.3 Preliminary Results on Compatible Domains

We performed a preliminary test on a variety of IPC classical domains that are supported by our implementation. The following domains worked without errors: barman-opt11-strips, blocks, depot, driverlog, elevators-opt11+sat11-strips, ferry, floortile-opt11-strips, freecell, gripper, hanoi, logistics00, miconic, mystery, nomystery-opt11-strips, parking-opt11+sat11-strips, pegsol-opt11-strips, pipesworld-notankage, pipesworld-tankage, rovers, satellite, scanalyzer-08-strips, sokoban-opt11-strips, tpp, transport-opt11+sat08-strips, visitall-opt11-strips, zenotravel.

A.4 Full Results

Fig. 3 contains the full results of Fig. 2 (Right).

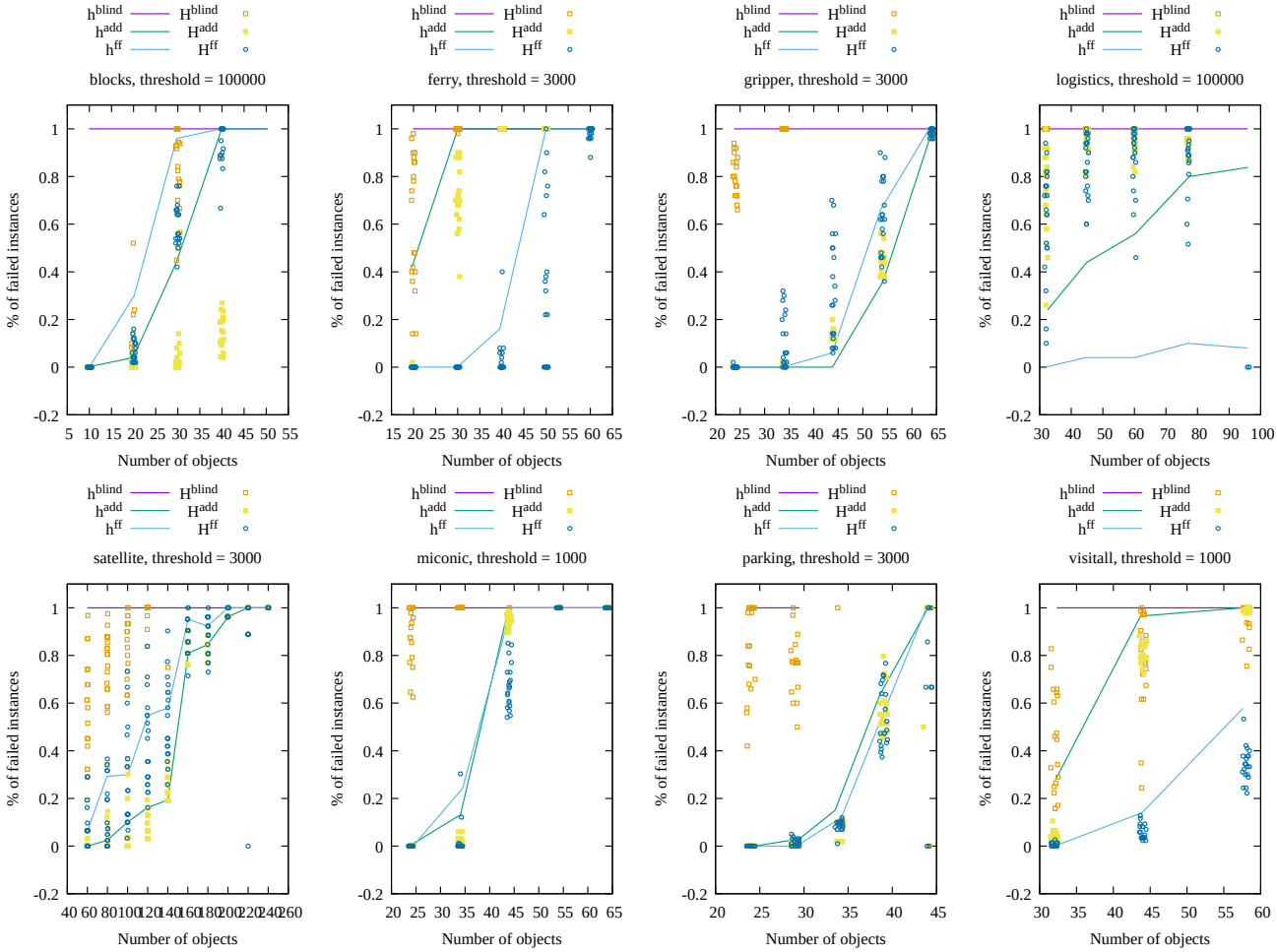

Figure 3: The rate of successfully finding a solution (y-axis) for instances with a certain number of objects (x-axis). Learned heuristic functions outperform their original baselines used for reward shaping in most domains. Since the initial maximum node evaluation is too permissive, we manually set a threshold for the number of node evaluations for each domain and filtered the instances when the node evaluation exceeded this threshold. This filtering emphasizes the difference because both the learned and the baseline variants may have solved all instances.

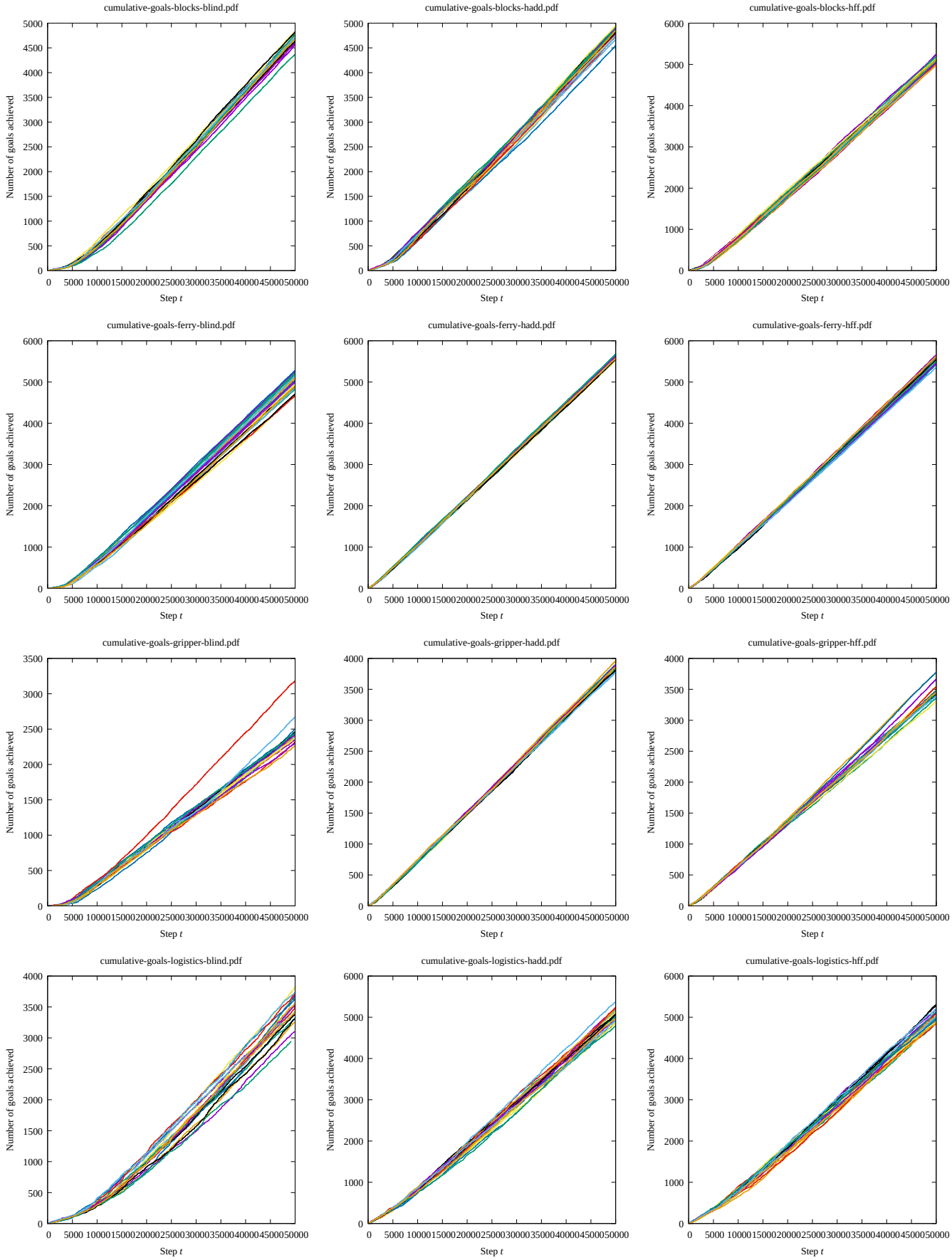

Figure 4: Cumulative number of instances that are solved during the training, where x-axis is the training step (part 1). Note that this may include solving the same instance multiple times.

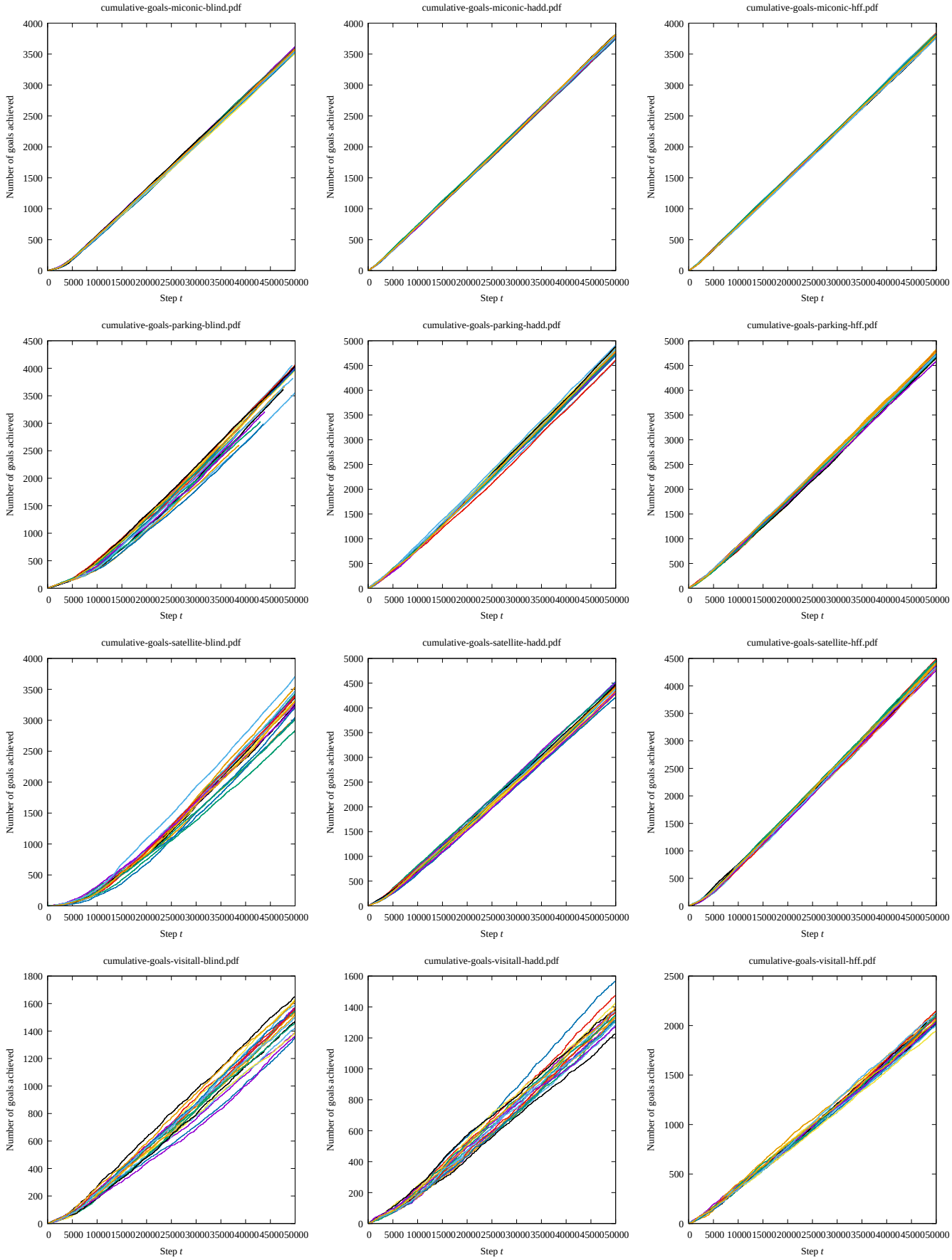

Figure 5: Cumulative number of instances that are solved during the training, where x-axis is the training step (part 2). Note that this may include solving the same instance multiple times.