Relational Abstractions for Generalized Reinforcement Learning on Symbolic Problems

Rushang Karia, Siddharth Srivastava

School of Computing and Augmented Intelligence, Arizona State University, U.S.A.
Rushang.Karia,siddharths@asu.edu

Abstract

Reinforcement learning in problems with symbolic state spaces is challenging due to the need for reasoning over long horizons. This paper presents a new approach that utilizes relational abstractions in conjunction with deep learning to learn a generalizable Q-function for such problems. The learned Q-function can be efficiently transferred to related problems that have different object names and object quantities, and thus, entirely different state spaces. We show that the learned, generalized Q-function can be utilized for zero-shot transfer to related problems without an explicit, hand-coded curriculum. Empirical evaluations on a range of problems show that our method facilitates efficient zero-shot transfer of learned knowledge to much larger problem instances containing many objects.

1 Introduction

Deep Reinforcement Learning (DRL) has been successfully used for sequential decision making in tasks using image-based state representations (Mnih et al. 2013). However, many problems in the real world cannot be readily expressed as such and are naturally described in factored representations in a symbolic representation language such as the Planning Domain Definition Language (PDDL) (Long and Fox 2003) or the Relational Dynamic Influence Diagram Language (RDDL) (Sanner 2010). For example, in a logistics problem, the objective consists of delivering packages to their destined locations using a truck to carry them. Symbolic description languages such as first-order logic (FOL) can easily capture states and objectives of this scenario using predicates such as *in-truck(p)* where p is a parameter that can be used to represent any package. Symbolic representations for such problems are already available in the form of databases and converting them to image-based representations would require significant human effort. Due to their practical use, symbolic descriptions and algorithms utilizing them are of keen interest to the research community.

A key difficulty in applying RL to problems expressed in such representations is that their state spaces generally grow exponentially as the number of state variables or objects increases. However, solutions to such problems can often be described by compact, easy-to-compute "generalized policies" that can transfer to a class of problems with differing object counts, significantly reducing the sample complexity for learning a good, instance-specific policy.

Running Example We illustrate the benefits of computing generalized policies using the SysAdmin(n) domain (Guestrin, Koller, and Parr 2001) that has been used as a benchmark domain in several planning competitions. A problem in this domain consists of a set of n computers connected to each other in an arbitrary configuration. At any time step, the computers can shutdown with an *unknown* probability distribution that depends on the network connectivity wherein a shutdown computer increases the tendency of its neighbors to shutdown. The agent is also awarded a positive reward that is proportional to the total number of running computers. Similarly, at each time step, the agent may reboot any one of the n computers bearing a small negative reward or may simply do nothing. In our problem setting, action dynamics are not available as closed-form probability distributions making RL the natural choice for solving such problems.

A state in this problem is succintly described by a factored representation with boolean state variables (propositions) that describe which computers are running and their connectivity. It is easy to to see that the state spaces grow exponentially as n increases. However, this problem has a very simple, greedy policy that can provide a very high cumulative reward; reboot any computer that is not running or do nothing. Even though a general policy for such a problem is easy to express, traditional approaches to RL like Q-learning cannot transfer learned knowledge, and thus, have difficulties scaling to larger problems with more computers. Our major contribution in this paper is learning a *generalized, relational Q-function* that can express such a policy and use it to efficiently transfer knowledge to larger instances.

Many existing techniques that compute generalized policies do so by using human-guided or automatic feature engineering to find relevant features that facilitate efficient transfer (see Sec. 5 for a detailed discussion of related work). For example, Ng and Petrick (2021a) use an external feature discovery module to learn first-order features for Q-function approximation. API (Fern, Yoon, and Givan 2006) uses a taxonomic language with beam search to form rule-based policies.

In this paper, we approach the problem of learning generalized policies from a Q-function approximation perspective. We utilize deep learning along with an automatically

generated feature list to learn a nonlinear approximation of the Q-function. Our approach learns a generalizable, relational Q-function that facilitates zero-shot transfer of knowledge to larger instances at the propositional level. We extend our previous work on *leapfrogging*, a data-efficient automatic self-training technique, to RL settings. Our empirical results show that our approach can outperform existing approaches for zero-shot transfer.

The rest of this paper is organized as follows: The next section presents the required background. Sec. 3 describes our approach for transfer followed by a description of our algorithm for generalized reinforcement learning (Sec. 3.3). Sec. 4 presents an extensive empirical evaluation along with a discussion of some limitations and future work. Sec. 5 provides an account of related work in the area and Sec. 6 concludes this paper by summarizing our contributions.

2 Formal Framework

We establish our problem in the context of reinforcement learning for Markov Decision Processes (MDPs). We represent relational MDPs using the notation used by Fern, Yoon, and Givan (2006). Let $D = \langle \mathcal{P}, \mathcal{A} \rangle$ be a problem domain where $\mathcal{P}$ is a set of predicates of arity no greater than 2,[1] and $\mathcal{A}$ is a set of parameterized action names. An MDP problem for a domain D is a tuple $M = \langle O, S, A, T, R, \gamma, s_0 \rangle$ where O is a set of objects. A fact is an instantiation of a predicate $p \in \mathcal{P}$ with the appropriate number of objects from O. A state s is a set of true facts and the state space S is a finite set consisting of all possible sets of true facts. Similarly, the action space A is composed of all possible instantiations of action names $a \in \mathcal{A}$ with objects from O. T is a transition system, implemented by a simulator, that returns a state s' according to some fixed, but *unknown* probability distribution $P(s'|s, a)$ when applying action a in a state s. We assume w.l.o.g. that the simulator only returns actions that are executable in a given state and that there is always one such action (which can be easily modeled using a *no-op*). $R : S \times A \to \mathbb{R}$ is a reward function that is also implemented by the simulator. γ is the discount factor, and s_0 is the initial state.

Example The SysAdmin domain introduced in the preceding section can be described by predicates *running*(c_x) and *connected*(c_x, c_y). The possible actions are reboot(c_x), and no-op(). c_x and c_y are parameters that can be grounded with objects of a specific problem. A state of a problem M_{eg} drawn from SysAdmin(2) with connectivity K_2 using computer names c_0 and c_1 where only c_0 is running can be described as $s_{\text{eg}} = \{$*running*(c_0), *connected*(c_0, c_1), *connected*$(c_1, c_0)\}$. The action space of M_{eg} would consist of actions no-op(), reboot(c_0), and reboot(c_1) with their dynamics implemented by a blackbox simulator.

A solution to an MDP is expressed as a deterministic *policy* $\pi : S \to A$, which is a mapping from states to actions. Let t be any time step, then, given a policy π, the value of taking action a in a state s is defined as the expected return starting from s, executing a, observing a reward r and following the policy thereafter (Sutton and Barto 1998).

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \middle| s_t = s, a_t = a \right]$$

The optimal action-value function (or Q-function) is defined as the maximum expected return over all policies for every $s \in S$ and every $a \in A$; $q_*(s, a) = \max_\pi q_\pi(s, a)$. It is easy to prove that the optimal Q-function function satisfies the *Bellman* equation (expressed in action-value form):

$$q_*(s, a) = \mathbb{E}_{s' \sim T} \left[r_{t+1} + \gamma \max_{a' \in A} q_*(s', a') \middle| s_t = s, a_t = a \right]$$

Reinforcement learning algorithms iteratively improve the Q-function *estimate* $Q(s, a) \approx q_*(s, a)$ by converting the Bellman equation into update rules. Given an observation sequence $(s_t, a_t, r_{t+1}, s_{t+1})$, the update rule for Q-learning (Watkins 1989) to estimate $Q(s_t, a_t)$ is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \delta_t$$

where $\delta_t = r_{t+1} + \gamma \max_{a' \in A} Q(s_{t+1}, a') - Q(s_t, a_t)$ is the temporal difference, TD(0), error, and α is the learning rate. Q-learning has been shown to converge to the optimal Q-function under certain conditions (Sutton and Barto 1998). Q-learning is an *off-policy* algorithm and generally uses an ϵ-greedy exploration strategy, selecting a random action with probability ϵ, and following the greedy policy $\pi(s) = \operatorname{argmax}_a Q(s, a)$ otherwise.

We define a feature f for a domain D as a first-order formula over $\mathcal{P}$ with one free variable. We then define a feature kernel $\phi_f(s)$ for a problem M as the set of objects $o \in O$ that satisfy f in a state $s \in S$. We utilize description logic to derive and express feature kernels building upon the recent work by Bonet, Francès, and Geffner (2019). This is described in Sec. 3.1.

3 Our Approach

Our goal is to compute approximate Q-functions whose induced policies zero-shot generalize to problem instances with differing object counts in a way that allows RL approaches to find good policies with minimal learning. To do so, we automatically generate domain-specific relational abstractions that lift problem-specific characteristics like object names and numbers (Sec. 3.1). Sec. 3.2 describes our method of representing these abstractions as input features to a deep neural network. Finally, Sec. 3.3 and Sec. 3.4 expand on how our algorithm, Generalized Reinforcement Learning (GRL), performs iterative automatic self-training of the deep neural network to learn approximate Q-values of abstract states and uses them for transfer, especially in the zero-shot setting.

3.1 Relational Abstraction

This paper develops a novel approach for learning, expressing, and transferring generalized Q-functions using logic-based features. One challenge in Q-function approximation

[1] Predicates with arity greater than 2 can be easily converted to binary predicates using a simple compilation.

is the selection of a representation language in which it would be possible to express features that provide useful information for the decision making process. Prior work on computing generalized plans and policies considers using logic-based features to develop lifted policy languages for feature synthesis (Khardon 1999; Cumby and Roth 2002; Martín and Geffner 2004; Fern, Yoon, and Givan 2006). Counters derived from logic-based features have been found to be useful for expressing generalized plans (Srivastava, Immerman, and Zilberstein 2008) and can be used to derive measures of progress for proving correctness (Srivastava, Immerman, and Zilberstein 2010; Srivastava et al. 2011, 2015). Bonet, Francès, and Geffner (2019) develop new approaches for learning logic-based features that can express such counters. Karia and Srivastava (2021) use such feature-based counters to learn generalized heuristics for planning.

We now provide a formal description of the general classes of abstraction-based, domain-independent, automatic feature synthesis algorithms that we used to yield counters for RL.

Description Logics (DLs) are a family of representation languages often used for knowledge representation (Baader et al. 2017). We chose DLs since they provide a good balance between expressiveness and tractability in feature expression.

In the relational MDP paradigm, unary predicates $\mathcal{P}_1 \in \mathcal{P}$ and binary predicates $\mathcal{P}_2 \in \mathcal{P}$ of a domain D can be viewed as *primitive concepts* C and *roles* R in DL. DL includes constructors for generating *compound* concepts and roles from primitive ones to form expressive terms. Our feature list F_{DL} consists of concepts and roles formed by using a reduced set of grammar from Bonet, Francès, and Geffner (2019):

$$C, C' \rightarrow \mathcal{P}_1 \mid \neg C \mid C \sqcap C' \mid \forall R.C \mid \exists R.C \mid R = R'$$
$$R, R' \rightarrow \mathcal{P}_2 \mid R^{-1}$$

where $\mathcal{P}_1$ and $\mathcal{P}_2$ represent the primitive concepts and roles, and R^{-1} represents the inverse. $\forall R.C = \{x \mid \forall y\ R(x,y) \wedge C(y)\}$ and $\exists R.C = \{x \mid \exists y\ R(x,y) \wedge C(y)\}$. $R = R'$ denotes $\{x \mid \forall y\ R(x,y) = R'(x,y)\}$. We also use $\mathit{Distance}(c_1, r, c_2)$ features that compute the minimum number of role r steps between two objects satisfying concepts c_1 and c_2 (Francès, Bonet, and Geffner 2021). We found this reduced grammar (that excludes transitive closure) to generate features that facilitated good generalization in our experiments.

We control the total number of features generated by only considering features up to a certain complexity k (a tunable hyperparameter) that is defined as the total number of grammar rules required to generate a feature.

Example The primitive concepts and roles of the SysAdmin domain are $\mathit{running}(c_x)$ and $\mathit{connected}(c_x, c_y)$ respectively. For the running example M_{eg}, the feature kernel for a feature $f_{up} \equiv \mathit{running}(c_x)$ evaluates to the set of objects satisfying it, i.e., $\phi_{f_{up}}(s_{eg}) = \{c_0\}$. This feature can be interpreted as tracking the set of computers that are running (or up). Similarly, $\phi_{f_{con}}(s_{eg}) = \{c_1\}$ for a different feature $f_{con} \equiv \exists \mathit{connected}.\mathit{running}$. f_{con} can be viewed as tracking the set of computers that are connected to at least one running computer. It is easy to see that DL features such as f_{up} and f_{con} capture relational properties of the state and can be applied to problems with differing object names and quantities.

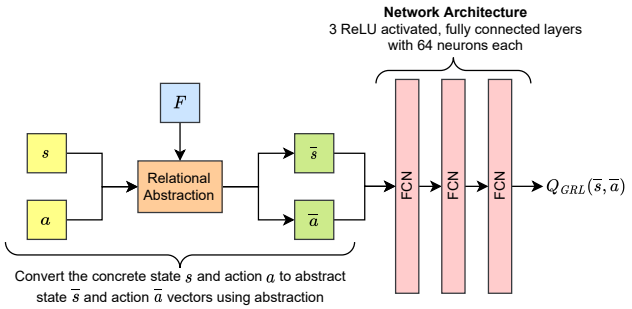

Figure 1: Our process for estimating generalized Q-values.

We implemented a reduced version of the D2L system (Francès, Bonet, and Geffner 2021) for generating such DL based features and describe our process for doing so in Sec. 4.2.

3.2 Deep Learning for Q-value Approximation

Given a list of DL features F, the key challenge is to identify a subset $F' \subseteq F$ of features that can be used to learn a good approximation of the Q-function. We use deep learning, utilizing the entire feature list F for Q-value estimation.

Given a feature list F and a domain D, the input to our network is a vector of size $|F| + |\mathcal{A}| + N \times |F|$ where $|\mathcal{A}|$ is the total number of actions in the domain, and N is the maximum number of parameters of any action in $\mathcal{A}$. Since the dimensionality of the input vector is fixed and does not depend upon a specific problem instance, the same network can be used to predict Q-values across problem instances. This is the key insight into our method for transfer.

Given a concrete state s, the *abstract state feature vector* is defined as $\bar{s} = \langle |\phi_{f_1}(s)|, \ldots, |\phi_{f_n}(s)| \rangle$ for all features $f_i \in F$. Similarly, given a grounded action $a(o_1, \ldots, o_n)$, the *abstract action feature vector* is defined as a vector $\bar{a} = \langle A_{\mathit{name}} | F_{o_1} | \ldots | F_{o_N} \rangle$ where A_{name} is a one-hot vector of length $|\mathcal{A}|$ encoding the action name a, F_{o_i} is a vector of length $|F|$ encoded with values $1_{[o_i \in \phi_{f_j}(s)]}$ for every feature $f_j \in F$, and $|$ represents vector concatenation. The vector $\langle \bar{s} | \bar{a} \rangle$ comprises the input to our network.

Since *Distance* features directly return numerical values, we simply define $|\phi_f(s)| = \phi_f(s)$ and $\forall o \in O, o \in \phi_f(s) = 0$ for such distance-based features.

Example Let $F_{DL} = \langle f_{up}, f_{con} \rangle$ for the SysAdmin domain where $f_{up} \equiv \mathit{running}(c_x)$ and $f_{con} \equiv \exists \mathit{connected}.\mathit{running}$. Then, the abstract state vector $\overline{s_{eg}}$ for the concrete state s_{eg} of the running example would be $\langle 1, 1 \rangle$ and it indicates an abstract state where there is a single computer running and where there is only a single computer that is connected to a running computer. The same vector would be generated for any SysAdmin problem where these properties hold irrespective of the total number of computers or what computer names are used to represent the objects in the state.

Assuming actions are indexed in alphabetical order, for s_{eg}, no-op() would be encoded as $\langle 1, 0|0, 0\rangle$. Similarly, reboot(c_0) and reboot(c_1) would be encoded as $\langle 0, 1|1, 0\rangle$ and $\langle 0, 1|0, 1\rangle$ respectively.

Fig. 1 illustrates our process for estimating the Q-values. Given a concrete state s and action a, our network (that we call Q_{GRL}) predicts the estimated Q-value $Q_{GRL}(\overline{s}, \overline{a}) \approx q_*(s, a)$ by converting s and a to abstract state $\overline{s}$ and action $\overline{a}$ feature vectors based on the feature list F.

The abstract state captures high-level information about the state structure, whereas the abstract action captures the membership of the instantiated objects in the state, allowing our network to learn a generalized, relational Q-function that can be transferred across different problem instances.

3.3 Generalized Reinforcement Learning

Intuitively, Alg. 1 presents our approach for Generalized Reinforcement Learning (GRL). For a given MDP M, an initial Q_{GRL} network, and a list of features F, GRL works as follows: Lines $1-5$ transfer knowledge from the Q_{GRL} network by converting every concrete state s and action a to abstract state $\overline{s}$ and abstract action $\overline{a}$ vectors using the approach in Sec. 3.2. Next, every concrete Q-table entry $Q(s, a)$ is initialized with the predicted value $Q_{GRL}(\overline{s}, \overline{a})$. The Q-table for different problems $M' \neq M$ are different since their state and action spaces are different, however, $\overline{s}$ and $\overline{a}$ are fixed-sized vector representations of any state and action in these problems. This allows Q_{GRL} to transfer knowledge to any problem M' with any number of objects. Lines $9-12$ do Q-learning on M to improve the bootstrapped policy further. Lines $13-16$ further improve the generalization capabilities of Q_{GRL} by incorporating any policy changes that were observed while doing Q-learning on M. GRL returns the task-specific policy Q and the updated generalized policy Q_{GRL}.

Algorithmic Optimization Lines $2-5$ can be intractable for large state spaces. We optimized transfer by only initializing entries in a lazy evaluation fashion, i.e., we start with an empty Q-table and only transfer values for states that do not have an entry in the Q-table. An added benefit is that updates to Q_{GRL} for any abstract state can be easily reflected when encountering a new state that maps to the same abstract state. We empirically observed this to be helpful in improving the sample efficiency for solving the task.

Theorem 3.1. *Solving problem M using GRL converges under standard conditions of convergence for Q-learning.*

Proof (Sketch). The proof is based on the following intuition. Q_{GRL} is used to initialize every $Q(s, a)$ entry of M exactly once after which Q-learning operates as normal. The rest of the proof follows from the proof of convergence for Q-learning (Sutton and Barto 1998). □

3.4 Scaling Up Q-learning and Transfer

Transfer capabilities can often be improved if the training strategy uses a curriculum that organizes the tasks presented to the learner in increasing order of difficulty (Bengio et al. 2009). However, the burden of segregating tasks in order of difficulty often falls upon a domain expert.

Algorithm 1: Generalized Reinforcement Learning (GRL)

Require: MDP M, GRL network Q_{GRL}, feature list F, epsilon ϵ, learning rate α

1: $Q \leftarrow$ initializeEmptyTable()
2: **for** $s \in S, a \in A$ **do**
3: $\quad \overline{s}, \overline{a} \leftarrow$ abstraction(F, s, a)
4: $\quad Q(s, a) = Q_{GRL}(\overline{s}, \overline{a})$
5: **end for**
6: $\mathbb{B} \leftarrow$ initialize replay buffer
7: $s \leftarrow s_0$
8: **while** stopping criteria not met **do**
9: $\quad a \leftarrow$ getEpsilonGreedyAction(s, ϵ)
10: $\quad s', r \leftarrow$ executeAction(s, a)
11: $\quad \delta = r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)$
12: $\quad Q(s, a) = Q(s, a) + \alpha\delta$
13: $\quad \overline{s}, \overline{a} \leftarrow$ abstraction(F, s, a)
14: $\quad$ Add $(\overline{s}, \overline{a}, Q(s, a))$ to $\mathbb{B}$
15: $\quad$ Sample a mini-batch B from $\mathbb{B}$
16: $\quad$ Train Q_{GRL} using B
17: $\quad s \leftarrow s'$ $\{s \leftarrow s_0$ if episode ends$\}$
18: **end while**
19: **return** Q, Q_{GRL}

We extend our work on *leapfrogging*, which is an automatic, data-efficient self-training method (Groshev et al. 2018; Karia and Srivastava 2021), to RL settings. Leapfrogging follows the "learning-from-small-examples" paradigm by using a problem generator to automatically create a curriculum for learning. It is an iterative process for speeding up learning when used in conjunction with a transfer learning algorithm such as GRL. Leapfrogging is analogous to a loose curriculum, enabling self-supervised training in contrast to curriculum learning, which on its own, does not enable automatic self-training.

Leapfrogging operates by initially generating a small problem, M_{small}, that can be easily solved by vanilla Q-learning without any transfer. It applies GRL (Alg. 1) to this problem using an uninitialized Q_{GRL} network. Once this problem is solved, leapfrogging generates a slightly larger problem and invokes GRL again. The Q_{GRL} network learned in the previous iteration allows GRL to utilize knowledge transfer to solve this new problem quickly while also improving the generalization capabilities of the next generation Q_{GRL} network.

4 Empirical Evaluation

We performed an empirical evaluation on four different tasks and our results show that GRL outperforms the baseline in zero-shot transfer performance. We also show that GRL is competitive with approaches receiving additional information in the form of closed-form action models. We now describe the evaluation methodology that we employed for assessing these hypotheses.

We ran our experiments utilizing a single core and 16 GiB of memory on an Intel Xeon E5-2680 v4 CPU containing 28 cores and 128 GiB of RAM.

We used the network architecture from Fig. 1 for all of our experiments. Our system is implemented in Python[2] and we used PyTorch (Paszke et al. 2019) with default implementations of mean squared error (MSE) as the loss function and Adam (Kingma and Ba 2015) as the optimization algorithm for training each domain-specific Q_{GRL} network.

Our system uses RDDLsim as the simulator, and thus, accepts problems written in a subset of RDDL.

4.1 Baselines

As our baseline, we compare our approach with a first-order Q-function approximation based approach for transfer; MBRRL (Ng and Petrick 2021a,b). We also compare with SymNet (Garg, Bajpai, and Mausam 2020), an approach that requires access to closed-form action models, information that is unavailable to MBRRL and GRL in our setting.[3]

MBRRL computes first-order abstractions using conjunctive sets of features and learns a linear first-order approximation of the Q-function over these features. They employ "mixed approximation," where both the concrete $Q(s, a)$ values as well as the approximated Q-values are used to select actions for the policy.

SymNet uses a Graph Neural Network (GNN) representation of a parsed Dynamic Bayes Network (DBN) for a problem. SymNet thus has access to additional domain knowledge in the form of closed-form action models, and as a result, it is not directly applicable in the RL setting that we consider. Nevertheless, it can serve as a good indicator of the transfer capabilities of GRL, which does not need such closed-form representations of action models. We also tried modifying TraPSNet (Garg, Bajpai, and Mausam 2019), a precursor of SymNet that does not require action models, but could not run it due to the limited support for the domains we considered.

4.2 Tasks, Training, and Test Setup

We consider tasks used in the International Probablistic Planning Competition (IPPC) (Sanner 2011, 2014), some of which have been used by SymNet and MBRRL as benchmarks for evaluating transfer performance.

SysAdmin(n) is the IPPC version of the SysAdmin domain that was described earlier in the paper. The IPPC version also allows for computers to automatically restart with a fixed probability p.

Academic Advising(l, c, p) is a domain where the objective is to complete a degree program by passing a certain number of levels l containing c courses, each of which need p prerequisites to be passed first. Courses have a higher probability of passing if all of their prerequisites have been passed first. At each time step, the agent is provided with a large negative reward if the agent has not yet completed the program. Thus, the objective is to complete the program in the shortest number of time steps.

Game of Life(x, y) is John Conway's Game of Life environment on a grid of size $x \times y$ (Izhikevich, Conway, and Seth 2015). The rules are as follows: (a) a live cell with fewer than two or greater than three live neighbors dies, (b) cells with two or three live neighbors live on to the next time step, and (c) any dead cell with exactly three live neighbors becomes a live cell. At each time step, the agent is awarded with a positive reward proportional to the total number of live cells.

Wildfire(x, y) is an environment set in a grid of size $x \times y$ with some cells containing fuel. Cells that contain fuel can spontaneously ignite with a probability proportional to the total number of their neighbors that are on fire. At each time step, the agent is awarded a large negative reward for each cell that is burning. The initial state starts with some cells already on fire and thus the objective is to put out all the fires as quickly as possible.

The IPPC versions of Game of Life and Wildfire contain 4-ary predicates that we converted to an equivalent binary version by converting predicates like *neighbor*(x_1, y_1, x_1, y_2) to *neighbor*(l_{11}, l_{12}) for use with all baselines and GRL.

Domain	Training Sizes: Baselines	Training Sizes: GRL	Test Sizes
SysAdmin	$2^{10} - 2^{20}$	$2^{3} - 2^{6}$	$2^{30} - 2^{50}$
Academic Advising	$2^{20} - 2^{30}$	$2^{8} - 2^{32}$	$2^{40} - 2^{60}$
Game of Life	$2^{9} - 2^{9}$	$2^{4} - 2^{9}$	$2^{16} - 2^{30}$
Wildfire	$2^{18} - 2^{32}$	$2^{8} - 2^{32}$	$2^{50} - 2^{72}$

Table 1: Sizes of the state spaces (min−max) for the problems used in training and testing the baselines and GRL.

Training For SymNet, we utilized the same problems (IPPC instances $1, 2$, and 3) for training as published by the authors. We trained each problem for 1250 episodes. For MBRRL, we utilized the original authors' training procedure wherein we used IPPC instance $\#3$ for training. We trained this problem for 3750 episodes using Q-learning with an initial $\epsilon = 1$ and a decay rate of 0.997.

For training GRL with our leapfrogging approach, we used problem generators that were used in the IPPC to randomly generate problems for training. We used SysAdmin(n) with $n \in \{3, 4, 6\}$, Academic Advising(n, n, n) with $n \in \{2, 3, 4\}$, Game of Life(n, n) with $n \in \{2, 3\}$ and Wildfire(n, n) with $n \in \{2, 3, 4\}$ for generating problems used for training each domain respectively. We trained each problem for 1250 episodes using GRL using a fixed $\epsilon = 0.1$. for each problem.

Testing We used the same set of problems as SymNet: instances $5 - 10$ from the IPPC problem collection. The only exception was Academic Advising where we used instances $5, 7$, and 9 since instances 6 and 10 used concurrent actions and this was not compatible with our system. To better evaluate transfer performance, the test problems are selected such that their state spaces are much larger than the training problems used by GRL. For example, the state space size of SysAdmin(n) is 2^n. For training, the largest problem used by GRL was SysAdmin(6), whereas the largest test problem,

[2]Our code is available at: https://github.com/AAIR-lab/GHN

[3]We thank the authors of SymNet and MBRRL for help in setting up and using their source code.

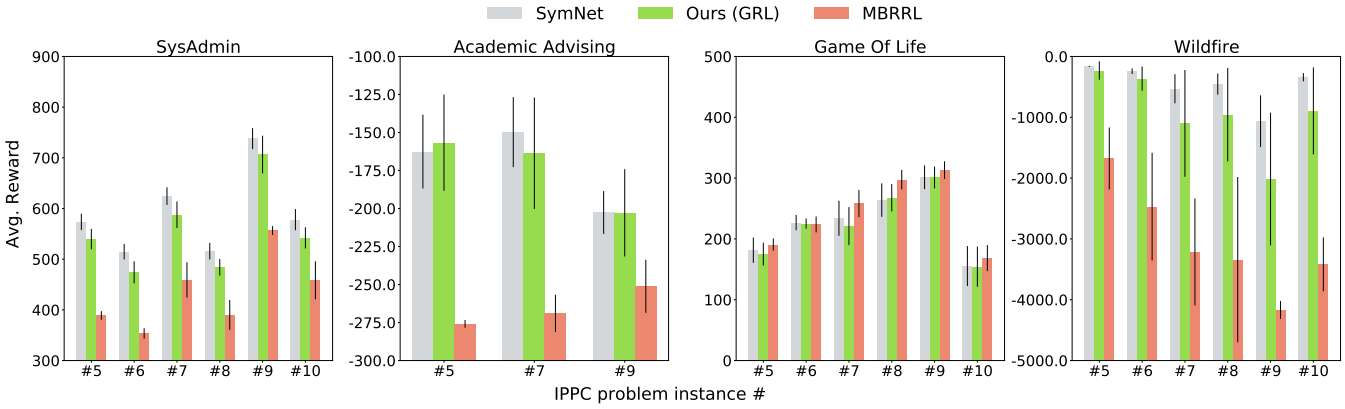

Figure 2: Zero-shot transfer performance of GRL (higher values better) compared to MBRRL when averaged across 10 different runs of 100 episodes each. Error bars indicate one standard deviation. The problem number refers to the instance number in the IPPC problem collection. We also compare with SymNet, an approach that needs closed-form action models and thus is not applicable in our setting. As such, we plot SymNet results as gray bars. We assume minimum reward is obtained when the Q-values are set to NaN as was the case for some problems in Academic Advising for MBRRL. For Academic Advising, we report instances 5, 7 and 9 since instances 6 and 10 contained settings that were incompatible with our system and as a result were not run.

instance 10 (SysAdmin #10) of the IPPC is SysAdmin(50).

Table 1 lists the minimum and maximum state space sizes of the training and test problems that we used in our evaluation respectively. A detailed description of the problem instances and parameters that we used in our training and test sets can be found in the extended version of this paper (Karia and Srivastava 2022).

Hyperparameters We used the IPPC horizon H of 40 time steps for each episode after which the simulator was reset to the initial state. To train a Q_{GRL} network, we used a replay buffer of size 20000, a mini-batch size of 32, and a training interval of 32 time steps with 25 steps of optimization per interval. For our test setup, we used Q-learning with $\epsilon = 0.1$ for GRL and MBRRL. We used $\gamma = 0.9$ and $\alpha = 0.05$ for the SysAdmin and Game of Life domains and used $\gamma = 1.0$ and $\alpha = 0.3$ for Academic Advising and Wildfire.

For MBRRL and SymNet, we used the default values of all other settings like network architecture, feature discovery threshold, etc., that were published by the authors.

Evaluation Metric To showcase the efficacy of transfer learning, our evaluation metric compares the performance of MBRRL and our approach after zero-shot transfer. We freeze the policy after training, transfer it to the test instances and run it greedily for 100 episodes. We report our results using mean and standard deviation metrics computed using 10 individual runs of training and testing.

Feature Generation We generated sampled state spaces and transitions between states using random walks on the first problem used by GRL for training per domain. These transitions were used in conjunction with the DL grammar from Sec. 3.1 to generate the DL feature list F_{DL} using a modified version of the D2L system (Francès, Bonet, and Geffner 2021), which does not require closed-form action models or knowledge of goals. We set a complexity bound of $k = 5$ for *goal-independent* feature generation. We empirically observed that using this single *small* problem instance and complexity bound together with the reduced DL grammar from Sec. 3.1 to generate F_{DL} was sufficient in generating features that enabled good generalization using GRL.

4.3 Analysis of Results

Our results are shown in Fig. 2. It is easy to see that GRL has excellent zero-shot transfer capabilities and can easily outperform or remain competitive with both MBRRL and SymNet. We now present our analysis followed by a brief discussion of some limitations and future work.

Comparison with MBRRL Our approach is able to significantly outperform MBRRL on SysAdmin, Academic Advising, and Wildfire. The DL abstractions used by GRL are more expressive than the conjunctive first-order features used by MBRRL, allowing GRL to learn policies that are more expressive. Additionally, leapfrogging allows scaling up training and learning of better generalized policies in the same number of training episodes in contrast to using a fixed instance for training.

For the Game of Life domain, we observed that even a random policy performs similarly to GRL and the baselines. This is surprising since PROST (Keller and Eyerich 2012), an approach that requires closed-form action models, has demonstrated that it is possible to achieve high reward in this domain (Sanner 2011). We leave this investigation to future work.

Comparison with SymNet SymNet utilizes significant domain knowledge in constructing the graphs. For example, edges are added between two nodes iff an action affects them. Such information is unavailable when just observing states as sets of predicates. It is impressive that despite not using such knowledge, GRL is able to remain competitive with SymNet in most of the problems.

4.4 Limitations and Future Work

In the SysAdmin domain, the probability with which a computer shuts down depends on how many shutdown computers it is connected to. Our representation of $o \in \phi_f(s)$ for representing the action vectors cannot capture such dependencies. However, this is easy to mitigate using a new feature that counts the number of shutdown computers a specific computer is connected to. We plan to investigate the automatic generation and use of such features in future work.

Leapfrogging requires an input list of object counts for the problem generator that we hand-coded. However, we believe that our approach is a step forward in curriculum design by relieving the designer from knowing intrinsic details about the domain, which is often imperative for assessing the difficulty of tasks. The lack of a problem generator can be mitigated by combining leapfrogging with techniques that sample "subgoals" (Fern, Yoon, and Givan 2006; Andrychowicz et al. 2017) and utilizing GRL to learn a generalized policy that can later be efficiently transferred to any subsequent problems.

5 Related Work

Our work adds to the vast body of literature on learning in relational domains. Several of these approaches (Khardon 1999; Guestrin et al. 2003; Wu and Givan 2007; Garg, Bajpai, and Mausam 2020) assume that action models are available in an analytical form and thus are not directly applicable to RL settings. For example, FOALP (Sanner and Boutilier 2005) learns features for approximating the value function by regressing over action models. D2L (Francès, Bonet, and Geffner 2021) learns abstract policies for deterministic problems assuming an action model where actions can increment or decrement features. We focus our discussion on relational RL (see Tadepalli, Givan, and Driessens (2004) for an overview).

Q-estimation Approaches Q-RRL (Dzeroski, Raedt, and Driessens 2001) learns an approximation of the Q-function by using logical regression trees. GBQL (Das et al. 2020) learns a gradient-boosted tree representation of the Q-function. Their tree representations use "lifted" predicates thus enabling transfer across problem instances with differing object counts. These approaches were evaluated on relatively simple tasks using hand-coded *support* predicates, demonstrating the difficulty of transfer using a tree-based approach. Our evaluation does not use any support predicates and shows that GRL can learn good policies that transfer well to larger problems.

RePReL (Kokel et al. 2021) uses a high-level planner together with hand-coded abstractions to train task-specific RL agents for transfer learning. Rosenfeld, Taylor, and Kraus (2017) use hand-crafted features and similarity functions to speed up Q-learning. MBRRL (Ng and Petrick 2021a,b) learns conjunctive first-order features for Q-function approximation using hand-coded contextual information for improved performance. In contrast to these approaches, GRL does not require any hand-coded or expert knowledge.

Policy-based approaches Fern, Yoon, and Givan (2006) use taxonomic syntax with beam search and approximate policy iteration to learn decision-list policies. They sample sub-goals using random walks to scale up learning. However, it is not clear how to apply their approach to problems that do not have goals. GRL can work on problems with or without goals. Janisch, Pevný, and Lisý (2020) use graph neural network (GNN) representations of the state to compute policies. GNNs are reliant on the network's receptive field unlike Q_{GRL} which uses multilayer perceptrons and thus have limited generalization capabilities w.r.t. the number of objects. TraPSNet (Garg, Bajpai, and Mausam 2019) also uses a GNN and is limited to domains with a single binary predicate and actions with a single parameter. Moreover, the binary predicate in TraPSNet is required to be a *non-fluent* meaning that its truth value in a problem can never change. GRL can be used in domains with any number of action parameters and binary predicates and allows binary predicates to change their valuations across different states.

Automatic Curriculum Generation Fern, Yoon, and Givan (2006) sample goals from random walks on a single problem. Their approach relies on the target problem to adequately represent the goal distribution for generalization. Similar ideas are explored in Ferber, Helmert, and Hoffmann (2020) and Andrychowicz et al. (2017). These techniques are *intra-instance*, sampling different goals from the same state space and are orthogonal to GRL, that addresses *inter-instance* transfer. Our approach of leapfrogging is most similar to that of Groshev et al. (2018) and Karia and Srivastava (2021) and the learn-from-small-examples approach of Wu and Givan (2007). We extend their ideas to RL settings and demonstrate its efficacy in transfer. Narvekar and Stone (2019) automatically generate a curriculum for tasks by solving a curriculum MDP (CMDP). However, they require domain-dependent, hand-coded basis features for effective curriculum learning whereas leapfrogging does not require any domain-dependent hand-coding.

6 Conclusion

We presented an approach for reinforcement learning in relational domains that can learn good policies with effective zero-shot transfer capabilities. Our results show that Description Logic based features acquired simply through state trajectory sequences can offer performance similar to that of of analytical (closed-form) action models. In the future, we plan to investigate improving the features so that abstract actions can also take into account relationships between the instantiated parameters and the abstract state.

Acknowledgements

We would like to thank the Research Computing Group at Arizona State University for providing compute hours for our experiments. This research was supported in part by the NSF under grant IIS 1942856.

References

Andrychowicz, M.; Crow, D.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Abbeel, P.; and Zaremba, W. 2017. Hindsight Experience Replay. In *NeurIPS*.

Baader, F.; Horrocks, I.; Lutz, C.; and Sattler, U. 2017. *An Introduction to Description Logic*. Cambridge University Press. ISBN 978-0-521-69542-8.

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *ICML*.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *AAAI*.

Cumby, C.; and Roth, D. 2002. Learning with Feature Description Logics. In *ILP*.

Das, S.; Natarajan, S.; Roy, K.; Parr, R.; and Kersting, K. 2020. Fitted Q-Learning for Relational Domains. *arXiv*, abs/2006.05595.

Dzeroski, S.; Raedt, L. D.; and Driessens, K. 2001. Relational Reinforcement Learning. *Mach. Learn.*, 43(1/2): 7–52.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Reinforcement Learning for Planning Heuristics. In *ICAPS PRL workshop*.

Fern, A.; Yoon, S. W.; and Givan, R. 2006. Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes. *J. Artif. Intell. Res.*, 25: 75–118.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning General Planning Policies from Small Examples Without Supervision. In *AAAI*.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*.

Garg, S.; Bajpai, A.; and Mausam. 2020. Symbolic Network: Generalized Neural Policies for Relational MDPs. In *ICML*.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing Plans to New Environments in Relational MDPs. In *IJCAI*.

Guestrin, C.; Koller, D.; and Parr, R. 2001. Max-norm Projections for Factored MDPs. In *IJCAI*.

Izhikevich, E. M.; Conway, J. H.; and Seth, A. 2015. Game of Life. *Scholarpedia*, 10(6): 1816.

Janisch, J.; Pevný, T.; and Lisý, V. 2020. Symbolic Relational Deep Reinforcement Learning based on Graph Neural Networks. *arXiv*, abs/2009.12462.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*.

Karia, R.; and Srivastava, S. 2022. Relational Abstractions for Generalized Reinforcement Learning on Symbolic Problems. *arXiv*, abs/2204.12665.

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *ICAPS*.

Khardon, R. 1999. Learning Action Strategies for Planning Domains. *Artif. Intell.*, 113(1-2): 125–148.

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

Kokel, H.; Manoharan, A.; Natarajan, S.; Ravindran, B.; and Tadepalli, P. 2021. RePReL: Integrating Relational Planning and Reinforcement Learning for Effective Abstraction. In *ICAPS*.

Long, D.; and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *J. Artif. Intell. Res.*, 20: 1–59.

Martín, M.; and Geffner, H. 2004. Learning Generalized Policies from Planning Examples Using Concept Languages. *Appl. Intell.*, 20(1): 9–19.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv*, abs/1312.5602.

Narvekar, S.; and Stone, P. 2019. Learning Curriculum Policies for Reinforcement Learning. In *AAMAS*.

Ng, J. H. A.; and Petrick, R. 2021a. First-Order Function Approximation for Transfer Learning in Relational MDPs. In *ICAPS PRL workshop*.

Ng, J. H. A.; and Petrick, R. 2021b. Generalised Task Planning with First-Order Function Approximation. In *CoRL*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

Rosenfeld, A.; Taylor, M. E.; and Kraus, S. 2017. Speeding up Tabular Reinforcement Learning Using State-Action Similarities. In *AAMAS*.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf. Accessed: 2022-01-01.

Sanner, S. 2011. The 2011 International Probabilistic Planning Competition. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/index.html. Accessed: 2022-01-01.

Sanner, S. 2014. The 2014 International Probabilistic Planning Competition. http://users.cecs.anu.edu.au/~ssanner/IPPC_2014/index.html. Accessed: 2022-01-01.

Sanner, S.; and Boutilier, C. 2005. Approximate Linear Programming for First-order MDPs. In *UAI*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning Generalized Plans Using Abstract Counting. In *AAAI*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2010. Computing Applicability Conditions for Plans with Loops. In *ICAPS*.

Srivastava, S.; Zilberstein, S.; Gupta, A.; Abbeel, P.; and Russell, S. 2015. Tractability of Planning with Loops. In *AAAI*.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative Numeric Planning. In *AAAI*.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement learning - an introduction*. MIT Press. ISBN 978-0-262-19398-6.

Tadepalli, P.; Givan, R.; and Driessens, K. 2004. Relational Reinforcement Learning: An overiew. In *ICML RRL workshop*.

Watkins, C. 1989. *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK.

Wu, J.; and Givan, R. 2007. Discovering Relational Domain Features for Probabilistic Planning. In *ICAPS*.