Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)

Specifying Goals to Deep Neural Networks with Answer Set Programming

Forest Agostinelli[1,2], **Rojina Panta**[1,2], **Vedant Khandelwal**[1,2]
[1] AI Institute, University of South Carolina, USA
[2]Department of Computer Science and Engineering, University of South Carolina, USA
foresta@cse.sc.edu, rpanta@email.sc.edu, vedant@mailbox.sc.edu

Abstract

Recently, methods such as DeepCubeA have used deep reinforcement learning to learn domain-specific heuristic functions in a largely domain-independent fashion. However, such methods either assume a predetermined goal or assume that goals will be given as fully-specified states. Therefore, specifying a set of goal states to these learned heuristic functions is often impractical. To address this issue, we introduce a method of training a heuristic function that estimates the distance between a given state and a set of goal states represented as a set of ground atoms in first-order logic. Furthermore, to allow for more expressive goal specification, we introduce techniques for specifying goals as answer set programs and using answer set solvers to discover sets of ground atoms that meet the specified goals. In our experiments with the Rubik's cube, sliding tile puzzles, and Sokoban, we show that we can specify and reach different goals without any need to re-train the heuristic function. Our code is publicly available at https://github.com/forestagostinelli/SpecGoal.

Introduction

Deep reinforcement learning algorithms (Sutton and Barto 2018), such as DeepCubeA (McAleer et al. 2019; Agostinelli et al. 2019) and Retro* (Chen et al. 2020), have successfully trained deep neural networks (DNNs) (Schmidhuber 2015) to be informative heuristic functions. Combined with heuristic search methods such as A* search (Hart, Nilsson, and Raphael 1968), Q* search (Agostinelli et al. 2021), or Monte Carlo Tree Search (Kocsis and Szepesvári 2006), these learned heuristic functions can solve puzzles, perform retrosynthesis, as well as compile quantum algorithms (Zhang et al. 2020). However, these DNNs do not generalize across goals where, in this context, a goal is a set of states in the state space that are considered goal states. Instead, these DNNs are either trained for a predetermined goal or use methods such as hindsight experience replay (Andrychowicz et al. 2017) to generalize across pairs of start and goal states. As a result, specifying a goal to a DNN requires either training a DNN for that specific goal or obtaining the heuristic values for some representative set of goal states and taking the minimum heuristic value. This computationally burdensome process significantly reduces the practicality of DNNs for solving planning problems without predetermined goals. Furthermore, if one can only describe properties that a goal state should or should not have, but does not know what states meet this criteria, obtaining a representative set of goal states is not possible.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

To train DNNs to estimate the distance between a state and a set of goal states, we introduce DeepCubeA$_g$, a deep reinforcement learning and search method that builds on DeepCubeA (McAleer et al. 2019; Agostinelli et al. 2019) and hindsight experience replay (Andrychowicz et al. 2017) to learn heuristic functions that generalize across states and goals. Training data in the form of pairs of states and goals is obtained by starting from a given start state and taking a random walk to obtain a goal state. Given a process to convert a state to a set of ground atoms that represents what holds true in that state, we convert the obtained goal state to a set of ground atoms and then obtain a set of goal states by taking a subset of this set of ground atoms. We then train a heuristic function with deep approximate value iteration (DAVI) (Bertsekas and Tsitsiklis 1996; Agostinelli et al. 2019) to map states and goals to an estimated cost-to-go. When solving problem instances, we use the trained heuristic function with a batched version of A* search. We evaluate this approach on the Rubik's cube, 15-puzzle, 24-puzzle, and Sokoban (Dor and Zwick 1999) and results show that DeepCubeA$_g$ is able to find solutions for the vast majority of test instances and does so better than the domain-independent fast downward planner (Helmert 2006).

To allow for expressive goal specification, we build on the fact that goals are represented as sets of ground atoms. Therefore, to specify a goal, any specification language that can be translated to a set of ground atoms can be used. We choose answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011), a form of first-order logic programming, as the specification language because one can obtain stable models (Gelfond and Lifschitz 1988), also known as answer sets, for a given specification, where specifications are answer set programs and each stable model is a set of ground atoms. Results show that diverse goals can be specified with simple answer set programs and reached using the learned heuristic function and search. An overview of our approach is described in Figure 1.

Preliminaries

Our method builds on the DeepCubeA algorithm (Agostinelli et al. 2019) that was used to train a DNN as a heuristic function using deep approximate value iteration (Puterman and Shin 1978; Bertsekas and Tsitsiklis 1996). This heuristic function was then used in a batched version of weighted A* search (Pohl 1970) to solve puzzles such as the Rubik's cube and Sokoban. For specifying goals, we use ASP. In this section, we will describe the background of deep approximate value iteration as well as the background of ASP. We also describe the basics of the Rubik's cube.

Deep Approximate Value Iteration

In the context of deterministic, finite-horizon, shortest path problems, approximate value iteration is a reinforcement learning (Sutton and Barto 2018) algorithm to learn a function, h, that maps a state s to its estimated cost-to-go. The optimal heuristic function, h^*, returns the cost of a shortest path. The value iteration algorithm (Puterman and Shin 1978) takes a given h and updates it to h' according to Equation 1

$$h'(s) = \min_a (g^a(s, s') + h(s')) \tag{1}$$

where $g^a(s, s')$ is the cost to transition from s to state s' using action a and s' is the state resulting from taking action a in state s.

In the tabular setting, value iteration has been shown to converge to h^*. However, for domains with large state spaces, such as the Rubik's cube, we do not have enough memory, or time, to do tabular value iteration. Therefore, we represent h with a parameterized function, h_ϕ, with parameters, ϕ. The parameters of the function are trained to minimize the loss function in Equation 2

$$L(\phi) = (\min_a g^a(s, s') + h_{\phi^-}(s') - h_\phi(s))^2 \tag{2}$$

where ϕ^- are parameters of a target function that remains fixed for a certain number of training iterations and is periodically updated to ϕ. This has been shown to make the training process more stable because the target remains stationary for extended periods of time (Mnih et al. 2015). When h_ϕ is a deep neural network, this approach is referred to as deep approximate value iteration (DAVI).

Answer Set Programming

Answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011) is a form of logic programming that is built on the stable model semantics (Gelfond and Lifschitz 1988) which describes when a set of ground atoms, M, is a stable model, also known as an answer set, of a program, Π. Program Π is restricted to be a set of rules in first-order logic of the form:

$$A \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{3}$$

where A, B_i, and C_i are atoms in first-order logic. A is in the "head", or the consequent, and B_i and C_i are in the "body", or the antecedent. In this notation, $\neg$ represents negation, a comma represents conjunction, and $\leftarrow$ represents implication. Since all literals in the body are connected with conjunction, the body is true if and only if all literals in the body are true. Since the head has just one atom, the head is true if and only if A is true. Since the head and the body are connected by implication, the entire logical sentence is true if and only if one of the two following conditions are met: 1) the body is false; 2) the body is true and the head is true. If there are no literals in the body, then semantics dictate that the body is always true; therefore, the head must also always be true. If there are no atoms in the head (also known as "headless" rules), then semantics dictate that the head is always false; therefore, the body must also always be false. In practice, headless rules are used as constraints and are implicitly represented with a literal, A, in the head and a literal, $\neg A$, in the body that is in conjunction with the rest of the body literals. Therefore, headless rules are rules with negation in the body.

To determine if M is a stable model of Π, we first must consider the grounded program of Π, which we will denote Π_g. To obtain Π_g, for all rules, R, in Π, every possible grounded version of R, R_g, is obtained and added to Π_g. A ground rule, R_g, is obtained from a rule, R, by substituting all variables in R for a ground term appearing in Π. If there are no rules in Π_g with negation, then there is one unique minimal stable model of Π_g (Van Emden and Kowalski 1976; Gelfond and Lifschitz 1988) which corresponds to all atoms that are derivable from Π_g. An atom is derivable if it is in the head of a rule with a body that is true. If there are rules with negation in Π_g, then we can check if a given set of ground atoms, M, is a stable model of Π_g by first computing the reduct (Marek and Truszczyński 1999) of Π_g with respect to M, which we will denote Π_g^M. Π_g^M is obtained by starting with Π_g and deleting all rules that have a negative literal, $\neg C_i$, in the body if C_i is in M and then deleting all negative literals in the body of the remaining rules. Π_g^M is now a negation free program, which means that it has one unique minimal stable model. If this stable model of Π_g^M is equivalent to M, then M is a stable model of Π. Π can have multiple stable models if it contains negation.

Some ASP solvers, such as clingo (Gebser et al. 2014, 2022), allow for choice rules. Choice rules have a conjunction of literals in the body and a set of ground atoms in the head. If the body is true, then zero or more ground atoms in the head may be added to the stable model. Furthermore, clingo also allows for the use of disjunction, which can result in more than one stable model, even if negation is not present.

The Rubik's Cube

The Rubik's cube is a three dimensional cube where each face of the cube consists of a 3 x 3 grid of stickers, with 54 stickers in total. Each sticker can be one of six colors: white, yellow, orange, red, blue, or green. These stickers combine where the faces intersect to form cubelets, where center cubelets, edge cubelets, and corner cubelets have 1, 2, and 3 stickers, respectively. While the canonical goal state for the Rubik's cube is one where all stickers on each face

have the same color, there are many other patterns that interest the Rubik's cube community (Ferenc 2013).

Methods

Learning Heuristic Functions for Goals

To learn a function that estimates the distance between a state, s, and a goal, $\mathcal{G}$, we must explicitly add the specified goal as an input to the heuristic function. Therefore, the heuristic function now becomes $h(s, \mathcal{G})$, that represents the cost to go from s to a closest state in $\mathcal{G}$. We assume a function, $G(s)$, that maps states to a set of ground atoms that represents what holds true in a given state and a process to convert $\mathcal{G}$ to a representation suitable for the DNN. To train the DNN, we must first have the ability to sample state and goal pairs. From these pairs, we can then train the DNN using DAVI.

To sample state and goal pairs, the agent starts at a randomly generated state, s_0. The agent then takes t actions, where t is drawn from a random uniform distribution between 0 and a given number T. Each action is sampled according to a random uniform distribution[1]. The last observed state, s_t, is then selected to create a goal, $\mathcal{G}$, by first obtaining $G(s_t)$. Since any $G(s_t)$ that is a superset of a goal, $\mathcal{G}$, also represents a goal, we can randomly remove atoms from $G(s_t)$ to create $\mathcal{G}$ such that $\mathcal{G} \subseteq G(s_t)$ and; therefore, s_t is a member of the set of goal states represented by $\mathcal{G}$. The loss for the DNN is computed according to Equation 4. The parameters of the target network, ϕ^-, are periodically updated to ϕ. This training procedure is outline in Figure 1.

$$L(\phi) = (\min_a g^a(s, s') + h_{\phi^-}(s', \mathcal{G}) - h_\phi(s, \mathcal{G}))^2 \quad (4)$$

Specifying Goals with Answer Set Programming

A logic program, Π, used to specify a goal contains background knowledge, B, which is a set of rules that describes relevant domain knowledge, a goal specification, H, which is a set of rules with the atom `goal` in the head, a headless rule, `:- not goal`, that ensures `goal` must be true in all stable models, and a choice rule with an empty body that contains the set of all possible ground atoms, K, that can be used to represent a set of states. Given a stable model, M, of Π, the subset of M in K, M_K, represents a set of states. When obtaining a stable model, we would like to find a minimal M_K to ensure the stable model is as general as possible (De Raedt 2008). M_K is minimal if and only if removing ground atoms from M_K will result in `goal` no longer being true. To accomplish this, for an M_K obtained from an answer set solver, we pick a ground atom, a, in M_K and remove it. We check if `goal` can still be true for $M_K \setminus \{a\}$. If so, we set M_K to $M_K \setminus \{a\}$ and repeat this process. If not, we choose another atom to remove. If we cannot remove any atoms and ensure `goal` is also true, then we terminate.

We will now formally define what a goal state and goal model is and how this relates to negation as failure.

[1]Future work could use intrinsic motivation (Barto et al. 2004) to encourage the exploration of diverse states.

Definition 1 (Goal state)**.** Given a program, Π, a state, s, is a goal state if and only if $G(s)$ is a subset of some stable model of Π.

Definition 2 (Goal model)**.** Given a program, Π, a set of ground atoms, M, is a goal model if and only if M is a stable model of Π and for every state, s, such that $G(s)$ is a superset of M_K, s is a goal state.

If M is indeed a goal model, then M_K represents a set of goal states. However, it is not the case that all stable models of Π are goal models since, in general, logic programs in ASP can exhibit non-monotonic behavior due to the closed world assumption. A logic program is non-monotonic if some atoms that were previously derived can be retracted by adding new knowledge. To handle this, we will iteratively look for larger models in an attempt to reduce the number of stable models that are not goal models. We will use the clingo (Gebser et al. 2014, 2022) ASP software package to specify goals.

Reaching Goals

Given a DNN trained to estimate the distance between a state and a goal, where a goal is represented as a set of ground atoms, as well as a specification in the form of a logic program, Π, we can now describe how goals are reached. We first start by finding a stable model, M, of Π. Since M_K is not guaranteed to be a goal model, it is possible that the terminal state along some path to M_K is not a goal state. Therefore, we will use the DNN with A* search to find a path to M_K. If we find a terminal state that is a goal state, then we can return the path to that state. If we do not find a terminal state, then M_K may represent a set of unreachable states (see the Future Work Section) and we ban M_K, where banning prevents the ASP solver from returning stable models that contain M_K, and sample a new stable model. Otherwise, if the terminal state is not a goal state, we refine M by searching for a stable model that contains a strict superset of M_K. This corresponds to finding a new stable model, M', where M'_K represents a subset of the states represented by M_K. To accomplish this, a new stable model, M', is found with the constraint M'_K must contain all atoms in M_K and that the size of M'_K must be bigger than M_K. This process is outlined in Algorithm 1. Note, in this algorithm, a stable model is deemed to be "None" if the answer set solver does not find any stable model.

Similar to previous work (Agostinelli et al. 2019, 2021), to take advantage of the parallelism of graphics processing units (GPUs), we do a batched version of A* search that removes multiple nodes from the priority queue at each iteration.

Experiments

Representation and Training

To specify a set of states for the Rubik's cube, we use a predicate, `at_idx`, of arity 2, that holds when a given color is at a given index. For example, `at_idx(red,12)` holds if a red sticker is at index 12 on the Rubik's cube. To represent a set of these ground atoms to the DNN, we first use a vector of

Algorithm 1: Reaching a Specified Goal

Input: Program Π, DNN h_ϕ, start state s_0, max itrs I
Sample stable model M of Π
$i = 0$
while (M is not None) and ($i < I$) **do**
  s_g = A*(s_0, M_K, h_ϕ)
  if s_g is None **then**
    Ban M
  else if $G(s_g) \subseteq$ some stable model of Π **then**
    return path to s_g
  else
    Sample stable model M' of Π s.t. $M_K \subset M'_K$
    $M = M'$
  end if
  if M is None or s_g is None **then**
    Sample stable model M of Π
  end if
  $i++$
end while
return failure

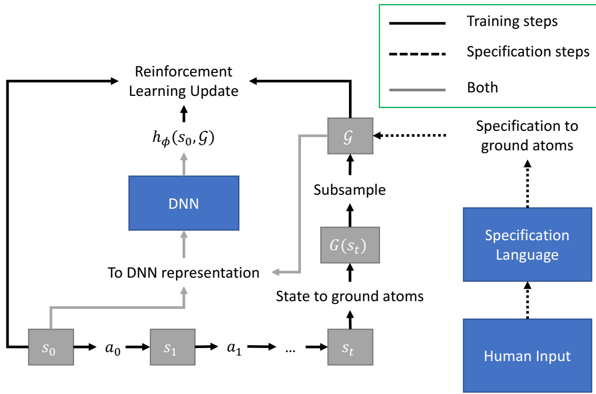


Figure 1: Outline of training and goal specification.

length 54 to represent each sticker. We then set colors values of 0 through 5 based on the at_idx predicate. Unspecified indices in the vector are set to 6. We then use a one-hot representation of this vector as the input to the DNN. To specify a set of states for the sliding tile puzzles, we use a predicate at_idx, of arity 3, that holds when a given tile is at a given x and y coordinate. The representation given to the DNN is a one-hot vector of tiles where there is a special tile for those whose position is unspecified. To specify a set of states for Sokoban, we use the predicates agent, box, and wall of arity 2, which holds true if a given agent, box, or wall is at a given x and y coordinate. The representation given to the DNN is three binary matrices that represent the locations of the specified agent, boxes, and walls.

The architecture of the DNN is similar to the one described in Agostinelli et al. (2019), with the only difference being the addition of a goal input. Furthermore, when optimizing the DNN, the target network is updated based on a test set instead of a training loss, as was done in Agostinelli et al. (2019). Specifically, we generate a test set on which we periodically test the greedy policy and an update is done when the number of states solved by the greedy policy increases. To randomly generate start states for the Rubik's cube, for each state, we start from the canonical goal state and randomly take between 100 and 200 actions. To generate start states for the sliding tile puzzle, we create random permutations and check for validity with parity. To generate start states for Sokoban, we start from a provided 900,000 training states (DeepMind 2018) and take a random walk with a length between 0 and 30. We set the maximum number of actions to take from the start state to generate goal states, T, to 30, for the Rubik's cube and 1,000 for the 15-puzzle, 24-puzzle, and Sokoban. We train and test using two NVIDIA Tesla V100 GPUs and 48 2.4 GHz Intel Xeon Platinum CPUs. Training is done with a batch size of 10,000 for 2 million iterations for the Rubik's cube and 15-puzzle, four million iterations for the 24-puzzle, and one million iterations for Sokoban.

Specifying Goals with Sets of Ground Atoms

Since we will use ASP to find stable models to specify to the trained heuristic function, where stable models are sets of ground atoms, we first test the ability of DeepCubeA$_g$ to reach goals specified as sets of ground atoms. Test states are obtained from Agostinelli et al. (2019), which contains 1,000 randomly generated states for the Rubik's cube, 500 randomly generated states for the 15-puzzle and 24-puzzle, and 1,000 randomly generated states for Sokoban. We use pattern databases (Culberson and Schaeffer 1998) to validate the cost of a shortest path. For the Rubik's cube, we use a pattern database that takes advantage of domain-specific mathematical group properties of the Rubik's cube (Rokicki 2010, 2016). We use the 12 atomic actions for the Rubik's cube, so the maximum cost-to-go is 26. For the sliding tile puzzles, we use additive pattern database heuristics described in Felner, Korf, and Hanan (2004).

The goal for the aforementioned test states for the Rubik's cube and sliding tile puzzles is only the single canonical goal state. To test the ability of DeepCubeA$_g$ to generalize across any given goal, we randomly generate 500 pairs of start and goal states. We do this by starting from a randomly generated start state, s_0, taking a random walk with k steps where k is uniformly distributed between 1,000 to 10,000 steps, and sampling a random subset of $G(s_k)$ to represent the goal, where s_k is the final state in the random walk. This random subset of $G(s_k)$ can be as large as $G(s_k)$, itself, or as small as the empty set. Therefore, the size of the set of goal states that is represented by these sampled goals could be between one and every state in the state space. We compare DeepCubeA$_g$ to DeepCubeA, and the fast downward planning system (Helmert 2006). For all test examples, we give each solver 200 seconds to solve them. DeepCubeA$_g$ is implemented in Python and uses two NVIDIA Tesla V100 GPUs for computing the heuristic function and a single 2.4 GHz Intel Xeon Platinum CPU, otherwise. Batch A* search is performed with a batch size of 10,000 for the Rubik's cube, 1,000 for the sliding tile puzzles, and 100 for Sokoban. For the fast downward planner, we perform A* search with the goal count heuristic, fast forward heuristic,

and the causal graph heuristic. We note that we could not run DeepCubeA on the test set with randomly generated goals since, for DeepCubeA, goals must be predetermined before training and training would take over a day for each of the 500 test examples.

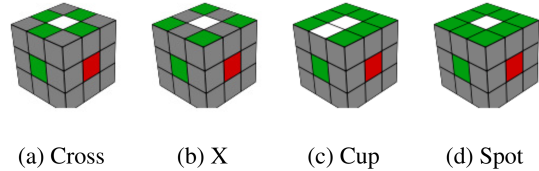
Results Results are shown in Table 1. The results show that DeepCubeA$_g$ consistently outperforms the fast downward planning system in terms of the percentage of states that are solved. DeepCubeA$_g$ solves either 100% of states or close to 100% of states. In the single domain where the fast downward planner solved 100% of test cases, Sokoban, DeepCubeA$_g$ also solved 100% of test cases while also finding shorter paths. In cases such as the Rubik's cube and 24-puzzle, for the canonical goal states, DeepCubeA$_g$ solves 100% of test states, while the fast downward planner solves between 0% and 1.1%.

Specifying Goals with Answer Set Programming

Rubik's Cube The background knowledge for the Rubik's cube defines colors, cubelets, and what color stickers the cubelets have. We also define directions (clockwise, counterclockwise, and opposite), faces, face colors (the same as the center cubelet), and their relation to one another (for example, the blue face is a clockwise turn away from the orange face with respect to the white face). We also describe what it means for a cubelet to have a sticker on a face as well as for a cubelet to be "in place" (all colors matching the center cubelet). We add constraints to the program to prune stable models that represent impossible states. These constraints include saying that different stickers from the same cubelet cannot be on the same face or opposite faces as well as saying that a cubelet cannot have a sticker on more than one face.

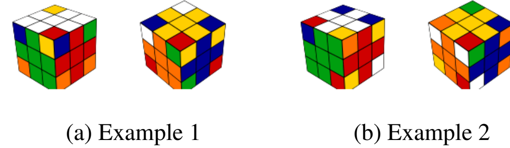
To specify goals, we draw from Ferenc (2013) to come up with goals that combine different Rubik's cube patterns shown in Figure 2. We also test our method with the canonical solved state for the Rubik's cube where all faces have a uniform color. Note that the training procedure is not told of these patterns and is not aware that these patterns will be used for testing. Given the background knowledge, many patterns only require a few lines of code. A few are shown here:

```
cross(F, CrossCol) :- face(F),
color(CrossCol), #count{Cbl:
edge_cbl(Cbl), onface(Cbl, CrossCol,
F)} = 4.

spot(F, BCol) :- color(BCol), face(F),
face_col(F, FCol), dif_col(FCol, BCol),
#count{Cbl: onface(Cbl, BCol, F),
edge_or_corner(Cbl)} = 8.

face_same(F) :- face_col(F, FCol),
#count{Cbl : onface(Cbl, FCol, F)}=9.
canon :- #count{F : face_same(F)}=6.
```



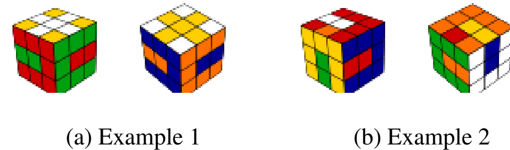
(a) Cross (b) X (c) Cup (d) Spot

Figure 2: Examples of patterns used to create goals.



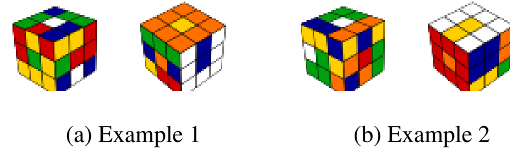
(a) Example 1 (b) Example 2

Figure 3: Reached goal of having a cross on all 6 faces where the center cubelet and cross are the same color.



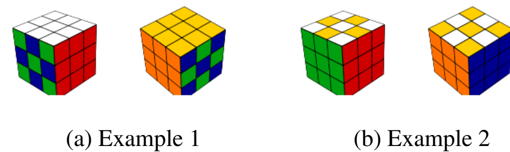
(a) Example 1 (b) Example 2

Figure 4: Reached goal of having cups on red, green, blue, and orange faces.



(a) Example 1 (b) Example 2

Figure 5: Reached goal of having a cup adjacent to a spot.



(a) Example 1 (b) Example 2

Figure 6: Reached goal of having two checkerboards on opposite faces.

In addition to the canonical goal, we specify four other goals: (1) all faces have a cross where the cross is the same color as the center piece; (2) the red, green, blue, and orange faces have a cup on them (3) there is a spot adjacent to a cup with the opening of the cup facing the spot; (4) there are two checkerboard patterns (a cross combined with an X) on opposite faces.

Sokoban The background knowledge for Sokoban defines the dimensions of the grid, the relations of coordinates in terms of up, down, left, and right, what it means for a box

Puzzle	Solver	Path Cost	% Solved	% Opt	Nodes	Secs	Nodes/Sec
RC (Canon)	PDBs$^+$	**20.67**	**100.00%**	**100.0%**	**2.05E+06**	**2.20**	**1.79E+06**
	DeepCubeA	21.50	**100.00%**	60.3%	6.62E+06	24.22	2.90E+05
	DeepCubeA$_g$	22.03	**100.00%**	35.00%	2.44E+06	41.99	5.67E+04
	FastDown (GC)	-	0.00%	0.0%	-	-	-
	FastDown (FF)	-	0.00%	0.0%	-	-	-
	FastDown (CG)	-	0.00%	0.0%	-	-	-
RC (Rand)	DeepCubeA$_g$	15.22	**99.40%**	-	1.91E+06	32.24	5.19E+04
	FastDown (GC)	7.18	32.80%	-	2.67E+06	13.79	1.41E+05
	FastDown (FF)	6.49	31.20%	-	4.87E+05	13.83	2.93E+04
	FastDown (CG)	7.85	33.80%	-	1.12E+06	11.62	5.81E+04
15-P (Canon)	PDBs	**52.02**	**100.00%**	**100.0%**	**3.22E+04**	**0.002**	**1.45E+07**
	DeepCubeA	52.03	**100.00%**	99.4%	3.85E+06	10.28	3.93E+05
	DeepCubeA$_g$	**52.02**	**100.00%**	**100.0%**	1.81E+05	2.61	6.94E+04
	FastDown (GC)	36.75	0.80%	0.80%	9.05E+07	102.11	8.66E+05
	FastDown (FF)	52.75	80.80%	24.80%	2.92E+06	42.11	6.93E+04
	FastDown (CG)	41.95	4.40%	1.20%	2.00E+07	80.58	2.47E+05
15-P (Rand)	DeepCubeA$_g$	**33.98**	**100.00%**	-	**1.11E+05**	**1.60**	**6.16E+04**
	FastDown (GC)	14.92	38.00%	-	1.61E+07	18.77	5.46E+05
	FastDown (FF)	32.66	89.20%	-	1.24E+06	17.39	5.65E+04
	FastDown (CG)	20.45	51.20%	-	3.90E+06	21.41	1.20E+05
24-P (Canon)	PDBs	**89.41**	**100.00%**	**100.00%**	8.19E+10	4239.54	**1.91E+07**
	DeepCubeA	89.49	**100.00%**	96.98%	6.44E+06	19.33	3.34E+05
	DeepCubeA$_g$	90.47	**100.00%**	55.24%	**3.38E+05**	**5.22**	6.48E+04
	FastDown (GC)	-	0.00%	0.00%	-	-	-
	FastDown (FF)	81.00	1.01%	0.40%	2.68E+06	89.84	2.91E+04
	FastDown (CG)	-	0.00%	0.00%	-	-	-
24-P (Rand)	DeepCubeA$_g$	66.28	**99.60%**	-	3.10E+05	4.91	6.16E+04
	FastDown (GC)	9.86	10.00%	-	9.54E+06	11.88	4.27E+05
	FastDown (FF)	26.35	26.00%	-	5.99E+05	19.57	2.41E+04
	FastDown (CG)	13.75	12.60%	-	1.42E+06	14.42	6.85E+04
Sokoban	DeepCubeA	32.88	**100.00%**	-	**5.01E+03**	2.71	1.84E+03
	DeepCubeA$_g$	**32.02**	**100.00%**	-	1.80E+04	0.95	1.79E+04
	FastDown (GC)	31.94	99.80%	-	3.17E+06	5.93	5.85E+05
	FastDown (FF)	33.15	**100.00%**	-	2.92E+04	**0.32**	**7.49E+04**
	FastDown (CG)	33.12	**100.00%**	-	4.43E+04	0.51	7.25E+04

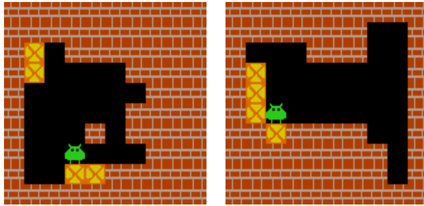
Table 1: Comparison of DeepCubeA$_g$ with optimal solvers based on pattern databases (PDBs) that exploit domain-specific information and the domain-independent fast downward planning system with the goal count (GC), fast forward (FF), and causal graph (CG) heuristics. Comparisons are along the dimensions of solution length, percentage of instances solved, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), and number of nodes generated per second. For the Rubik's cube and sliding tile puzzles, experiments are done on canonical goal states (Canon) and randomly generated goals (Rand). For testing DeepCubeA on Sokoban, we report numbers obtained from the DeepCubeA GitHub repository[2].

to be at the edge of the grid, what it means for a box to be immovable, as well as basic constraints that state that two objects cannot share the same location. In this domain, the start state determines the ground atoms that will be present in a goal state. In particular, the walls cannot be modified; therefore, the specification of a goal must also take this into account. To address this, we add the location of the walls to the specification. We investigate the following goals: (1) all boxes are immovable; (2) all boxes form a larger box; (3) the four boxes occupy the four corners next to the agent.

Results Our experiments use 100 start states from the test states used in Table 1 and follow Algorithm 1 (without setting a maximum iteration) to find a path from these start states to the goal. Given a specified goal, which is an answer set program, we use clingo to find stable models and use batch weighted A* search with a batch size of 1,000, a weight of 0.6 on the path cost, and a search budget of 50 iterations to find a path to a sampled stable model. Visualizations of reached goals for the four non-canonical Rubik's cube goals are shown in Figures 3, 4, 5, and 6, and for Sokoban goals are shown in Figures 7, 8, and 9. A table summarizing the path cost of solutions, number of models sampled, time it takes to find stable models, and time it takes to do search is shown in Table 2.

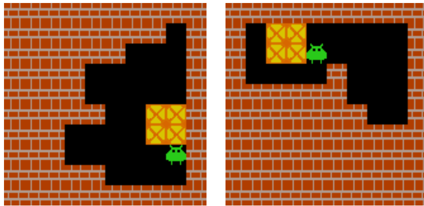
Goal	Path Cost	% Solved	# Models	Model Time	Search Time
Rubik's Cube (Canon)	24.41	100%	1	0.37	4.39
Rubik's Cube (Cross6)	13.11	100%	1	0.41	2.14
Rubik's Cube (Cup4)	24.33	100%	42.5	34.65	374.11
Rubik's Cube (CupSpot)	17.99	100%	27.68	38.66	241.08
Rubik's Cube (Checkers)	23.85	100%	1	0.49	4.2
Sokoban (Immov)	35.15	100%	6.37	6.83	16.16
Sokoban (BoxBox)	33.77	88%	1.89	0.58	6.08
Sokoban (AgentInBox)	34.42	77%	1.26	0.38	4.09

Table 2: Performance of DeepCubeA$_g$ when reaching goals specified with ASP.



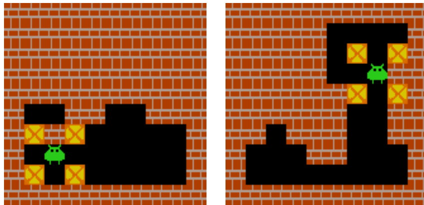
(a) Example 1 (b) Example 2

Figure 7: Reached goal where all boxes are immoveable.



(a) Example 1 (b) Example 2

Figure 8: Reached goal where all boxes form a larger box.

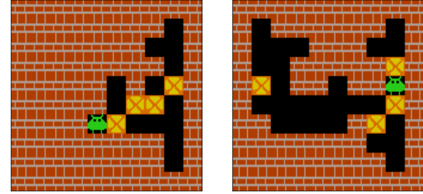


(a) Example 1 (b) Example 2

Figure 9: Reached goal where four boxes are at the four corners of the agent.

Discussion

In Table 2 we see that the path cost for finding the Cross6 goal is almost half that of finding the canonical goal, even though the canonical goal is a subset of the Cross6 goal. This indicates that the trained heuristic function is capable of estimating the cost-to-go to a closest state in a set of goal states without needing to be explicitly told of a closest state. This ability to discover paths to goal states which, themselves, are not known until a path is found, could be extended to domains such as chemical synthesis. For example, this would allow practitioners to specify properties a molecule should or should not have, discover synthesis routes to such molecules and, as a result, discover molecules that meet these specifications.



(a) Example 1 (b) Example 2

Figure 10: Start states that failed to reach both BoxBox and AgentInBox.

In Algorithm 1, we sample a new stable model if we fail to find a goal state. From Table 2, we see that the number of models we need to sample for the canonical Rubik's cube goal state and Cross6 is only one. However, for Cup4 and CupSpot, we must sample, on average, 42.5 and 27.68 models, respectively, to find a goal state. In cases where a goal state was not found, A* search failed to find a path to the sampled stable model. This may be because the sampled stable models represented only unreachable states. We discuss ways to overcome this in the Future Work Section.

For Sokoban, we see that the BoxBox and AgentInBox goals did not achieve a 100% success rate. Since we did not set a maximum iteration for Algorithm 1, all failure cases involved the algorithm terminating because all models were banned. Therefore, A* search failed to find a path to all stable models, which may indicate that the goal was not reachable for these start states. Figure 10 shows start states that failed to reach both the BoxBox and AgentInBox goals. The figure shows that there was not enough room to reach these goals.

Related Work

Action Schema Networks (ASNets) (Toyer et al. 2020) are neural networks that exploit the structure of the Planning Domain Definition Language (PDDL) to learn a policy that generalizes across problem instances. However, ASNets are

trained using imitation learning, which assumes a solver that can solve moderately difficult problems. On the other hand, we use reinforcement learning, which does not require the existence of any solver to learn. Furthermore, ASNets does not support arbitrary goal formulae. However, our approach of obtaining stable models from logic programs could be extended to the specification of goals in both PDDL and ASNets. Additionally, the learned heuristic function could be combined with existing planners, such as the fast downward planner.

Learning from partial interpretations (Fensel et al. 1995; De Raedt 1997) is a setting in inductive logic programming (Muggleton 1991; De Raedt 2008; Cropper and Dumančić 2022) where the training examples are not fully specified. This setting has also been applied to learning answer set programs from partial stable models (Law, Russo, and Broda 2014). This work has parallels with our work, except, instead of learning an answer set program as in Law, Russo, and Broda (2014), the specification is given in the form of an answer set program. Furthermore, instead of being given partial stable models as examples as in Law, Russo, and Broda (2014), the goal specification produces partial stable models that are then used by the DNN to reach the goal. In this context, partial means that a stable model does not have to be a completely specified goal state.

Research on training deep neural networks to generalize over both states and goals has mainly focused on goals that are represented by a single state. In reinforcement learning, Universal Value Function Approximators (Schaul et al. 2015) were proposed to learn a value function with an additional input of a goal state. Hindsight experience replay (Andrychowicz et al. 2017) built on this approach to learn from failures by using states observed during an episode as goal states, even if they were not the intended goal state. This approach has enabled learning for solving pathfinding problems, such as those involving object manipulation, and has shown to generalize to goal states not seen during training. However, this approach becomes impractical when only high-level qualities of a goal are known, but not enough low-level details are known to fully specify a goal state.

Future Work

In the Sokoban domain, unlike the Rubik's cube domain, not all states in the state space are reachable from all other states. As a result, not all goals will be reachable from every possible start state. In these cases, the training process could be augmented by mining for "negative" goals (Tian et al. 2021) that cannot be reached. The DNN should then give a very high cost-to-go when a goal is not reachable from a given start state. We can then sample stable models that are below some threshold to filter out unreachable goals.

In addition to goals that are unreachable from some states, one could also specify goals that are unreachable from all states due to them violating the constraints of the domain. For example, the set of all Rubik's cube states that have more than 9 white stickers is zero. While constraints could be manually added to the program to ensure no such stable models are found, preventing all such occurrences may require sophisticated domain-specific knowledge. Inductive logic programming (Muggleton 1991; De Raedt 2008; Cropper and Dumančić 2022) could be used to discover new constraints based on previously failed searches.

Our approach of using ground atoms to represent a goal comes with the advantage of being agnostic to the specification language as long as it can produce a set of ground atoms. Therefore, in the case of using ASP as the specification language, changes can be made to the predicates or even the ASP software used without having to re-train the DNN. However, this comes with the computational cost of solving for a set of ground atoms given a specification. One could instead train the heuristic function to estimate the distance between a state and a lifted specification that either implicitly or explicitly contains variables. Given the ability to sample diverse start state and goal pairs, this could be done for any kind of specification, such as first-order logic or even natural language. The downside to this approach is that any change in the vocabulary of the specification may require re-training of the DNN. Furthermore, this approach may put more representational burden on the DNN as it may need to implicitly consider stable models of a given specification.

Conclusion

We have introduced DeepCubeA$_g$, a deep reinforcement learning and search method that trains DNNs to estimate the distance between a state and a goal, where a goal is a set of states represented as a set of ground atoms. Goals can be communicated to a DNN without the need to re-train the DNN for that particular goal and without the need for the DNN to see that particular goal during training. When compared to other domain-independent planners, DeepCubeA$_g$ consistently solved more test states and found shorter paths.

To allow for more expressive goal specification, we have formalized a method for specifying goals using a specification language that is accessible to humans. Furthermore, the language used to specify goals only needs to be able to be translated into a set of ground atoms, which makes the DNN agnostic to the specification language. Using answer set programming, one can easily specify properties that a goal state should or should not have without having to specify any goal state in particular. As a result, this method has the ability to discover novel goals and; therefore, facilitate the discovery of new knowledge.

Acknowledgments

This work was funded in part by an ASPIRE-II grant from the University of South Carolina.

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.

Agostinelli, F.; Shmakov, A.; McAleer, S.; Fox, R.; and Baldi, P. 2021. A* search without expansions: Learning heuristic functions with deep Q-networks. *arXiv preprint arXiv:2102.04518*.

Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Abbeel, O. P.; and Zaremba, W. 2017. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, 5048–5058.

Barto, A. G.; Singh, S.; Chentanez, N.; et al. 2004. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Development and Learning*, volume 112, 19. Citeseer.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific. ISBN 1-886529-10-8.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Chen, B.; Li, C.; Dai, H.; and Song, L. 2020. Retro*: learning retrosynthetic planning with neural guided A* search. In *International Conference on Machine Learning*, 1608–1616. PMLR.

Cropper, A.; and Dumančić, S. 2022. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74: 765–850.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

De Raedt, L. 1997. Logical settings for concept-learning. *Artificial Intelligence*, 95(1): 187–201.

De Raedt, L. 2008. *Logical and relational learning*. Springer Science & Business Media.

DeepMind. 2018. boxoban-levels. https://github.com/deepmind/boxoban-levels/tree/master/unfiltered. Accessed: 2024-03-31.

Dor, D.; and Zwick, U. 1999. SOKOBAN and other motion planning problems. *Computational Geometry*, 13(4): 215–228.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22: 279–318.

Fensel, D.; Zickwolff, M.; Wiese, M.; et al. 1995. Are substitutions the better examples? Learning complete sets of clauses with Frog. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 453–474. Citeseer.

Ferenc, D. 2013. Pretty Rubik´s Cube patterns with algorithms. https://ruwix.com/the-rubiks-cube/rubiks-cube-patterns-algorithms/. Accessed: 2023-03-28.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2022. *Answer set solving in practice*. Springer Nature.

Gelfond, M.; and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, 1070–1080. Cambridge, MA.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive learning of answer set programs. In *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings 14*, 311–325. Springer.

Marek, V. W.; and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's Cube with Approximate Policy Iteration. In *International Conference on Learning Representations*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Muggleton, S. 1991. Inductive logic programming. *New generation computing*, 8: 295–318.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Puterman, M. L.; and Shin, M. C. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11): 1127–1137.

Rokicki, T. 2010. Twenty-Two Moves Suffice for Rubik's Cube®. *The Mathematical Intelligencer*, 32(1): 33–40.

Rokicki, T. 2016. cube20src. https://github.com/rokicki/cube20src. Accessed: 2024-03-31.

Schaul, T.; Horgan, D.; Gregor, K.; and Silver, D. 2015. Universal value function approximators. In *International Conference on Machine Learning*, 1312–1320.

Schmidhuber, J. 2015. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117.

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

Tian, S.; Nair, S.; Ebert, F.; Dasari, S.; Eysenbach, B.; Finn, C.; and Levine, S. 2021. Model-Based Visual Planning with Self-Supervised Functional Distances. In *International Conference on Learning Representations*.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Van Emden, M. H.; and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4): 733–742.

Zhang, Y.-H.; Zheng, P.-L.; Zhang, Y.; and Deng, D.-L. 2020. Topological Quantum Compiling with Reinforcement Learning. *Physical Review Letters*, 125(17): 170501.