Extending Graph Neural Networks for Generalized Stochastic Planning

Ziqi Zhang, Florian Geißer
The Australian National University

Abstract

Probabilistic planning problems are often formulated in terms of a domain class that describes the general problem structure, and in terms of an instantiation of the domain, which yields a particular MDP. Most algorithms in probabilistic planning focus on computing policies for a single MDP.

A relational policy represents a solution to any MDP induced from the same domain class. We present a graph neural network architecture that is based on a representation of the relation between types of a given domain and which allows us to generalize from small instances to large instances of the same domain class. Unlike other work, we do not impose restrictions on the structure of the domain. We conduct a preliminary study which compares the relational policies obtained from a network trained on small instances against a policy computed by a state-of-the-art domain-independent planner. The evaluation shows that the network generalizes well across instances of a domain, and is even able to outperform the instance-dependent policy in some of the benchmarks.

Introduction

Probabilistic decision making problems allow to model an agent navigating an inherent uncertain world. Markov decision processes (MDPs) (Puterman 1994) are a well-known problem formalism for such problems, where we aim for a policy, i.e. a mapping from states to actions, that maximises some optimisation criterion, usually the average expected reward. A vast amount of research inspired from various fields of AI is concerned with finding 'good' or even optimal policies. Popular solution approaches are heuristic search algorithms such as LAO* (Hansen and Zilberstein 2001) and LRTDP (Bonet and Geffner 2003), tree-search algorithms such as MCTS (Browne et al. 2012) and UCT (Kocsis and Szepesvári 2006), hybrids that combine systematic heuristic search with the exploit/explore property of tree search such as UCT*(Keller and Helmert 2013), or hybrids that combine search and learning such as AlphaZero (Silver et al. 2017). A common property of these approaches is that they are defined on a problem-independent level and thus applicable to a large variety of problems. Yet, the solution that is computed is instance-dependent, i.e. the policy that is good for one problem is not necessarily good for another problem, even if both problems belong to the same domain class.

Recently, there has been an increased interest in finding *generalized policies*, i.e. a policy that is applicable to every problem of a given domain class. This is partially inspired from the field of planning, where the input to planning algorithms is a domain file, specifying the general problem structure, and an instance file, that fixes free variables in the domain to a specific instance. Examples of modeling languages that support this approach are the classical planning modeling language PDDL (McDermott 2000), its probabilistic counterpart PPDDL (Younes and Littman 2004), and the dynamic Bayesian net inspired modeling language RDDL (Sanner 2010). Different work concerned with generalized policies supports different modeling languages and has its focus on different aspects of generalization. For example, the STRIPS-HGN architecture of Shen, Trevizan, and Thiébaux (2020) aims to represent value functions that allow to generalize across domains. Action schema networks (Toyer et al. 2018) take the lifted representation of SSPs as input and compute a generalized policy applicable to all instantiated SSPs of that representation. In particular, they exploit that actions of different instances are instantiations of a common action schema. SymNet (Garg, Bajpai, and Mausam 2020) aims to compute generalized policies for problems specified in RDDL and its generalized policy is based on the information hidden in the dynamic Bayesian net that underlies each RDDL model. The work by Garg, Bajpai, and Mausam (2019) is the most closest related to our work. TraPSNet computes a generalized policy based on the first-order representation of an MDP. However, a significant limitation of the approach is that they assume that all fluents are unary, except one non-fluent that is binary.

Our work can be seen as a generalization of TraPSNet to arbitrary fluent size. Given a first-order MDP, we construct the domain graph which represents types and relations between types of a domain. The instance graph shares the same topology as the domain graph, but is induced from a given instance of the domain. Like aforementioned works we use graph neural networks (Battaglia et al. 2018) to represent relational policies. The domain graph determines the architecture of the graph neural network, while the instance graph serves as input to the network. This allows us to train the network on small-sized instances and use the trained network to compute a policy for arbitrary larger instances. We conduct a preliminary study and compare our relational policies against policies computed by the PROST planner (Keller and Eyerich 2012), the winner of the international probabilistic

planning competition 2014. Depending on the domain, the relational policy obtained from training on small instances for a moderate amount of time can outperform the policy computed by PROST.

Background

An MDP (Puterman 1994) is a tuple $\mathcal{M} = \langle \mathcal{S}, A, \mathcal{P}, \mathcal{R} \rangle$, where $\mathcal{S}$ is a finite set of states, A is a finite set of actions, $\mathcal{R} : \mathcal{S} \times A \to \mathbb{R}$ is the reward function, and the transition function $\mathcal{P} : \mathcal{S} \times A \times \mathcal{S} \to [0, 1]$ defines the probability $\mathcal{P}(s'|s, a)$ that applying action a in state s leads to state s'. We define the set of successors of state s and action a as $\mathsf{succ}(s, a) = \{s' \in \mathcal{S} | \mathcal{P}(s'|s, a) > 0\}$. We say action a is applicable in state s iff $\mathsf{succ}(s, a) \neq \varnothing$ and denote the set of applicable actions in s as $A(s)$.

In a *finite-horizon MDP* the horizon $H \in \mathbb{N}$ limits the number of action applications. A common approach (Mausam and Kolobov 2012) is to augment the state space such that the number of remaining steps is part of a state, denoted by $s[h]$ for $s \in \mathcal{S}$. We have $\mathcal{P}(s'|s, a) = 0$ if $s[h] \neq s'[h] - 1$, to enforce that the number of remaining steps decreases by one in each transition. We further assume that $A(s) \neq \varnothing$ for all $s \in \mathcal{S}$ to ensure that there are no dead-ends. A *terminal state* is a state with $s[h] = 0$, and s_I specifies the *initial state*.

A solution to an MDP is a policy $\pi : \mathcal{S} \to A$, i.e. a mapping from states to actions. The *expected reward* of π is given by the *state-value function* $V^\pi(\mathcal{M}) = V^\pi(s_I)$ with

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \text{ is terminal,} \\ Q^\pi(s, \pi(s)) & \text{otherwise,} \end{cases}$$

where $Q^\pi(s, a) = \mathcal{R}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) \cdot V^\pi(s')$ is the *action-value function*. An optimal policy $\pi^\star$ is a solution to the well-known Bellman optimality equation (Bellman 1957):

$$V^*(s) = \begin{cases} 0 & \text{if } s \text{ is terminal,} \\ \max_{a \in A} Q^*(s, a) & \text{otherwise,} \end{cases}$$
$$Q^* = \mathcal{R}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) \cdot V^*(s').$$

A *factored MDP* is a compact description of an MDP which describes states and actions in terms of state variables $\mathcal{V}$ and action variables $\mathcal{A}$, each associated with a finite domain $\mathcal{D}_\mathsf{v} \subset \mathbb{Q}$ for $\mathsf{v} \in \mathcal{V} \cup \mathcal{A}$. A *partial state assignment* is a partial function $s : \mathcal{V} \hookrightarrow \bigcup_{\mathsf{v} \in \mathcal{V}} \mathcal{D}_\mathsf{v}$, such that $s(\mathsf{v}) \in \mathcal{D}_\mathsf{v}$ for all $\mathsf{v} \in \mathcal{V}$ where s is defined. If s assigns a value to each $\mathsf{v} \in \mathcal{V}$ we call s simply a state. *Partial action assignments* are defined analogously, and actions correspond to full action assignments.

In this work we consider first-order representations of factored MDPs. A *Relational Markov Decision Process* (RMDP) (Garg, Bajpai, and Mausam 2020) is a tuple $\mathcal{R} = \langle \mathsf{T}, \mathsf{S}, \mathsf{A}, \mathsf{N}, \mathsf{P}, \mathsf{R} \rangle$, where T is a set of *types*, S, A and N are sets of predicate symbols denoted as *state fluents*, *action fluents*, and *static fluents*, respectively; we denote the set of all fluents[1] as $\mathsf{F} = \mathsf{S} \cup \mathsf{A} \cup \mathsf{N}$. The lifted transition function P and the lifted reward function R describe the dynamics of the first-order MDP. Predicate symbols denote relations of and between *objects*. Given a set of objects O, the function $\mathsf{type} : \mathsf{O} \to \mathsf{T}$ associates an object with its type. Each predicate symbol $\mathsf{f} \in \mathsf{F}$ takes a tuple $\langle \mathsf{o}_1, \ldots, \mathsf{o}_n \rangle$ of type-consistent objects as arguments, i.e. f is associated with a tuple of types $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$, and each argument o_i is associated with t_i. An object o is then a valid argument for o_i if $\mathsf{type}(\mathsf{o}) = \mathsf{t}_i$. We denote the tuple of arguments for f with $\mathsf{arg}(\mathsf{f})$. We say a predicate symbol is *unparameterized* if it takes no arguments as input. Applying predicates to type-consistent objects is known as the process of *grounding* and yields a factored MDP. The set of state variables $\mathcal{V}$ consists of the application of all $\mathsf{f} \in \mathsf{S}$ to all type-consistent objects O. Analogously, the set of action variables $\mathcal{A}$ consists of the application of $\mathsf{f} \in \mathsf{A}$. The initial state value of each variable is obtained from a partial initial state description $\hat{s_I} : \mathsf{F} \to \mathbb{Q}$, which assigns the same value to those variables that are induced by a common predicate. Static fluents yield static state variables whose initial domain value is fixed and does not change throughout planning. Thus, given a RMDP $\mathcal{R}$, a set of typed objects O, and an initial state description $\hat{s_I}$ the process of grounding induces a factored MDP, where state and action variables are obtained by grounding the relational MDP, transition and reward functions are obtained by application of the lifted transition and reward function, and the initial state is given by $\hat{s_I}$. For simplification, we consider the specific process of grounding as a black-box that takes a set of objects O and a predicate symbol $\mathsf{f} \in \mathsf{F}$ and generates all valid applications of f to O, i.e. a set of tuples, such that each tuple is consistent with $\mathsf{arg}(\mathsf{f})$. Additionally, the grounding process may prune irrelevant tuples. We are not interested in the details of the grounding process, and instead only consider the resulting set of tuples (instead of explicit variables), which we denote as $\mathsf{ground}(\mathsf{f}, \mathsf{O})$.

Often, a relational MDP is called a *domain*, and the factored MDP resulting from grounding $\mathcal{R}$ with a set of objects O and initial state description $\hat{s_I}$ is called an *instance*. Most probabilistic planning algorithms are defined on the instance level, i.e. the algorithm computes a policy for an MDP $\mathcal{M}$. Planners relying on such algorithms receive a *domain description* which corresponds to the RMDP $\mathcal{R}$, and an *instance description* which corresponds to the set of objects in this instance and the initial state description. Before planning starts, the planner then performs grounding which results in the MDP $\mathcal{M}$. The aim of this paper is to compute a *generalized policy* for a domain which yields a policy $\pi^\mathcal{M}$ for any instance $\mathcal{M}$ of this domain. Optimally, we would like $\pi^\mathcal{M} = \pi^\star$.

Throughout the paper we will use the elevator domain as a running example. In this domain, a number of elevators will be controlled to deliver passengers who arrive randomly at each floor to a destination floor. In this domain, the *types* T are {*floor*, *elevator*}. For

[1] In the literature, the term fluents is sometimes used to denote state and action variables of a factored MDP. In this work we use the term 'fluents' to denote first-order predicates instead.

simplicity, we denote predicates together with their argument tuples. Then, the *state fluents* S are {*person-waiting-at-floor(elevator, floor)*, *elevator-at-floor(elevator, floor)*} which are used to specify people's and elevators' states. The *action fluents* A are {*go-up(elevator)*, *go-down(elevator)*, *open-door(elevator)*, *close-door(elevator)*} which are used to control one of the elevators. And the *static fluents* N are {*TOP-FLOOR(floor)*, *BOTTOM-FLOOR(floor)*, *ADJACENT-FLOORS(floor, floor)*} which are used to specify the topology of the floors. The reward function R is such that there are costs if some person is waiting for an elevator at some floor, and the transition function P is used to specify the dynamics of the elevators. Given a set of objects $\mathsf{O} = \{e_0, f_0, f_1, f_2\}$ such that $\mathsf{type}(e_0) =$ *elevator* and $\mathsf{type}(f_0) = \mathsf{type}(f_1) = \mathsf{type}(f_2) =$ *floor*, we get $\mathsf{ground}(\textit{elevator-at-floor}, \{e_0, f_0, f_1, f_2\}) = \{\langle e_0, f_0\rangle, \langle e_0, f_1\rangle, \langle e_0, f_2\rangle\}$. As mentioned, grounding may remove irrelevant argument tuples. For example, $\langle f_0, f_0\rangle$, is a possible argument of *ADJACENT-FLOORS* but may be pruned if the initial state description only specifies the topology between f_0, f_1, and f_2, i.e. the grounding process may realize that floor f_0 is not adjacent to itself.

Graph Neural Networks

Graph neural networks (Battaglia et al. 2018) define functions over graph representations $\mathsf{G} = (\mathsf{V}, \mathsf{E})$. Our definition slightly differs from Battaglia et al. to fit the context of the paper. Each vertex $v \in \mathsf{V}$ is associated with an embedding vector $h_0^v \in \mathbb{R}^{d_0^v}$ of dimension d_0^v. The graph neural network (GNN) takes G as input and iteratively updates the embedding of each vertex v over K steps. More specifically, at each forward step $k \in \{1, \dots, K\}$ the embedding of vertex v at step k is defined as $h_k^v = \phi^{\theta_1}(h_{k-1}^v \| \rho^{\theta_2}(E_v, k-1))$ where $\|$ denotes vector concatenation, E_v denotes the set of edges connecting to v, ϕ^{θ_1} is an update function parameterized by weights θ_1, and ρ^{θ_2} is an aggregation function parameterized by weights θ_2 whose output is a single aggregated vector.

Graph neural networks allow us to define update and aggregation function on the domain level, while the network input and therefore each update step operates on the domain level. As a result, we can train the network with input from small-sized instances and use it to compute a policy for large instances. The following sections are concerned with the definition of the graphs that serve as input, and the resulting network architecture which allows to represent a general policy.

Graph Representations of RMDPs

We first describe the domain and instance graph, which we will use to construct a graph neural network architecture. Given a RMDP $\mathcal{R} = \langle \mathsf{T}, \mathsf{S}, \mathsf{A}, \mathsf{N}, \mathsf{P}, \mathsf{R}\rangle$ the *domain graph* $\mathsf{G_d} = (\mathsf{V_T} \cup \mathsf{V_F}, \mathsf{E_d})$ is a bipartite undirected edge-labeled multigraph that shows relations between types in a domain. Vertices are partitioned into two sets: *Type vertices* $\mathsf{V_T}$ correspond to types. There is a vertex for each type $\mathsf{t} \in \mathsf{T}$ and an additional *master type vertex* denoted as $\mathsf{master_d}$. We will sometimes abuse notation and refer with t to the type vertex associated with t if it is clear from the context, thus $\mathsf{V_T} = \mathsf{T} \cup \{\mathsf{master_d}\}$. *Type relation vertices* $\mathsf{V_F}$ correspond to argument tuples of predicate symbols F. Let $\mathsf{Args} = \{\mathsf{arg(f)} | \mathsf{f} \in \mathsf{F}\}$ be the set of all predicate argument tuples. Then, there is a relation vertex for each $\mathsf{a} \in \mathsf{Args}$. Additionally, there is a relation vertex $(\mathsf{t}, \mathsf{master_d})$ for each $\mathsf{t} \in \mathsf{T}$, and finally a single relation vertex $(\mathsf{master_d})$, which will be used to represent unparameterized fluents. By abuse of notation we thus have $\mathsf{V_F} = \mathsf{Args} \cup \bigcup_{\mathsf{t}\in\mathsf{T}}(\mathsf{t}, \mathsf{master_d}) \cup \{(\mathsf{master_d})\}$.

Edges connect relation vertices to type vertices. Let v_F be a relation vertex associated with argument tuple $\langle \mathsf{t}_1, \dots, \mathsf{t}_n\rangle$ and v_t be a type vertex associated with type t. Then, for each $i \in \{1, \dots, n\}$ there is an edge between v_F and v_t labeled with i if and only if $\mathsf{t} = \mathsf{t}_i$. Observe that this implies that $\mathsf{G_d}$ may be a multigraph, i.e. two vertices may be connected by multiple edges. Additionally, for each $\mathsf{t} \in \mathsf{t}$ there is an edge labeled with 1 between v_t and $(\mathsf{t}, \mathsf{master_d})$ and an edge labeled with 2 between $\mathsf{master_d}$ and $(\mathsf{t}, \mathsf{master_d})$. Finally, there is an edge labeled with 1 between $\mathsf{master_d}$ and $(\mathsf{master_d})$.

While the domain graph is based on the first-order representation of an MDP, the *instance graph* $\mathsf{G_i} = (\mathsf{V_O} \cup \mathsf{V_{ground}}, \mathsf{E_i})$ represents the structure of a particular problem instance and thus shows relations between objects in the grounded representation. Again, vertices are partitioned into two sets: *object vertices* and *object relation vertices*. Object vertices $\mathsf{V_O}$ correspond to objects, and there is a vertex for each object $\mathsf{o} \in \mathsf{O}$, and again an additional *master object vertex*, which will be denoted by $\mathsf{master_i}$. Again, we may refer with o to the object vertex associated with o, thus $\mathsf{V_O} = \mathsf{O} \cup \{\mathsf{master_i}\}$. Object relation vertices $\mathsf{V_{ground}}$ correspond to grounded argument tuples of predicate symbols F. Recall that $\mathsf{ground(f, O)}$ corresponds to the set of grounded argument tuples for f. Let $G = \{\mathsf{ground(f, O)} | \mathsf{f} \in \mathsf{F}\}$ be the set of all grounded argument tuples. There is a vertex for each tuple in G. Additionally, there is a vertex $(\mathsf{o}, \mathsf{master_i})$ for each $\mathsf{o} \in \mathsf{O}$ and finally single object relation vertex $(\mathsf{master_i})$. By abuse of notation we thus have $\mathsf{V_{ground}} = \{\mathsf{ground(f, O)} | \mathsf{f} \in \mathsf{F}\} \cup \bigcup_{\mathsf{o}\in\mathsf{O}}(\mathsf{o}, \mathsf{master_i}) \cup \{(\mathsf{master_i})\}$. In the instance graph, edges connect relation vertices to object vertices. Let v_ground be a relation vertex associated with grounded argument tuple $\langle \mathsf{o}_1, \dots, \mathsf{o}_n\rangle$ and v_o be an object vertex associated with object o. Then, for each $i \in \{1, \dots, n\}$ there is an edge labeled with i between v_ground and v_o if and only if $\mathsf{o} = \mathsf{o}_i$. Again, observe that this may result in multiple edges between v_ground and v_o. Edges between v_o and $(\mathsf{o}, \mathsf{master_i})$, respectively $(\mathsf{o}, \mathsf{master_i})$ and $(\mathsf{master_i})$, are defined analogously to their corresponding edges in the domain graph.

Informally, the instance graph corresponds to a grounded form of the domain graph. We can express this formally by making use of the type function that maps objects to their types and we define a graph homomorphism $\mathsf{map} : \mathsf{V_O} \cup \mathsf{V_{ground}} \to \mathsf{V_T} \cup \mathsf{V_F}$ in the following way:

1. $\mathsf{map}(\mathsf{master_i}) = \mathsf{master_d}$,
2. $\mathsf{map}(\mathsf{o}) = \mathsf{type}(\mathsf{o})$,
3. $\mathsf{map}((\mathsf{o}, \mathsf{master_i})) = (\mathsf{type}(\mathsf{o}), \mathsf{master_d})$;
4. Let v_ground be an object relation vertex associated with grounded argument tuple $\langle \mathsf{o}_1, \dots, \mathsf{o}_n\rangle$. Then $\mathsf{map}(v_\mathsf{ground}) = v_\mathsf{F}$, where v_F is the type relation

vertex associated with argument tuple $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ and $\mathsf{type}(\mathsf{o}_i) = \mathsf{t}_i$ for $i \in \{1, \ldots, n\}$.

Consider again the elevator domain with the set of objects $\{e_0, f_0, f_1, f_2\}$. The left-hand side of Figure 1 shows a subgraph of the domain graph, and the corresponding subgraph of the instance graph on the right-hand side. Type consistency is indicated by colours and the predicates inducing the argument tuples are annotated for the type relation and object relation vertices. Note that in the instance subgraph, the vertex f_1 connects to the vertex $\langle f_0, f_1 \rangle$ with a 2-labeled (purple dashed) edge, but to the vertex $\langle f_1, f_2 \rangle$ with a 1-labeled (cyan) edge. Intuitively, the label of an edge represents the argument index of the object in the corresponding object relation.

Network Architecture

The previously introduced graphs serve as the basis for our graph neural network. The underlying idea is that the domain graph determines the architecture of the graph neural network, while the instance graph serves as input to the network. For this, every vertex v in the instance graph is associated with an initial real-valued embedding vector h_0^v and the embedding at step $k+1$ is computed by a forward pass over the network at step k, defined formally as $h_{k+1}^v = \phi(h_k^v \| \rho(E_v, k-1))$. We will now define the initial embedding h_0^v, and update and aggregation functions ϕ and ρ. The final embedding at step $K+1$ will then be used by a decoder module to compute a policy for the MDP.

Instance vertex embedding

Let $\mathsf{G_i} = (\mathsf{V_O} \cup \mathsf{V_{ground}}, \mathsf{E_i})$ be an instance graph. For every vertex $v \in \mathsf{V_O} \cup \mathsf{V_{ground}}$ and every $k \in \{0, \ldots, K+1\}$ we define h_k^v as the *embedding* of v at step k, where h_k^v is a real-valued vector with dimension d_v^k. While two embeddings may have a different dimension, we emphasize that the dimension only depends on the domain, the type of the vertex, and the current step k. In particular, the dimension is independent of the instance description.

For object vertices $v \in \mathsf{V_O}$ we define the initial embedding h_0^v as an empty vector of dimension 0. The initial embedding of object relation vertices $v \in \mathsf{V_{ground}}$ depends on the type of v:

1. If $v = (\mathsf{master_i})$, the initial embedding h_0^v is a vector consisting of the initial state values of all unparameterized fluents.
2. Let $\mathsf{f} \in \mathsf{S} \cup \mathsf{N}$ and $\mathsf{V'_{ground}}$ be the set of object relation vertices associated with $\mathsf{ground}(\mathsf{f}, \mathsf{O})$. Observe that f is not an action fluent. Let $v = \langle \mathsf{o}_1, \ldots, \mathsf{o}_n \rangle \in \mathsf{V'_{ground}}$ and $\mathsf{F'}$ be the set of fluents for which $\langle \mathsf{o}_1, \ldots, \mathsf{o}_n \rangle$ is a grounded argument tuple. Then, we define $h_0^v = \langle \hat{s_I}(\mathsf{f}_1), \ldots, \hat{s_I}(\mathsf{f}_n) \rangle$, i.e. the initial state value associated with each fluent $\mathsf{f}_i \in \mathsf{F'}$. [2]
3. The embedding of all other object relation vertices is an empty vector of dimension 0. Note that these are the vertices for which the associated grounded argument tuple is only obtained from action fluents, and the vertices that consist of $(\mathsf{o}, \mathsf{master_i})$ for each $\mathsf{o} \in \mathsf{O}$.

Consider the instance subgraph of Figure 1. The initial embedding of the object relation vertex $\langle f_0, f_1 \rangle$ is a one-dimensional vector $\big[\hat{s_I}\big(\textit{ADJACENT-FLOORS}\langle f_0, f_1 \rangle\big)\big]$, and the initial embedding of the object relation vertex $\langle f_1 \rangle$ is a two-dimensional vector $\big[\hat{s_I}\big(\textit{BOTTOM-FLOOR}\langle f_1 \rangle\big), \hat{s_I}\big(\textit{TOP-FLOOR}\langle f_1 \rangle\big)\big]$. Observe that for any instance that has an object f_1, the dimension of the associated vertex embedding is 2, which is instance independent. What is instance dependent is the information whether f_1 is the top and/or bottom floor given by $\hat{s_I}$.

Updating object relation vertices

Let $\mathsf{G_d} = (\mathsf{V_T} \cup \mathsf{V_F}, \mathsf{E_d})$ be a domain graph. Node embeddings of the instance graph serve as input to the neural network modules associated with the domain graph. For each type relation vertex $v_\mathsf{F} \in \mathsf{V_F}$ and each step $k \in \{0, \ldots, K\}$ there is a distinct neural network module $\phi_k^{v_\mathsf{F}}$ associated with v_F and k. An object relation vertex v_{ground} associated with grounded argument tuple $\langle \mathsf{o}_1, \ldots, \mathsf{o}_n \rangle$ and $\mathsf{map}(v_{\mathsf{ground}}) = v_\mathsf{F}$ will use $\phi_k^{v_\mathsf{F}}$ to update its embedding at step k in the following way: $h_{k+1}^{v_{\mathsf{ground}}} = \phi_k^{v_\mathsf{F}}(x_k^{v_{\mathsf{ground}}})$, where $x_k^{v_{\mathsf{ground}}} = [h_k^{v_{\mathsf{ground}}}, h_k^{\mathsf{o}_1}, \ldots, h_k^{\mathsf{o}_n}]^T$, i.e. the embedding of v_{ground} at step k together with the embeddings of the associated object vertices o_i.

Note that the concatenation order of the object vertex embeddings (i.e. the information that the embedding of o_i is the $(i+1)$-th vector in $x_k^{v_{\mathsf{ground}}}$) is obtained from the edge label between v_{ground} and o_i. Additionally, the dimension of $h_{k+1}^{v_{\mathsf{ground}}}$ is determined by the output size of $\phi_k^{v_\mathsf{F}}$, and thus is a hyper-parameter of the graph neural network.

Consider again Figure 1 with object relation vertex $\langle f_0, f_1 \rangle$. Applying the update once gives us $h_1^{\langle f_0, f_1 \rangle} = \phi_0^{\langle floor, floor \rangle}\big(x_0^{\langle f_0, f_1 \rangle}\big)$, s.t. $x_0^{\langle f_0, f_1 \rangle} = [h_0^{\langle f_0, f_1 \rangle}, h_0^{f_0}, h_0^{f_1}]^T$.

Updating object vertices

By definition, object relation vertices that map to the same type relation vertex v_F have an equal number of adjacent vertices (the number of arguments associated with v_F), therefore a simple multi-layer perceptron with fixed input size is sufficient for those vertices to perform aggregation. This is however not guaranteed for object vertices that map to the same type vertex. We therefore have to aggregate a set of vectors with unknown cardinality. For this, we use attention layers (Velickovic et al. 2018).

In the following, we denote an edge between two vertices v and v' labeled with i as (v, i, v'). For all $(\mathsf{t}, i, v_\mathsf{F}) \in \mathsf{E_d}$ and $k \in \{0, \ldots, K\}$ there is a distinct neural network module $\phi_k^{(\mathsf{t}, i, v_\mathsf{F})}$. An object vertex o will use these modules to aggregate the embeddings of its neighbours and update its own embedding at step k by using the attention mechanism, which we describe in the following steps:

[2] The observant reader may notice that this requires a total order on the fluents in $\mathsf{F'}$. In practice, we consider the fluents in lexicographical order of their label.

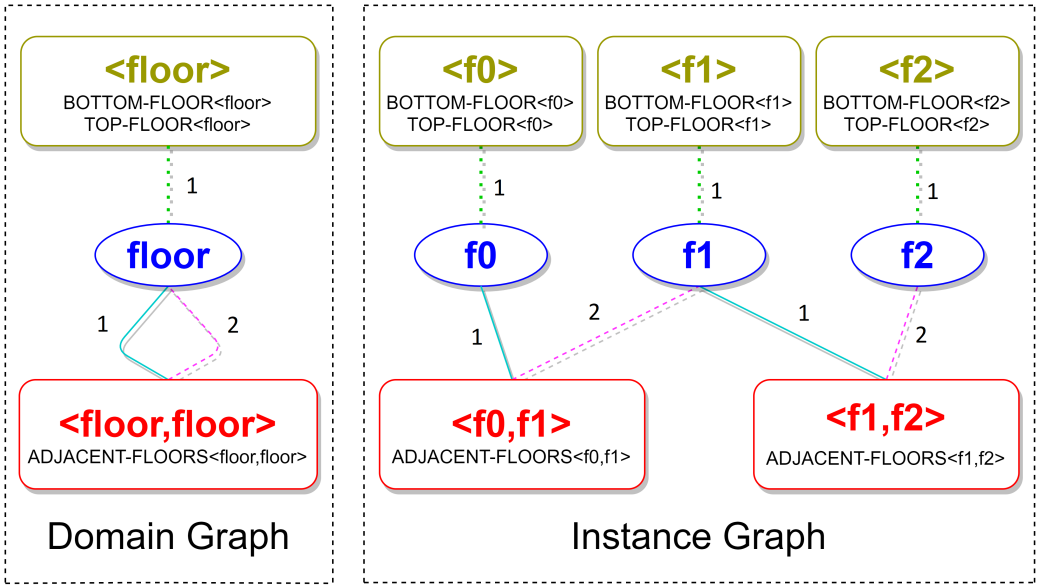

Figure 1: Subgraphs of the domain and instance graphs for the elevator domain. Rectangular vertices correspond to the type relation and object relation vertices, circular vertices correspond to type and object vertices.

1. Given object node o, the input for module $\phi_k^{(\mathsf{t},i,v_\mathsf{F})}$ is the embedding of object relation vertices $V' \subset \mathsf{V}_{\mathsf{ground}}$, such that $v' \in V'$ iff $(\mathsf{o}, i, v') \in \mathsf{E}_\mathsf{i}$, $\mathsf{map}(\mathsf{o}) = \mathsf{t}$ and $\mathsf{map}(v') = v_\mathsf{F}$.
2. Each module $\phi_k^{(\mathsf{t},i,v_\mathsf{F})}$ aggregates its input to output a vector $h_k^{\mathsf{o},(\mathsf{t},i,v_\mathsf{F})}$. Aggregation follows the attention layer mechanism of Velickovic et al. (2018).
3. The output vectors are concatenated, resulting in the single new object-embedding $h_{k+1}^{\mathsf{o}} = \|_{\mathsf{e}\in\mathsf{E}'} h_k^{\mathsf{o},\mathsf{e}}$, where $\mathsf{E}' = \{(\mathsf{t}, i, v_\mathsf{F}) \in \mathsf{E}_\mathsf{d} | \mathsf{type}(\mathsf{o}) = \mathsf{t}\}$.

Consider again Figure 1. We will show how the object vertex f_1 updates its embedding. Observe that f_1 has three connecting edges: $(f_1, 1, \langle f_1 \rangle)$ (green dotted edge), $(f_1, 2, \langle f_0, f_1 \rangle)$ (purple dashed edge) and $(f_1, 1, \langle f_1, f_2 \rangle)$ (cyan plain edge). Each edge corresponds to exactly one module: $\phi^{(floor,1,\langle floor \rangle)}$ will aggregate the set $\{h_1^{\langle f_1 \rangle}\}$ using an attention layer to obtain $h_1^{f_1,(floor,1,\langle floor \rangle)}$. Similarly, we can obtain $h_1^{f_1,(floor,2,\langle floor,floor \rangle)}$ by aggregating $\{h_1^{\langle f_0,f_1 \rangle}\}$ and $h_1^{f_1,(floor,1,\langle floor,floor \rangle)}$ by aggregating $\{h_1^{\langle f_1,f_2 \rangle}\}$. Finally, we concatenate the three vectors to obtain the new embedding $h_2^{f_1}$. Note that in this example all three attention layers are performed on singletons. In general this might not be the case.

Action Decoders

The aim of our network architecture is to output a policy, i.e. a distribution over the grounded action fluents. For this, we require a network module ϕ^{a} for each action fluent $\mathsf{a} \in \mathsf{A}$. Let $a = \mathsf{ground}(\mathsf{a}, \mathsf{O})$ be a grounded action fluent with argument tuple $\langle \mathsf{o}_1, \ldots, \mathsf{o}_n \rangle$ and let v_{ground} be the corresponding object relation vertex. The input to ϕ^{a} is a vector $[h_{K+1}^{v_{\mathsf{ground}}}, h_{K+1}^{\mathsf{o}_1}, \ldots, h_{K+1}^{\mathsf{o}_n}, h_{K+1}^{\mathsf{master}_\mathsf{i}}]^T$ and the output is $h_a \in \mathbb{R}$, which is an estimated value of action a. If a is unparameterized the input is simply $h_{K+1}^{\mathsf{master}_\mathsf{i}}$. To get the final probability distribution we pass the estimated values produced by the decoders through a softmax function, defined as $\mathsf{softmax}(z)_i = e^{z_i} / \sum_{j=1}^{C} e^{z_j}$ for $i \in \{1, \ldots, C\}$ and $z = (z_1, \ldots, z_c) \in \mathbb{R}^C$.

This concludes the description of our network architecture, resulting in the following workflow: given a graph neural network GNN associated with domain graph G_d and instance graph G_i, we compute the initial embedding of every vertex in the instance graph, perform K forward pass steps, update every embedding in each step, and run the corresponding action encoder for each action fluent, whose output will then become normalized action estimates resulting in a policy π. Algorithm 1 summarises this process.

Empirical Evaluation

We perform a preliminary study of the generalization power of our network architecture on four domains of the benchmark set of the probabilistic track of the international planning competition (IPC) 2014. Each problem is modeled in the relational dynamic influence language (RDDL) (Sanner 2010). A RDDL problem consists of a domain description which corresponds to the relational MDP $\mathcal{R}$, and an instance specification which yields the problem instance. We can thus construct the domain graph $\mathsf{G}_\mathsf{d} = (\mathsf{V}_\mathsf{T} \cup \mathsf{V}_\mathsf{F}, \mathsf{E}_\mathsf{d})$ from the domain description, and the instance graph $\mathsf{G}_\mathsf{i} = (\mathsf{V}_\mathsf{O} \cup \mathsf{V}_{\mathsf{ground}}, \mathsf{E}_\mathsf{i})$ from the instance specification.

Experiment setup

To evaluate the network's ability to generalise from small instances to large instances, the three smallest instances are

Algorithm 1: Forward pass of the GNN

for k *in* $[0, K]$ **do**
  for *each object relation vertex* $v_{\mathsf{ground}} \in \mathsf{V}_{\mathsf{ground}}$ **do**
    Update the embedding to get $h^{v_{\mathsf{ground}}}_{k+1}$;
  end
  for *each object vertex* $\mathsf{o} \in \mathsf{V}_{\mathsf{O}}$ **do**
    Update the embedding to get h^{o}_{k+1};
  end
end
for *each grounded action-fluent* $a \in \mathsf{ground}(\mathsf{a}, \mathsf{O})$ **do**
  Run the corresponding decoder ϕ^{a} to get action estimate h_a;
end
Normalize action estimates $\{h_a\}$ using $\mathsf{softmax}$ to get a policy π;
return *The estimated policy* π

used to train the network while the remaining instances are used to evaluate the network. To generate training data we use the $UCT^\star$ algorithm of the PROST planner (Keller and Eyerich 2012). For each of the three training instances we give the planner a time-limit of 30s per step and plan for 100 rounds. In many cases, this results in almost optimal policies for each of the three instances, which are used to train the network in a supervised learning way. More precisely, let $D_i = \bigcup_j \{s_j, \pi_j(s_j)\}$ be the training data for instance i, where each s_j is a state and $\pi_j(s_j)$ is the generated policy for that state. The overall training data for the network is then $D = \bigcup_i \{D_i\}$ containing results of the three instances. Then we apply the backpropagation algorithm to compute the gradients for each instance and those gradients are then summed up such that we can perform stochastic gradient descent to update the network. We use cross-entropy loss function and the learning rate is 0.01. We train each network for approximately 4 hours.

We implemented the network architecture in C++, using PyTorch and the PROST planner as the underlying planning framework. Training and evaluation are conducted on a PC with a i7-10870H CPU and 16GB RAM. We set the number of forward steps to $K = 5$ and each network module is implemented as a three-layer perceptron with ReLU nonlinearity.

Evaluation

For each problem instance we use the trained network to perform a forward-pass over the instance graph over K steps and apply the resulting policy π in every round. We compare against PROST using $UCT^\star$ with a time limit of 1 second per step, which was the competition setting of the IPC 2014. We emphasize that this is a preliminary evaluation, and the nature of *domain-independent* online planners is different to our pre-trained relational policies. Nevertheless, the PROST planner is the winner of the IPC 2014, thus we can get some insight on the performance of our network when evaluated on larger instances. It is noteworthy that the construction and evaluation of our policy π is several orders of magnitude faster than the total time that we allow for PROST.

Table 1 shows the average reward based on 100 rounds and the 95% confidence intervals of rewards for each domain. For Tamarisk the average reward of the network policy is generally higher, however, when we consider the range of the confidence intervals both algorithms perform equally strong. For Wildfire and Sysadmin the network significantly outperforms the PROST planner in almost every instance. However, this picture is reversed for the Elevator domain. While the performance on the trained instances is almost equal, PROST significantly outperforms the network on the remaining instances. Elevator is a domain where PROST computes close to optimal policies for every instance, which is clearly not the case for the network.

It is worth to mention that the PROST planner is highly efficient on the Elevator domain, which is solved optimally on many of the instances. This is different to Sysadmin, Tamarisk and Wildfire, which are more complex domains where it is harder to compute an optimal policy in the given time limit. Unfortunately, the average reward of an optimal policy for these domains is unknown, which makes it hard to precisely estimate the performance of our relational policies. Nevertheless, the generalisation aspect from small problems to larger problems is clearly noticable, as the performance of harder problems does not significantly deteriorate.

Conclusion

In this paper, we presented a novel network architecture based on the domain graph of a given problem class. Unlike the networks of Garg, Bajpai, and Mausam (2019) we can deal with fluents of arbitrary size. The comparison against the PROST planner shows that our networks can generalize well across instances of different size.

However, this study serves only as a first step towards an understanding of the generalization power of our network. In particular, it is an open question how our architecture compares to other related work, such as the RDDL specific SymNet architecture of Garg, Bajpai, and Mausam (2020), or the Action Schema networks of Toyer et al. (2018). One particularly interesting aspect that deserves a thorough inspection is that unlike other works our network architecture is in principle independent of modeling language in which the lifted MDP is described, i.e. we should be able to support PPDDL problems as well without having to adapt the network structure.

Acknowledgements

We thank the anonymous reviewer for their detailed and insightful comments.

References

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; Gulcehre, C.; Song, F.; Ballard, A.; Gilmer, J.; Dahl, G.; Vaswani, A.; Allen, K.; Nash, C.; Langston, V.; Dyer, C.; Heess, N.; Wierstra, D.; Kohli, P.; Botvinick, M.; Vinyals, O.; Li, Y.; and Pascanu, R.

	Tamarisk domain		Wildfire domain	
Instance	**THTS**	**Network**	**THTS**	**Network**
1(trained on)	-151±19.7	**-146±15.4**	-647.1± 288	**-548.2±300**
2(trained on)	-542±27.7	**-530±24.2**	-11201.3±499	**-10048.1±408**
3(trained on)	**-206±27.3**	-222±23.8	**-1694.3±592**	-1890.2±550
4	-826±28.4	**-822±28.6**	-20126.8±1180	**-14616.3±816**
5	-723.9±41.4	**-655±38**	-2905.0±686	**-1515.6±423**
6	-1071±26.2	**-1045±33.2**	-25866.2±864	**-8862.6±1030**
7	-891±43.2	**-823±41.7**	-8816.2±748	**-6845.2±646**
8	-1285±23.5	**-1251±28.2**	-15811.8±1400	**-11124±910**
9	-902±53.2	**-860±59.9**	-14457.6±629	**-8721.6±906**
10	-1346±36.3	**-1290±39.3**	-21766.5±1110	**-11331.1±619**
	Elevator domain		**Sysadmin domain**	
Instance	**THTS**	**Network**	**THTS**	**Network**
1(trained on)	**-42.5±2.5**	-45.8±2.6	340.1±3.7	339.5±4.8
2(trained on)	-23.8±2.2	-23.6±2.6	**315.3±7.2**	303.5±9.9
3(trained on)	**-61.6±1.9**	-62.6±1.9	**550.2±14.1**	541.4±13.7
4	**-54.2±4.2**	-96.7±5.1	**495.4±16.3**	459.8±14.3
5	**-64.9±4.6**	-104.2±4.9	581.0±17	**587.9±15.2**
6	**-83.3±3.8**	-120.0±3.8	529.8±15.4	**553.6±15.8**
7	**-79.4±5.4**	-133.2±6.3	611.0±14.7	**683.7±16.1**
8	**-88.2±5.2**	-151.3±5.8	505.3±16.5	**532.1±13.6**
9	**-107.5±5.4**	-160.8±5.5	739.9±17.1	**825.5±14.3**
10	**-66.5±5.9**	-117.0±7.7	553.8±14.4	**606.5±15.6**

Table 1: Average reward over 100 rounds and 95% confidence intervals for four domains of the IPC 2014.

2018. Relational inductive biases, deep learning, and graph networks. arXiv:1806.01261 [cs.LG].

Bellman, R. E. 1957. *Dynamic Programming*. Princeton University Press.

Bonet, B.; and Geffner, H. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In Giunchiglia, E.; Muscettola, N.; and Nau, D., eds., *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003)*, 12–21. AAAI Press.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions Computational Intelligence and AI in Games* 4(1): 1–43.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 631–636. AAAI Press.

Garg, S.; Bajpai, A.; and Mausam. 2020. Symbolic Network: Generalized Neural Policies for Relational MDPs. In *International Conference on Machine Learning*, 3397–3407.

Hansen, E. A.; and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1–2): 35–62.

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 119–127. AAAI Press.

Keller, T.; and Helmert, M. 2013. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 135–143. AAAI Press.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer-Verlag.

Mausam; and Kolobov, A. 2012. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2): 35–55.

Puterman, M. L. 1994. *Markov Decision Processes: Dis-*

crete Stochastic Dynamic Programming. John Wiley & Sons, Inc.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 574–584. AAAI Press.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815v1 [cs.AI].

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6294–6301. AAAI Press.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018*. OpenReview.net. URL https://openreview.net/forum?id=rJXMpikCZ.

Younes, H. L. S.; and Littman, M. L. 2004. PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. Technical Report CMU-CS-04-167, Carnegie Mellon University, School of Computer Science.