Learning General Policies with Policy Gradient Methods

Simon Ståhlberg[1] , **Blai Bonet**[2] , **Hector Geffner**[3,1]
[1]Linköping University, Sweden
[2]Universitat Pompeu Fabra, Spain
[3]RWTH Aachen University, Germany
simon.stahlberg@liu.com, bonetblai@gmail.com, hector.geffner@ml.rwth-aachen.de

Abstract

While reinforcement learning methods have delivered remarkable results in a number of settings, generalization, i.e., the ability to produce policies that generalize in a reliable and systematic way, has remained a challenge. The problem of generalization has been addressed formally in classical planning where provable correct policies that generalize over all instances of a given domain have been learned using combinatorial methods. The aim of this work is to bring these two research threads together to illuminate the conditions under which (deep) reinforcement learning approaches, and in particular, policy optimization methods, can be used to learn policies that generalize like combinatorial methods do. We draw on lessons learned from previous combinatorial and deep learning approaches, and extend them in a convenient way. From the former, we model policies as state transition classifiers, as (ground) actions are not general and change from instance to instance. From the latter, we use graph neural networks (GNNs) adapted to deal with relational structures for representing value functions over planning states, and in our case, policies. With these ingredients in place, we find that actor-critic methods can be used to learn policies that generalize almost as well as those obtained using combinatorial approaches while avoiding the scalability bottleneck and the use of feature pools. Moreover, the limitations of the DRL methods on the benchmarks considered have little to do with deep learning or reinforcement learning, and result from the well-understood expressive limitations of GNNs, and the tradeoff between optimality and generalization (general policies cannot be optimal in some domains). Both of these limitations are addressed without changing the basic DRL methods by adding derived predicates and an alternative cost structure to optimize.

1 Introduction

Reinforcement learning (RL) has delivered remarkable results in a number of settings like game playing and robotics (Mnih et al. 2015; Levine et al. 2016; Silver et al. 2018). However, generalization, the ability to produce policies that generalize in a reliable and systematic way, remains a challenge (Kirk et al. 2023). A basic question, for example, is whether RL methods can be used to learn policies that generalize to any instance of a classical planning domain like Blockworld; namely, instances that feature an arbitrary number of objects that must be mapped into an arbitrary goal configuration. The power of deep learning for delivering such policies has been explored in a number of works (Toyer et al. 2020; Garg, Bajpai, and Mausam 2020; Rivlin, Hazan, and Karpas 2020); yet these approaches use supervised or model-based methods, or do not yield in general nearly perfect policies.

The problem of learning general policies has been addressed formally in the KR and planning setting (Srivastava, Immerman, and Zilberstein 2008; Hu and De Giacomo 2011; Bonet and Geffner 2015; Belle and Levesque 2016; Bonet et al. 2017; Illanes and McIlraith 2019) and some logical approaches appealing to combinatorial optimization methods have been used to learn provable correct policies that generalize over a number of classical planning domains (Francès, Bonet, and Geffner 2021; Drexler, Seipp, and Geffner 2022). Roughly, the (unsupervised) learning problem is cast as the joint problem of selecting state features f from a pool of features $\mathcal{F}$ and classify state transitions (s, s') into "good" and "bad", such that good transitions lead to the goal with no cycles, and good and bad transitions can be distinguished by their effects on the selected features. The use of state transitions (s, s') to select actions, rather than selecting the actions directly, is because the set of (ground) actions changes with the set of objects, while the set of predicates, and hence, the features derived from them, remain fixed (Martín and Geffner 2004).

The RL and planning approaches for learning general policies have different strengths and weaknesses. The deep RL (DRL) approach does not offer a meaningful formal language to encode general policies, nor a meaningful metalanguage to study them, but its strengths are that it does not require symbolic states (e.g., it can deal with states represented by pixels), and it does not have a scalability bottleneck (stochastic gradient descent is efficient and works surprisingly well). On the other hand, the planning approach offers meaningful languages for representating instances and domains, and policies and states, while its shortcomings are its reliance on human-provided domain encodings, and the computational bottleneck of the combinatorial optimization methods: training instances must be large enough so that the resulting policies generalize, and small enough so that the solvers can solve them optimally.

Works that combine planning languages for representing states, and deep networks for encoding either value and policy functions have brought the DRL and planning approaches closer. Ståhlberg, Bonet, and Geffner (2022a) use a graph neural network (GNN) for mapping planning states s into a value function $V(s)$ that is trained, in a supervised way, to approximate the optimal value function $V^*(s)$ that measures the minimum number of steps from s to the goal. The learned value function V encodes the greedy policy π_V that selects successor states s' with lowest $V(s')$ value. More recently, the same neural architecture has been used to learn a value function V without supervision by minimizing the Bellman errors $|V(s) - \min_{s'}[1 + V(s')]|$, where s' ranges over the possible successors of state s and the cost of actions is assumed to be 1 (Ståhlberg, Bonet, and Geffner 2022b). The resulting method is a form of *asynchronous value iteration* where the value function V is encoded by a deep net and the error is minimized at selected states s by stochastic gradient descent.

These works, however, do not answer the question of *whether RL methods can be used to learn policies that generalize in a systematic manner.* We are particularly interested in *policy gradient* approaches (Williams and Peng 1991; Sutton and Barto 2018) because their broader scope, unlike value-based methods; they can be used 1) when the state and action spaces are continuous, 2) when the system state is not fully observable, and also 3) when the system state is not clearly defined, as in ChatGPT (Ouyang et al. 2022). Additionally, expressing a policy with features is often "easier" than expressing a value function with the same features (Sutton and Barto 2018; Francès et al. 2019; Francès, Bonet, and Geffner 2021).

Can policy gradient methods and in particular actor-critic algorithms be used to learn nearly perfect policies for classical planning benchmark domains? And if so, how are they to be used and what are their limitations? The literature does not provide a crisp answer to these questions, as the use of RL methods has focused on performance relative to baselines, and in classical planning, on learning heuristics (Shen, Trevizan, and Thiébaux 2020; Karia and Srivastava 2021; Ferber et al. 2022). Approaches that have aimed at nearly perfect general policies have not relied on RL methods, and much less on standard RL methods off the shelf. This is what we aim to do in this work.

The paper is organized as follows. We first provide background on classical planning, dynamic programming (DP), and general policies. Then, we review RL algorithms as approximations of exact DP methods, and adapt them to learn general policies. We look then at the representation of value and policy functions in terms of deep networks, present and analyze the experimental results, discuss related work, and draw the conclusions.

2 Background

We review classical planning, the two basic dynamic programming methods, and general policies.

2.1 Classical Planning

A classical planning problem is a pair $P = \langle D, I \rangle$ where D is a first-order *domain* and I contains information about the instance (Geffner and Bonet 2013; Ghallab, Nau, and Traverso 2016; Haslum et al. 2019). The domain D has a set of predicate symbols p and a set of action schemas with preconditions and effects given by atoms $p(x_1, \ldots, x_k)$ where p is a predicate symbol of arity k, and each x_i is an argument of the schema. An instance is a tuple $I = \langle O, \mathit{Init}, \mathit{Goal} \rangle$ where O is a set of object names c_i, and *Init* and *Goal* are sets of *ground atoms* $p(c_1, \ldots, c_k)$.

A classical problem $P = \langle D, I \rangle$ encodes a state model $S(P) = \langle S, s_0, S_G, \mathit{Act}, A, f \rangle$ in compact form where the states $s \in S$ are sets of ground atoms from P, s_0 is the initial state I, S_G is the set of goal states s such that $S_G \subseteq s$, *Act* is the set of ground actions in P, $A(s)$ is the set of ground actions whose preconditions are (true) in s, and f is the induced transition function where $f(a, s)$, for $a \in A(s)$, represents the state s' that follows action a in the state s. An action sequence $a_0, \ldots, a_n$ is applicable in P if $a_i \in A(s_i)$ and $s_{i+1} = f(a_i, s_i)$, for $i = 1, \ldots, n$, and it is a plan if $s_{n+1} \in S_G$. The *cost* of a plan is assumed to be given by its length and a plan is *optimal* if there is no shorter plan.

2.2 Dynamic Programming

In generalized planning one is interested in plans that reach the goal from any state s of a large collection of domain instances. Such plans can be represented in compact form by means of value $V(s)$ or policy functions $\pi(s)$ that map states into real numbers and actions respectively. Dynamic programming offers two basic methods for computing such functions, value and policy iteration, in the more general setting of Markov Decision Processes (MDPs) where the deterministic state transition function $f(a, s)$ and uniform action costs are replaced by transition probabilities $P_a(s'|s)$ and costs $c(a, s)$, $a \in A(s)$.

Value Iteration (VI) approximates the optimal value function $V^*(s)$, that provides the minimum (discounted) cost to reach the goal from s, by computing an approximate solution of the Bellman optimality equation

$$V(s) \;=\; \min_{a \in A(s)} \Big[c(a, s) + \gamma \sum_{s'} P_a(s'|s) V(s') \Big] \qquad (1)$$

where $V(s) = 0$ for goal states and $\gamma \in (0, 1)$ is the discount factor. This is done by initializing $V(s)$ to zero (although the initialization for non-goal states can be arbitrary) and updating the value vector V over non-goal states s as:

$$V(s) \;:=\; \min_{a \in A(s)} \Big[c(a, s) + \gamma \sum_{s'} P_a(s'|s) V(s') \Big] \,. \qquad (2)$$

In VI, these updates are done in parallel by using two V-vectors, while in *asynchronous* VI, a single V-vector is used, and there is no requirement that all states are updated at each iteration. In both cases, asymptotic convergence to V^* is guaranteed provided that all states are updated infinitely often (Bertsekas 1995). If V is optimal, i.e., $V = V^*$, the policy π_V that is greedy in V, is also optimal, where:

$$\pi_V(s) \;=\; \operatorname*{argmin}_{a \in A(s)} \Big[c(a, s) + \gamma \sum_{s'} P_a(s'|s) V(s') \Big] \,. \qquad (3)$$

Policy Iteration (PI) iterates on policies π rather than on value vectors V, and consists of two steps: policy evaluation and policy improvement. Starting with an arbitrary policy $\pi = \pi_0$, policy evaluation computes the values $V^{\pi}(s)$ that encode the expected cost to the goal from s when following policy π. These costs are computed by solving the linear Bellman equation for policy π:

$$V(s) \; = \; \Big[c(a,s) + \gamma \sum_{s'} P_a(s'|s)V(s') \Big] \qquad (4)$$

for $a = \pi(s)$ and $V(s) = 0$ for goal states s. If the policy π is not greedy with respect to the value function $V = V^{\pi}$, any policy π' that is greedy with respect to V^{π} is strictly better than π; i.e., $V^{\pi'}(s) \leq V^{\pi}(s)$ for all s with the inequality being strict for some states. Thus, in the policy improvement step, PI sets the policy π to π' and the process is repeated. This process finishes after a finite number of iterations when a policy π is obtained that is greedy relative to the value function $V = V^{\pi}$, as such a a policy cannot be improved and is optimal. The number of iterations is finite because the number of deterministic policies π that map states into actions is finite too.

2.3 General Policies

Generalized planning studies the representation and computation of policies that solve many classical planning instances from the same domain at once (Srivastava, Immerman, and Zilberstein 2008; Hu and De Giacomo 2011; Belle and Levesque 2016; Illanes and McIlraith 2019). A critical issue is how to represent general policies, considering that actions are deterministic in our setting. Clearly, they cannot be represented by policies that map states into (ground) actions because the number and name of the actions change with the set of objects. One simple representation of general policies is in terms of general value functions V (Francès et al. 2019; Ståhlberg, Bonet, and Geffner 2022a). These functions map states s over the instances in a class $\mathcal{Q}$ into non-negative scalar values $V(s)$ that are zero only at goal states, and they can be used to define greedy policies π_V that select the actions that lead to successor states s' with minimum $V(s')$ value. If the value of the child s' is always lower than the value of its parent state s, the value function V represents a general policy π_V that is guaranteed to solve any problem in the class $\mathcal{Q}$.

Another simple representation of general policies is as classifiers of state transitions into two categories, good and bad (Francès, Bonet, and Geffner 2021). If for each non-goal state s, there is a good state transition (s, s'), and chains of good transitions do not lead to dead-end states or cycles, the policy is guaranteed to solve any problem in the class.

A general policy is optimal if it results in shortest trajectories to the goal, yet optimality for general policies is not necessary and it is often impossible. For example, there are general policies for solving arbitraty instances of Blocksworld, but there are no optimal general policies,[1] as optimal planning in Blocksworld, like in many other "easy" classical planning domains, is intractable.

In this work, we utilize these ideas to represent policies as state transition classifiers using GNNs that accept state pairs s and s' and determine whether state transitions (s, s') are good or not. The policies are trained without supervision using policy optimization algorithms, to be discussed next.

3 From Exact DP to Approximate RL

While there are basically two exact model-based DP algorithms, there are many model-free RL/DRL methods. Rather than jumping directly to the algorithms that we will use, it will be convenient to understand the latter as suitable approximations of the former.

3.1 Approximate Prediction: Learning V

The main reason for doing approximation is that the number of states s may be too large or infinite, as in generalized planning. We focus on approximating value functions V^{π} since we aim at policy optimization algorithms, yet similar ideas can be applied for approximating the optimal value function V^*.

For policy optimization algorithms, it is convenient to consider *stochastic policies* π that assign probabilities $\pi(a|s)$ for selecting the action a in state s, rather than deterministic policies, as deep neural networks output real values. The Bellman equation for evaluating a stochastic policy π is:

$$V(s) \; = \; \sum_a \pi(a|s) \Big[c(a,s) + \sum_{s'} P_a(s'|s)V(s') \Big] \, . \quad (5)$$

The solution to this linear system of equations, provided that $V(s) = 0$ for goal states, is $V = V^{\pi}$, and a common method to solve it is by a form of value iteration that uses the Bellman equation for policy π instead of the optimality equation (1), with updates of the form:

$$V(s) := \sum_a \pi(a|s) \Big[c(a,s) + \gamma \sum_{s'} P_a(s'|s)V(s') \Big] \, . \quad (6)$$

Successive updates of this form ensure that V eventually converges to V^{π}. Three common approximations of this policy evaluation and resulting value function V^{π} are:

- **Sampling of (seed) states.** As in asynchronous VI, states s are selected for update at each iteration by some form of stochastic sampling which does not necessarily guarantee that all states are updated infinitely often.
- **Sampling of actions, costs, and successors.** Either because state transition probabilities and action costs are not known, or because there are too many actions or too many successor states, samples of the actions a, costs $c(a,s)$, and the successor states s' are used instead of considering all actions and successors. The resulting *sampled updated*

$$V(S) \; := \; V(S) + \alpha \big[C + \gamma V(S') - V(S) \big] \, , \qquad (7)$$

where α is the step size or *learning rate*, is characteristic of stochastic approximation methods (Robbins and Monro 1951; Harold, Kushner, and George 1997). If the action A is sampled with probability $\pi(A|S)$ and

[1] Policies that decide what action to apply in polynomial time.

S' with probability $P_A(S'|S)$, written $A \sim \pi(\cdot|S)$ and $S' \sim P_A(\cdot|S)$, the expression $C + \gamma V(S')$ is an *unbiased estimator* of the right-hand side of (6), and hence the sampled backups guarantee that V converges to V^π provided standard conditions on the step sizes and that all states s are updated infinitely often.

- **Function approximation.** Finally, the value function $V(s)$ can be represented by a deep neural network with adjustable parameters ω. The updates no longer change the value of entries in a table (tabular updates) but the value of the parameters ω for minimizing the *squared loss* of the *sampled Bellman residual*

  $$\frac{1}{2}\big[C + \gamma V(S') - V(S)\big]^2$$

  through a step of gradient descent:

  $$\omega \;:=\; \omega + \alpha\big[C + \gamma V(S') - V(S)\big]\nabla V(S)\,, \quad (8)$$

  where α is the step size, $A \sim \pi(\cdot|S)$ and $S' \sim P_A(\cdot|S)$, and $\nabla V(S)$ is the gradient of the function V relative to ω evaluated at the state S. This update expression follows from the standard formula for minimizing a function (locally) using its gradient except that it assumes that the "target" value $V(S')$ does not depend on ω, a so-called "semi-gradient" method (Sutton and Barto 2018).

These three approximations are largely independent of each other and do not have to be used together. For example, Agostinelli et al. (2019) learn an approximation of the optimal value function V^* for guiding the search in Rubik's Cube by using value iteration while representing the value function by a deep net. A similar approach is used by Ståhlberg, Bonet, and Geffner (2022b) for learning general value functions from small instances. In the approach below, however, it is not the value functions that "transfer" (generalize) to new instances but the learned policy π itself.

3.2 Approximate Control: Learning π

Policy gradient methods make use of approximations of the value functions V^π for improving a differentiable policy π via gradient descent (Williams and Peng 1991; Sutton and Barto 2018). If the expected cost $J(\pi)$ of a policy π over an MDP with prior h over the initial states and parameters θ is

$$J(\pi) \;=\; \textstyle\sum_s h(s) V^\pi(s)\,, \quad (9)$$

the gradient of $J(\pi)$ relative to the vector of parameters θ satisfies (Sutton and Barto 2018, Policy Gradient Theorem):

$$\nabla J(\pi) \;\propto\; \sum_s \mu(s) \sum_a Q^\pi(s,a) \nabla \pi(a|s) \quad (10)$$

where

$$Q^\pi(s,a) \;=\; c(s,a) + \sum_{s'} P_a(s'|s) V^\pi(s') \quad (11)$$

and $\mu(s)$ stands for the fraction of times that executions of the policy π visit the state s when the prior is $h(s)$. The sum in (10) corresponds to the expectation

$$\nabla J(\pi) \;\propto\; \mathbb{E}_{S\sim\mu, A\sim\pi(\cdot|S)}\big[Q^\pi(S,A) \nabla \ln \pi(A|S)\big] \quad (12)$$

where S and A are random variables that stand for a state and action sampled according to μ and $\pi(\cdot|S)$ respectively. Gradient descent adjusts the parameters θ of the policy as:

$$\theta \;:=\; \theta - \alpha \nabla J(\pi)\,. \quad (13)$$

As before, common approximations of the gradient (12) are:

- **Sampling states and actions.** Expectation in (12) can be replaced by the unbiased estimator $Q^\pi(S,A)\nabla \ln \pi(A|S)$ where state S and action A are sampled by executing π.
- **Approximating Q- and V-values.** The $Q^\pi(S,A)$ value can be replaced by the unbiased estimator $C + V^\pi(S')$ with $S' \sim P_A(\cdot|S)$ and $C \sim c(A,S)$, and $V^\pi(S')$ can be approximated as discussed above.
- **Use of baselines.** A term $b(S)$ is decremented from the $Q^\pi(S,A)$ value in (12), where $b(S)$ is a function that depends on the state S but not on the action A or the successor state S'. It can be shown that this offset does not affect the value of the expectation but reduces its variance (Konda and Tsitsiklis 1999).

By combining these approximations, the expression for the policy parameter update becomes, for example, one of the standard forms of the *actor-critic (AC) RL algorithm* (Konda and Tsitsiklis 1999; Sutton and Barto 2018) where the policy parameters are updated as:

$$\theta \;:=\; \theta - \alpha\big[C + V(S') - V(S)\big]\nabla \ln \pi(A|S) \quad (14)$$

where S and A are sampled by executing the policy π, C is the sampled cost, V is an approximation of $V^\pi(S)$, and the successor state S' is sampled with probability $S' \sim P_A(\cdot|S)$. In this case, the baseline is $b(S) = V(S)$, and the same samples for S, A and S' are used for updating V using (8).

Other AC variants can be obtained from other approximations of the "actor's" gradient $\nabla J(\pi)$ and the "critic" V^π. Other RL algorithms are the so-called value-based methods, like SARSA and Q-learning, that do not consider policies explicitly, and Monte-Carlo RL methods like REINFORCE, that do not consider explicit value functions.

4 RL for Generalized Planning

In our setting of generalized planning, states are planning states represented as sets of ground atoms $p(t)$ where p is a domain predicate and t is a tuple of objects of the same arity as p. In addition, for generalizing to arbitrary conjunctive, ground goals, the goals of a planning instance, which are given by ground atoms $p(t)$, are represented as part of the state s by using new predicate symbols p_G (Martín and Geffner 2004; Bonet, Francès, and Geffner 2019). Atoms like $clear(c)$ and $clear_G(c)$ in a state of Blocksworld, for example, say that the block c is clear in the state and that the block c must be clear in the goal, respectively. A goal state in this representation is a state s where for each goal atom $p_G(t)$ in s, $p(t)$ is in s.

The two actor-critic algorithms for learning general policies over classical planning domains using this representation of states are shown in Figs. 1 and 2.

The first algorithm, AC-1, closely follows the gradient update rule in (14) and the value update rule in (8). It is a standard AC algorithm where states S, actions A, and successor states S' are sampled and adapted to the generalized setting.

Algorithm 1 Standard Actor-Critic for generalized planning: successor states s' sampled with probability $\pi(s'|s)$.

```
Input: Training MDPs $\{M_i\}_i$, each with state priors $p_i$
Input: Differentiable policy $\pi(s|s')$ with parameter $\theta$
Input: Diff. value function $V(s)$ with parameter $\omega$
Parameters: Step sizes $\alpha, \beta > 0$, discount factor $\gamma$
Initialize parameters $\theta$ and $\omega$
Loop forever:
    Sample MDP index $i \in \{1, \ldots, N\}$
        Sample non-goal state $S$ in $M_i$ with probability $p_i$
        Sample successor state $S'$ with probability $\pi(S'|S)$
        Let $\delta = 1 + \gamma V(S') - V(S)$
        $\omega \leftarrow \omega + \beta\delta\nabla V(S)$                Eq. (14)
        $\theta \leftarrow \theta - \alpha\delta\nabla \log \pi(S'|S)$            Eq. (8)
        If $S'$ is a goal state, $\omega \leftarrow \omega - \beta V(S')\nabla V(S')$
```

- The stochastic policies π are not assumed to encode a probability distribution over the actions in a given state because the set of actions changes from instance to instance. Policies are assumed instead to map states s into a probability distribution $\pi(s'|s)$ over the set $N(s)$ of possible successor states s' of s, and only indirectly, into a probability distribution over the actions applicable in s.

- The training set is given by a collection $\{M_i\}_i$ of (deterministic) MDPs. States s are then sampled at training time from a sampled M_i using a state prior p_i. The distributions over states in M_i and over the training MDPs M_i are assumed to be uniform.

- Action costs are all equal to 1 and thus there are no explicit sampled costs in the algorithm.

- When a sampled successor state s' is a goal state, the value function parameters are updated to ensure that the value of the goal state is zero. This is achieved by minimizing the loss function $1/2\, V(s')^2$, which drives the value at the goal state s' towards zero.

- Finally, no trajectories are sampled; or equivalently, the only sampled trajectories have length $T = 1$, corresponding to a single state transition. The reasons for this are discussed in the experimental section.

The second algorithm, AC-M in Fig. 2, is a small variation of the first where the seed states S are sampled but not the successor states S'. Instead, in the update expressions of the policy and value function parameters, a sum over all possible successor states s' is used, which in the case of value updates, are weighted by the probabilities $\pi(s'|S)$ and result in full Bellman updates. Algorithm AC-M is an actor-critic algorithm that makes use of the training models M_i and which often converges faster than the model-free version AC-1. Notice that for applying the learned policy π from either AC-1 or AC-M, one must be able to determine the possible successors s' of a state s in order to assess the probabilities $\pi(s'|s)$ that define the stochastic policy. A full model-free approach for generalized planning would need to learn this structure as well.

Algorithm 2 All-Actions Actor-Critic: all successor states considered for updating policy and value functions.

```
Input: Training MDPs $\{M_i\}_i$, each with state priors $p_i$
Input: Differentiable policy $\pi(s|s')$ with parameter $\theta$
Input: Diff. value function $V(s)$ with parameter $\omega$
Parameters: Step sizes $\alpha, \beta > 0$, discount factor $\gamma$
Initialize parameters $\theta$ and $\omega$
Loop forever:
    Sample MDP index $i \in \{1, \ldots, N\}$
        Sample non-goal state $S$ in $M_i$ with probability $p_i$
        Let $V' = 1 + \gamma\Sigma_{s' \in N(S)} [\pi(s'|S)V(s')]$
        Let $b(S) = V' - 1$ be the baseline
        $\omega \leftarrow \omega + \beta(V' - V(S))\nabla V(S)$
        $\theta \leftarrow \theta - \alpha\Sigma_{s' \in N(S)} [(V(s') - b(S))\nabla\pi(s'|S)]$
        If $s' \in N(S)$ is goal state, $\omega \leftarrow \omega - \beta V(s')\nabla V(s')$
```

5 Neural Network Architecture

The value and policy functions, $V(s)$ and $\pi(s'|s)$, are represented using graph neural networks (Scarselli et al. 2008; Hamilton 2020) adapted for dealing with relational structures. The GNNs produce object embeddings $\phi(o)$ that then feed suitable readout functions. We adopt the GNN architecture of Ståhlberg, Bonet, and Geffner (2022a; 2022b) for representing value functions over planning states. This architecture is a variation of a similar one used for solving Max-CSP problems (Toenshoff et al. 2021).

5.1 GNNs on Graphs

GNNs represent trainable, parametric, and generalizable functions over graphs specified by means of aggregate and combination functions agg_i and $comb_i$, and a readout function F. The GNN maintains an embedding $f_i(v) \in \mathbb{R}^k$ for each vertex v of the input graph G. Here, i ranges from 0 to L, which is the number of iterations or layers. The vertex embeddings $f_0(v)$ are fixed and the embeddings $f_{i+1}(v)$ for all v are computed from the f_i embeddings as:

$$f_{i+1}(v) := comb_i(f_i(v), agg_i(\{\!\{f_i(w) | w \in N_G(v)\}\!\})) \quad (15)$$

where $N_G(v)$ is the set of neighbors for vertex v in G, and $\{\!\{\ldots\}\!\}$ denotes a multiset. This iteration is usually seen as an exchange of messages among neighbor nodes in the graph. Aggregation functions agg_i like max, sum, or smooth-max, map arbitrary collections of real vectors of dimension k into a single $\mathbb{R}^k$ vector. The combination functions $comb_i$ map pairs of $\mathbb{R}^k$ vectors into a single $\mathbb{R}^k$ vector. The embeddings $f_L(v)$ in the last layer are aggregated and mapped into an output by means of a readout function. All the functions are parametrized with weights that are learnable. By design, the function computed by a GNN is *invariant* with respect to graph isomorphisms, and once a GNN is trained, its output is well defined for graphs G of any size.

5.2 GNNs for Relational Structures

States s in planning do not represent graphs, but more general relational structures. These structures are defined by a set of objects, a set of domain predicates, and the atoms

Algorithm 3 Graph Neural Network (GNN) architecture that maps state s into object embedding $f(o) = f_L(o)$ from which value and policy functions defined.

```
Input: State $s$ (set of atoms true in $s$), set of objects
Output: Embeddings $f_L(o)$ for each object $o$
$f_0(o) \sim \mathbf{0}^k$ for each object $o \in s$
For $i \in \{0, \ldots, L-1\}$
  For each atom $q := p(o_1, \ldots, o_m)$ true in state $s$:
    $m_{q,o} := [\mathbf{MLP}_p(f_i(o_1), \ldots, f_i(o_m))]_j$
  For each object $o$ in state $s$:
    $f_{i+1}(o) := \mathbf{MLP}_U\big(f_i(o), agg(\{\!\{m_{q,o} | o \in q\}\!\})\big)$
```

$p(o_1, \ldots, o_m)$ that are true in the state. The objects define the universe, the domain predicates define the relations, and the atoms represent their denotations. The set of predicate symbols p and their arities are fixed by the domain, but the sets of objects o_i may change from instance to instance. The GNN used for dealing with planning states s computes object embeddings $f_i(o)$ for each of the objects o in the input state s, and rather than messages flowing from "neighbor" objects to objects as in (15), the messages flow from objects o_i to the atoms q in s that include o_i, $q = p(o_1, \ldots, o_m)$, $1 \leq i \leq m$, and from such atoms q to all the objects o_j involved in q as (see Fig. 3):

$$f_{i+1}(o) := comb_i(f_i(o), agg_i(\{\!\{m_{q,o} | o \in q, q \in s\}\!\})) \quad (16)$$

where $m_{q,o}$ for $q = p(o_1, \ldots, o_m)$ and $o = o_j$ is:

$$m_{q,o} := [comb_p(f_i(o_1), \ldots, f_i(o_m))]_j \,. \quad (17)$$

In these updates, the combination function $comb_i$ takes the concatenation of two real vectors of size k and outputs a vector of size k, while the combination function $comb_p$, that depends on the predicate symbol p, takes the concatenation of m vectors of size k, where m is the arity of p, and outputs m vectors of size k as well, one for each object involved in the p-atom. The expression $[\ldots]_j$ in (17) selects the j-th such vector in the output. In Figure 3, we use the same *agg* and *comb* in every layer, and their weights are shared. Each of the multilayer perceptrons (MLPs) is composed of a residual block, followed by a linear layer that reshapes the output to the desired size. A residual block consists of a linear layer, a non-linear activation function, and another linear layer. The output of the residual block is the sum of the input and the result of the second linear layer. We used the non-linear activation function *Mish* in our implementation (Misra 2020).

5.3 From the Object Embeddings to V and π

The readout functions $V(s)$ and $\pi(s'|s)$ are computed by aggregating object embeddings. If $f^s(o) = f_L(o)$ denotes the final embedding for object o in state s, the value for s is

$$V(s) = \mathbf{MLP}\big(\textstyle\sum_{o \in O} f^s(o)\big)\,, \quad (18)$$

where the $\mathbf{MLP}$ outputs a single scalar.

The policy π, in turn, must yield the probabilities $\pi(s'|s)$ for each successor state s' in $N(s)$. This is achieved by first computing *logits* for pairs (s, s') and then passing the logits through a *softmax*:

$$\text{logit}(s'|s) = \mathbf{MLP}\big(\textstyle\sum_{o \in O} \mathbf{MLP}(f^s(o), f^{s'}(o))\big)\,, \quad (19)$$

$$\pi(s'|s) \propto \exp\big(\text{logit}(s'|s)\big) \quad (20)$$

where the inner $\mathbf{MLP}$ outputs a vector of size $2k$, and the outer $\mathbf{MLP}$ outputs a single scalar. The purpose of the inner $\mathbf{MLP}$ is to derive new features that are specific to the transition. For example, it can identify that the agent is no longer holding an item. Note that we need to know all the successor states to determine $\pi(s'|s)$ as the logits for all successors are needed to compute the softmax.

It is also important to note that the nets for $V(s)$ and $\pi(s'|s)$ share weights since the object embeddings come from the same GNN. However, the weights used to define the readout functions are not shared.

6 Experimental Results

The experiments test the generalization, coverage, and quality of the plans obtained by the learned policies. We describe the data for training and testing, and the results obtained. We aim at crisp results that mean close to 100% generalization, and when this is not possible, to provide clear explanations and in some cases logical fixes that restore generalization.

Data To create the training and validation sets, we generate the reachable state space from the initial state, along with shortest paths to a goal state. We limit the training set to small instances with a maximum of 200,000 transitions. If this condition is not met by the IPC instances or if better diversity is needed, we generate our own instances, aiming for approximately 24 per domain in the test set.

We used the same domains as Ståhlberg, Bonet, and Geffner (2022b) in our experiments, with the addition of Grid. The domains and the data used for each are:

- **Blocks.** The goal is to build a single tower but in Blocks-multiple the goal consists of multiple towers. The training, validation and test sets have instances from the IPC with 4-7, 7, and 8-17 blocks, resp. For Blocks-multiple, the test set includes instances with up to 20 blocks.
- **Delivery.** The problem involves picking up objects in a grid with no obstacles and delivering them one by one to a target cell. The instances consist of up to 9 × 9 grids with up to 4 packages. The training, validation, and test sets are partitioned based on the size of the reachable state space. The training instances are smaller than the validation instances, and the largest are in the test set.
- **Grid.** The goal is to find keys, open locked doors, and place specific keys at certain locations. The instances consist of up to 9 × 7 grids with up to 3 locks and 5 keys. The instances are partitioned as in Delivery into training, validation and test set.
- **Gripper.** The task is to move balls from one room to another using a robot with two grippers. The training, validation, and test instances contain of 1-9, 10, and 12-50 balls, respectively. The IPC instances only go up to 42 balls.
- **Logistics.** Domain that involves packages, cities, trucks and airplanes. The instances vary between 1-2 airplanes, 1-4 cities with 2 locations each, and 1-6 packages, with exactly one truck in each city. Instances are partitioned into training, validation and test set on the size of rechable state space.

- **Miconic.** Planning for lift that picks and delivers passengers at different floors. The number of floors and passengers for training and validation vary between 2-8 and 1-5, resp. The instances with the largest state space are for validation. For the test set, we use IPC instances that feature up to 59 floors and 29 people
- **Reward.** Move in a grid with obstacles to pick up all rewards. The instances are from Francès, Bonet, and Geffner (2021). The training, validation, and test sets consist of square grids with widths in 3-10, 10, and 15-25, resp. The maximum number of rewards for training and validation is 8, and 23 for testing.
- **Spanner.** The task is to tighten nuts at one end of a corridor with spanners that need to be collected on the way. The number of locations varies between 1-20 for training, validation, and testing, with at most 4 nuts for training and validation, and 12 nuts for testing. The training and validation sets have no more than 6 spanners, while the test set have up to 24.
- **Visitall.** The task is to visit all or some cells in a grid with no obstacles. The grid sizes for training are limited to 3×3, 4×2, and smaller. The validation set uses 5×2 grids, while the test set includes grids of sizes ranging from 4×4 to 10×10.

Setup The GNN architecture is instantiated with hyperparameters $k = 64$ and $L = 30$, and the discount factor $\gamma = 0.999$ is used in the AC algorithms. The hyperparameters k and L affect training speed, memory usage, and generalization (e.g., the GNN cannot compute shortest paths of length longer than L). The architecture is implemented in PyTorch (Paszke and et. al. 2019) using Adam with learning rate of 0.0002 (Kingma and Ba 2015).[2] The networks are trained using NVIDIA A100 GPUs for up to 6 hours. Two models for each domain are trained, and the final model is the one with the best policy evaluation average on the validation set. The quality of the plans is determined by comparing their length to the length of optimal plans computed with Fast Downward (FD) (Helmert 2006) using the *seq-opt-merge-and-shrink* portfolio with a time limit of 10 minutes and 64 GB of memory on a Ryzen 9 5900X CPU.

Results We tested the learned policies in two modes. The first as a *stochastic policy* that selects actions randomly following the distribution provided by the policy. The second as a *deterministic policy* that selects the most probable successor state. Stochastic policies have the advantage of being able to escape cycles eventually. However, this is not possible with deterministic policies, so we used a closed set to avoid sampling an already visited successor. Executions are terminated when the goal is not reached within 10,000 steps.

Table 1 presents the experimental results, divided in two subtables: learning using the standard, sampled Actor-Critic algorithm (top), and learning using the all-actions Actor-Critic (bottom). The left side of each subtable shows the results obtained with the stochastic policy, while the right hand side shows the results obtained with the deterministic policy. We discuss test coverage and analyze the limitations encountered, which as we will see, *have more to do with representation and complexity issues than with reinforcement or deep learning*. Indeed, we will use the analysis to address the limitations in an informed manner. Later, we discuss plan quality and the impact of sampled trajectory length on learning, where the result in the table are all for length 1.

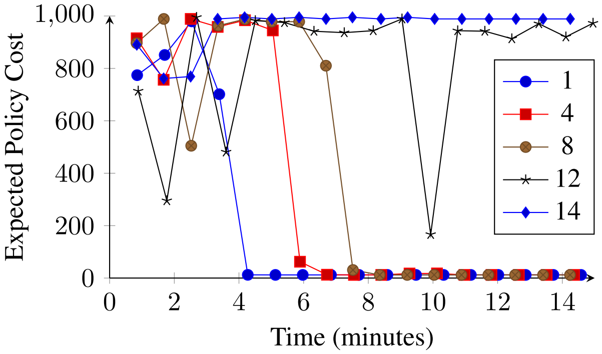

Figure 1: Expected policy cost over the validation set during training sessions for Gripper with standard Actor-Critic for different trajectory lengths $T = 1, 4, 8, 12, 14$. Table of results used $T = 1$ which as seen here, does best. The training set included instances with up to 7 balls, while the validation instance had 8 balls.

Coverage We got a very high coverage out of the box in 6 out of the 10 domains considered, where the learned policy either solves all instances or almost all instances. It is surprising, however, that one of the domains where we got perfect coverage for is Blocks. This is unexpected because finding an optimal solution for Blocks instances is an NP-hard problem, thus there does not exists a compact policy. In the IPC problems, the goal is to build a single tower. We tested whether it is possible to learn a policy that does not rely on this goal structure and can construct any number of towers instead. We achieved perfect coverage for this version, and the results are not due to "luck" because each successor state was selected with 100% probability (rounded to the nearest integer). As these policies were learned using actor-critic, we can also define a greedy policy based on the learned value function (critic), which selects the successor state with the lowest value. These policies solved all but three Blocks problems with eight blocks, for the model learned using all-actions Actor-Critic, and in these cases, the critic and the actor disagreed on the best successor state.

Quality The last two columns in Table 1 show the average (policy evaluation) value over the entire validation set for the optimal policy and the learned policy. This value represents the expected number of steps to a goal state from a uniformly sampled initial state. When the difference between V^{π} and V^{*} is small, we can expect the policy to generalize well and produce plans that are close to optimal.

Sampling and Length of Sampled Trajectories Algorithms AC-1 and AC-M implicitly assume sampled trajectories of length 1. After sampling a state s from a training MDP with a uniform prior, a successor state s' is sampled using probability $\pi(s'|s)$, and the process continues with another state s sampled uniformly. In RL practice, longer trajectories are commonly sampled, involving updating value and policy functions at each state before sampling another initial state. In our setting, where we seek general poli-

[2] Code and data: https://zenodo.org/record/7993858

	Stochastic Policy			Deterministic Policy			Validation	
Domain (#)	Coverage (%)	L	PQ = PL / OL (#)	Coverage (%)	L	PQ = PL / OL (#)	V_{π^*}	V_π
Standard Actor-Critic								
Blocks (23)	23 (100 %)	810	1.00 = 478 / 476 (16)	23 (100 %)	810	1.00 = 478 / 476 (16)	18.60	18.60
Blocks-multiple (26)	26 (100 %)	898	1.00 = 450 / 448 (16)	26 (100 %)	898	1.00 = 450 / 448 (16)	17.57	17.57
Delivery (24)	21 (88 %)	591	1.01 = 544 / 540 (20)	23 (96 %)	701	1.02 = 596 / 586 (21)	16.23	34.46
Grid (14)	5 (36 %)	43	1.00 = 43 / 43 (5)	10 (71 %)	516	2.68 = 356 / 133 (9)	18.13	668.28
Gripper (20)	17 (85 %)	5 031	1.00 = 176 / 176 (4)	16 (80 %)	1 312	1.00 = 176 / 176 (4)	14.89	14.89
Logistics (22)	3 (14 %)	33	1.10 = 33 / 30 (3)	17 (77 %)	37 667	77.4 = 23 839 / 308 (15)	9.04	636.43
Miconic (30)	29 (97 %)	1 438	1.00 = 185 / 185 (10)	29 (97 %)	1 438	1.00 = 185 / 185 (10)	6.42	6.42
Reward (15)	13 (87 %)	2 953	2.38 = 1 328 / 558 (7)	6 (40 %)	655	1.24 = 362 / 292 (4)	22.15	120.34
Spanner (22)	18 (82 %)	682	1.11 = 218 / 197 (7)	18 (82 %)	678	1.11 = 218 / 197 (7)	130.74	130.82
Visitall (24)	24 (100 %)	1 083	1.23 = 762 / 621 (20)	24 (100 %)	1 032	1.12 = 696 / 621 (20)	4.36	4.39
Total (220)	179 (81 %)	13 562	-	192 (87 %)	45 707	-	-	-
All-Actions Actor-Critic								
Blocks (23)	23 (100 %)	806	1.00 = 476 / 476 (16)	23 (100 %)	806	1.00 = 476 / 476 (16)	18.60	18.60
Blocks-multiple (26)	26 (100 %)	902	1.00 = 450 / 448 (16)	26 (100 %)	902	1.00 = 450 / 448 (16)	17.57	17.57
Delivery (24)	23 (96 %)	691	1.00 = 586 / 586 (21)	24 (100 %)	757	1.03 = 652 / 632 (22)	16.23	16.23
Grid (14)	6 (43 %)	71	1.00 = 71 / 71 (6)	11 (79 %)	292	1.50 = 248 / 165 (10)	18.13	84.21
Gripper (20)	20 (100 %)	1 840	1.00 = 176 / 176 (4)	20 (100 %)	1 840	1.00 = 176 / 176 (4)	14.89	14.89
Logistics (22)	0 (0 %)	-	-	8 (36 %)	7 981	63.8 = 7 981 / 125 (8)	9.04	511.76
Miconic (30)	30 (100 %)	1 527	1.00 = 185 / 185 (10)	30 (100 %)	1 527	1.00 = 185 / 185 (10)	6.42	6.42
Reward (15)	7 (47 %)	2 493	1.29 = 442 / 342 (4)	12 (80 %)	1 464	1.21 = 582 / 481 (6)	22.15	167.25
Spanner (22)	15 (68 %)	552	1.10 = 149 / 135 (5)	15 (68 %)	552	1.10 = 149 / 135 (5)	130.74	130.81
Visitall (24)	23 (96 %)	5 670	7.05 = 3 934 / 558 (19)	24 (100 %)	971	1.08 = 671 / 621 (20)	4.36	4.40
Total (220)	173 (79 %)	14 552	-	193 (88 %)	17 092	-	-	-

Table 1: Performance of learned policies. The top subtable displays the results obtained from using standard update rules for Actor-Critic, while the middle and bottom subtables display the results obtained from using all-actions update rules for Actor-Critic, where the critic is either a neural network or a table, respectively. The learned policies were tested as both stochastic policies and deterministic policies. In the former, a transition is selected with the probability given by the policy, while in the latter, the most likely transition is always selected and the set of visited states is tracked to prevent cycles. The domains are shown on the left along with the number of instances in the test set. The coverage refers to the number of solved problems, and each policy was given a maximum of 10.000 steps to reach a goal state. L represents the sum of the solution lengths over the test instances solved by the learned policy. PQ is a measure of overall plan quality given by the ratio of the sum of the plan lengths found by the learned policy and the optimal policy, which was determined with the help of Fast-Downward (FD). The number within parenthesis represents the number of instances FD was able to solve within our time and memory constraints. The columns V_{π^*} and V_π display the average value over the states in the validation set for the optimal policy and the learned policy, respectively, determined by policy evaluation using the stochastic policies.

cies and there are no privileged initial states, this alternative strategy was found to be detrimental to performance, as shown in Figure 1, and also in vanilla and tuned implementations of PPO that we tried (Schulman et al. 2017; Raffin et al. 2021). Sampling state transitions from a buffer of stored experiences is common in the so-called experience replay and off-policy methods (Mnih et al. 2015; Fujimoto, Hoof, and Meger 2018; Haarnoja et al. 2018).

Identifying and Overcoming Limitations Four domains, namely Grid, Logistics, Reward, and Spanner, were not solved nearly perfectly in any mode. Interestingly, failures can be more instructive than successes, and this is no exception. Below, we will analyze the causes of these failures and use them to address the issues. Methodologically, this is important, as the first instinct when an RL algorithm fails is to try a different one. This may improve the numbers, but in this case, as we will see, it will not be sufficient or necessary. We will demonstrate how to achieve perfect coverage in these domains by sticking to our basic RL algorithms. The limitations do not stem from RL but from logical and complexity considerations.

The performance in these four domains results is affected by the *limited expressivity of GNNs* and the *optimality/generality tradeoff* (Ståhlberg, Bonet, and Geffner 2022b). GNNs can capture the features that can be expressed in the two-variable fragment of first-order logic with counting, C_2, but not those in larger fragments like C_3 (Barceló et al. 2020; Grohe 2021), that are needed in Grid and Logistics. In addition, the number of layers L in the GNN puts a limit on the lengths of the distances that can be computed; a problem that surfaces in the Reward domain, where the learned “agent”cannot move to the closest reward because it is just too far away. The optimality/generalization tradeoff arises in domains like Logistics and Grid that admit compact general policies but no compact policies that are optimal, because optimal planning in both is NP-hard (Helmert 2003). Yet RL algorithms aim to learn optimal policies which thus cannot generalize properly.

In Logistics, the GNN cannot determine whether a vehicle carrying a package is located in the city where the package needs to be delivered as this requires C_3 expressiveness. For learning a policy with the same GNN architecture, the states are extended to include (derived) atoms indicating whether the package is in the correct city, regardless of whether it is on a plane, a truck, or at an incorrect location within the same city. This version of Logistics achieved 91% coverage (with a plan quality of $4.53 = 1612/356$ (18)), which we trained using an all-action Actor-Critic and evaluated as a deterministic policy. For addressing, the optimality/generalization tradeoff, we modified the cost structure of the training MDPs so that the learned policies optimize the probability of reaching the goal without entering a cycle instead of the expected cost.[3] Provided with the extra predicate and the new cost structure, a coverage of 100% was achieved in Logistics with a plan quality of $1.11 = 410/368$ (19) using a deterministic policy (we expect to obtain similar results in Grid with these two extensions, but we have not achieved them yet because the instance generator produces too many unsolvable or trivial instances).

The final domain without full coverage is Spanner (Table 1), where we observed that the learned policies fail when given instances with more spanners or nuts than those in the training instances. There is no logical reason for this failure since dead-end state detection is possible with C_2 features (Ståhlberg, Francès, and Seipp 2021). To tackle this issue, we tested a *tabular version* of the all-actions Actor-Critic algorithm where the learned transition probabilities $\pi(s' \mid s)$ are stored and then used to solve the linear Bellman equation for V^{π}. This approach gives accurate values, which for dead-end states are *always* $\frac{1}{1-\gamma}$ where γ is the discount factor. The learned policy achieved then 100% coverage in both modes (stochastic and deterministic) with a plan quality of $1.09 = 315/290$ (10), suggesting that the problem in the two Actor-Critic algorithms is that they do not sample dead-end states enough. The tabular evaluation of policies, however, can only deal with small instances, and for this reason it only worked in one other domain, Visitall.

7 Related work

General Policies Using Deep (Reinforcement) Learning DL and DRL methods (Sutton and Barto 2018; Bertsekas 1995; François-Lavet et al. 2018) have been used to learn general policies (Kirk et al. 2023). In some cases, the planning representation of the domains is used (Toyer et al. 2020; Garg, Bajpai, and Mausam 2020; Rivlin, Hazan, and Karpas 2020); in most cases, it is not (Groshev et al. 2018; Chevalier-Boisvert et al. 2019; Campero et al. 2021; Cobbe et al. 2020; Küttler et al. 2020). Also in some cases, the learning is supervised; in others, it is based on RL. Closest to our work is the use of GNNs for learning nearly-perfect general policies in planning domains using supervised and approximate value iteration methods (Ståhlberg, Bonet, and Geffner 2022a; Ståhlberg, Bonet, and Geffner 2022b)

General Policies Using Logical Methods Learning general policies has been studied in the planning setting (Srivastava, Immerman, and Zilberstein 2008; Hu and De Giacomo 2011; Belle and Levesque 2016; Illanes and McIlraith 2019; Segovia, Jiménez, and Jonsson 2016) where logical and combinatorial methods have been used (Khardon 1999; Martín and Geffner 2004). The representation of such policies as classifiers for state transition is from Francès, Bonet, and Geffner (2021).

General Policies and Causal Models Causal models (Pearl 2009) have been used to address the problem of out-of-distribution generalization in DL (Goyal and Bengio 2020; Schölkopf et al. 2021), as causal relations are modular and invariant (remain true after interventions). Recent work on policy learning has incorporated inductive biases motivated by causal considerations (Zhang et al. 2020; Sonar, Pacelli, and Majumdar 2021; Wang et al. 2022). Planning representations are causal too, and action schemas are modular and invariant in a domain. It is no accident that we learn policies that generalize well when using the resulting state languages. Methods for learning the action schemas and the predicates involved have also been developed (Asai 2019; Bonet and Geffner 2020; Rodriguez et al. 2021).

8 Conclusions

Deep learning (DL) and deep reinforcement learning (DRL) have caused a revolution in AI and a significant impact outside of AI. Yet, the methods developed in DL and DRL while incredibly powerful, are not reliable or transparent, and the research methodology is often too much focused on experimental performance relative to baselines and not on understanding. The problem of generalization has been central in DRL for many years but the analysis is hindered by the lack of a language to represent states, and a language for representing classes of problems over which the policies should generalize. The setting of classical planning gives us both, and has helped us to understand the power of DRL methods, and to identify and address limitations, that have little to do with DL or DRL. Indeed, we have shown that DRL methods learn nearly-perfect general policies out of the box in six of the ten domains considered, and that three domains fail for structural reasons: expressive limitations of GNNs and the optimality/generalization tradeoff. We have addressed these limitations as well by extending the states with derived predicates and by adapting the DRL algorithms to optimize a different cost measure. Our approach to learning general plans has its own limitation such as the assumption that a lifted model of the domains is known, that we and others have addressed elsewhere. An overall lesson is that there is no need to choose between crisp AI symbolic models and data-derived deep (reinforcement) learners. The latter can be understood as a new powerful class of solvers that are worth having in the toolbox.

[3] We omit the details of the resulting algorithm for lack of space, but it involves a third parametric function D that approximates this probability. The policy π optimizes D^{π} and uses V to avoid cycles (the only possible successors of s for evaluating D^{π} are those that decrease V by more than ϵ. In the experiments, $\epsilon = 0.75$).

Acknowledgements

This research was partially supported by the European Research Council (ERC), Grant No. 885107, and by project TAILOR, Grant No. 952215, both funded by the EU Horizon research and innovation programme. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were enabled by the supercomputing resource Berzelius provided by National Supercomputer Centre at Linköping University and the Knut and Alice Wallenberg foundation.

References

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence* 1(8):356–363.

Asai, M. 2019. Unsupervised grounding of plannable first-order logic representation from images. In *Proc. ICAPS*, 583–591.

Barceló, P.; Kostylev, E. V.; Monet, M.; Pérez, J.; Reutter, J.; and Silva, J. P. 2020. The logical expressiveness of graph neural networks. In *ICLR*.

Belle, V., and Levesque, H. J. 2016. Foundations for generalized planning in unbounded stochastic domains. In *Proc. KR*, 380–389.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, Vols 1 and 2*. Athena Scientific.

Bonet, B., and Geffner, H. 2015. Policies that generalize: Solving many planning problems with the same policy. In *Proc. IJCAI*, 2798–2804.

Bonet, B., and Geffner, H. 2020. Learning first-order symbolic representations for planning from the structure of the state space. In *Proc. ECAI*, 2322–2329.

Bonet, B.; De Giacomo, G.; Geffner, H.; and Rubin, S. 2017. Generalized planning: Non-deterministic abstractions and trajectory constraints. In *Proc. IJCAI*, 873–879.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning features and abstract actions for computing generalized plans. In *Proc. AAAI*, 2703–2710.

Campero, A.; Raileanu, R.; Kuttler, H.; Tenenbaum, J. B.; Rocktäschel, T.; and Grefenstette, E. 2021. Learning with AMIGo: Adversarially motivated intrinsic goals. In *ICLR*.

Chevalier-Boisvert, M.; Bahdanau, D.; Lahlou, S.; Willems, L.; Saharia, C.; Nguyen, T. H.; and Bengio, Y. 2019. Babyai: A platform to study the sample efficiency of grounded language learning. In *ICLR*.

Cobbe, K.; Hesse, C.; Hilton, J.; and Schulman, J. 2020. Leveraging procedural generation to benchmark reinforcement learning. In *Proc. ICML*, 2048–2056.

Drexler, D.; Seipp, J.; and Geffner, H. 2022. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS*, 62–70.

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In *Proc. ICAPS*, 583–587.

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized potential heuristics for classical planning. In *Proc. IJCAI*, 5554–5561.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning general planning policies from small examples without supervision. In *Proc. AAAI*, 11801–11808.

François-Lavet, V.; Henderson, P.; Islam, R.; Bellemare, M. G.; and Pineau, J. 2018. An introduction to deep reinforcement learning. *Found. Trends. Mach. Learn.*

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing function approximation error in actor-critic methods. In *Proc. ICML*, 1587–1596.

Garg, S.; Bajpai, A.; and Mausam. 2020. Symbolic network: generalized neural policies for relational MDPs. In *Proc. ICML*, 3397–3407.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge U.P.

Goyal, A., and Bengio, Y. 2020. Inductive biases for deep learning of higher-level cognition. *arXiv preprint arXiv:2011.15091*.

Grohe, M. 2021. The logic of graph neural networks. In *Proc. LICS*, 1–17.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *Proc. ICAPS*, 408–416.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proc. ICML*, 1861–1870.

Hamilton, W. L. 2020. Graph representation learning. *Synth. Lect. on Artif. Intell. Mach. Learn.* 14(3):1–159.

Harold, J.; Kushner, G.; and George, Y. 1997. *Stochastic Approximation Algorithms and Applications*. Springer, New York.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artif. Intell.* 143(2):219–262.

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

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *Proc. IJCAI*, 918–923.

Illanes, L., and McIlraith, S. A. 2019. Generalized planning via abstraction: arbitrary numbers of objects. In *Proc. AAAI*, 7610–7618.

Karia, R., and Srivastava, S. 2021. Learning generalized relational heuristic networks for model-agnostic planning. In *Proc. AAAI*, 8064–8073.

Khardon, R. 1999. Learning action strategies for planning domains. *Artif. Intell.* 113:125–148.

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

Kirk, R.; Zhang, A.; Grefenstette, E.; and Rocktäschel, T. 2023. A survey of zero-shot generalisation in deep reinforcement learning. *JAIR* 76:201–264.

Konda, V., and Tsitsiklis, J. 1999. Actor-critic algorithms. *Adv. NIPS* 12.

Küttler, H.; Nardelli, N.; Miller, A.; Raileanu, R.; Selvatici, M.; Grefenstette, E.; and Rocktäschel, T. 2020. The nethack learning environment. *Adv. NIPS* 33.

Levine, S.; Finn, C.; Darrell, T.; and Abbeel, P. 2016. End-to-end training of deep visuomotor policies. *JMLR* 17(1):1334–1373.

Martín, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Appl. Intell.* 20(1):9–19.

Misra, D. 2020. Mish: A self regularized non-monotonic activation function. In *Proc. BMVC*.

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.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*.

Paszke, A., and et. al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Adv. NIPS* 32.

Pearl, J. 2009. *Causality: Models, Reasoning, and Inference (2nd Edition)*. Cambridge University Press.

Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M.; and Dormann, N. 2021. Stable-baselines3: Reliable reinforcement learning implementations. *JMLR* 22:1–8.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*.

Robbins, H., and Monro, S. 1951. A stochastic approximation method. *Ann. Math. Statist.* 22(3):400–407.

Rodriguez, I. D.; Bonet, B.; Romero, J.; and Geffner, H. 2021. Learning first-order representations for planning from black-box states: New results. In *Proc. KR*, 539–548.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2008. The graph neural network model. *IEEE transactions on neural networks* 20(1):61–80.

Schölkopf, B.; Locatello, F.; Bauer, S.; Ke, N. R.; Kalchbrenner, N.; Goyal, A.; and Bengio, Y. 2021. Toward causal representation learning. *Proc. IEEE* 109(5):612–634.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *CoRR* abs/1707.06347.

Segovia, J.; Jiménez, S.; and Jonsson, A. 2016. Generalized planning with procedural domain control knowledge. In *Proc. ICAPS*, 285–293.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning domain-independent planning heuristics with hypergraph networks. In *Proc¿ ICAPS*, 574–584.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Sonar, A.; Pacelli, V.; and Majumdar, A. 2021. Invariant policy optimization: Towards stronger generalization in reinforcement learning. In *Learning for Dynamics and Control*, 21–33.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. AAAI*, 991–997.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proc. ICAPS*, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning generalized policies without supervision using GNNs. In *Proc. KR*, 474–483.

Ståhlberg, S.; Francès, G.; and Seipp, J. 2021. Learning generalized unsolvability heuristics for classical planning. In *Proc. IJCAI*, 4175–4181.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT Press.

Toenshoff, J.; Ritzert, M.; Wolf, H.; and Grohe, M. 2021. Graph neural networks for maximum constraint satisfaction. *Front. Artif. Intell. Appl.* 3:580607.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. Asnets: Deep learning for generalised planning. *JAIR* 68:1–68.

Wang, Z.; Xiao, X.; Xu, Z.; Zhu, Y.; and Stone, P. 2022. Causal dynamics learning for task-independent state abstraction. In *Proc. ICML*.

Williams, R. J., and Peng, J. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science* 3(3):241–268.

Zhang, A.; Lyle, C.; Sodhani, S.; Filos, A.; Kwiatkowska, M.; Pineau, J.; Gal, Y.; and Precup, D. 2020. Invariant causal prediction for block mdps. In *Proc. ICML*, 11214–11224.