Specifying Goals to Deep Neural Networks with Answer Set Programming

Anonymous submission

Abstract

The ability to easily and unambiguously specify a goal to a planner is fundamental to human and AI collaboration. In this context, a goal corresponds to a set of states in the state space that are considered goal states. This specification should be possible without explicitly describing any goal state in its entirety. Instead, a goal should be specified by simply describing certain properties that a goal state should or should not have. Recently, deep reinforcement learning has been used to train deep neural networks (DNNs) as heuristic functions for planning problems. While DNNs can be powerful function approximators that, combined with reinforcement learning, require little to no domain-specific knowledge to learn, there is no formal way to specify goals to DNNs. We introduce a method of training DNN heuristics to estimate the distance between a given state and a set of states, where a set of states is represented as a set of atoms in first-order logic. We then use answer set programming to specify goals, where a set of atoms representing a goal is obtained from the stable model of an answer set program. The DNN heuristic is then combined with search to achieve this goal. In our experiments with the Rubik's cube, we show that we can specify and achieve a variety of different goals without any need to re-train the DNN.

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 DNNs (Schmidhuber 2015) to be informative heuristic functions. Combined with 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 for compile quantum algorithms (Zhang et al. 2020). However, these DNNs are either trained for a pre-determined 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 every goal state in the set of goal states and taking the minimum heuristic value. This computationally burdensome process significantly reduces the practicality of DNNs for solving planning problems with dynamic goals.

To address this issue, we present an approach for training a DNN to estimate the distance between a state and a goal using first-order logic. In our setting, a goal is represented as a set of ground atoms that represents what should hold true in a goal state, but does not assume that ground atoms that are not in this set are false. In other words, the conclusion that something is a goal should exhibit monotonic behavior; therefore, any superset of a set of ground atoms representing a goal also represents a goal. Therefore, given a process to convert a state to a set of ground atoms that represents what holds true in that state, any state whose set of ground atoms is a superset of a given goal is considered a member of the set of goal states.

To train a DNN to estimate the distance between a state and a goal, we build on hindsight experience replay (Andrychowicz et al. 2017), a reinforcement learning approach that estimates the distance between pairs of start and goal states, with the modification that goal states are transformed into sets of states. Given the aforementioned semantics, one can then obtain a set of states that contains a given state by obtaining the set of ground atoms for that given state and simply removing atoms from the set. A conversion process then translates these logical atoms to a representation that is convenient for the DNN. For example, one can represent a set of Rubik's cube states as a set of atoms that represent that it holds that a given color sticker is at a given position. This can then be converted to a one-hot representation where there is an extra color for the positions that are not specified. In the domain of chemistry, one can represent a set of molecules as a set of atoms that specify the connections among atoms as well as the charge of atoms. This can then be converted to a graph for a graph convolutional neural network.

Since a set of states is represented as a set of ground atoms, any specification language that can be translated to a set of ground atoms can be used to specify a goal, provided that the conclusion that something is a goal behaves monotonically. We choose answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011) as the specification language because one can obtain stable models (Gelfond and Lifschitz 1988), also known as answer sets, for a given specification (answer set program) where each stable model is a set of ground atoms. We put restrictions on negation in goal specification to ensure that the program behaves mono-

tonically when deriving whether or not a model is a goal. Furthermore, since the particular ASP system that we use is clingo (Gebser et al. 2012, 2014), we can randomly sample a subset of stable models in cases where there are many to still achieve the specified goal. We evaluate this approach on the Rubik's cube and results show that one can specify diverse goals to a DNN with simple answer set programs and achieve them by combining the DNN with search. An overview of our approach is described in Figure 1. In the Future Work Section, we will discuss handling unreachable goals, using negation as failure, and representing goals to the DNN using lifted representations.

Related Work

Planning languages such as the Planning Domain Definition Language (PDDL) specify goals using formal logic. Furthermore, a planning problem can be described to a wide variety of planners and heuristics can be computed to guide the planning process. However, when using heuristic functions represented by DNNs, there is no formal way to represent what the goal of the planning problem is. Our approach of obtaining stable models from logic programs could be extended to descriptions of goals in PDDL. Furthermore, in the Future Work Section, we discuss ways goals can be represented with logic, itself, without having to solve for stable models.

Learning from partial interpretations (Fensel et al. 1995; De Raedt 1997) is a setting in inductive logic programming (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 achieve the goal.

When training deep neural networks to generalize over both states and goals has mainly focused on goals that are limited to 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 in sparse reward environments, such as those involving object manipulation, and has shown to generalize to goal states not seen during training. After training, one can then specify what the goal state is, provided the practitioner has the ability to fully specify a goal state. However, this approach becomes impractical in cases where there are a diverse set of acceptable goal states that the agent could possibly achieve or where only high-level qualities of a goal are known, but the low-level details are not.

Preliminaries

Our method builds on the DeepCubeA algorithm (Agostinelli et al. 2019) that was used to train a DNN as a heuristic function using 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. Recently, research has shown that Deep Q-Networks (DQNs) (Mnih et al. 2015) can be used in a modified version of A* search, called Q* search (Agostinelli et al. 2021), to significantly reduce time and memory requirements of search by calculating the sum of the transition costs and heuristic values of the children of a node with a single forward pass through a DQN. For specifying goals, we use ASP. In this section, we will describe the background of Q-learning and Q* Search, as well as the background of ASP. We also describe the basics of the Rubik's cube.

Q-learning

In the context of deterministic, finite-horizon, shortest path problems, Q-learning is a reinforcement learning (Sutton and Barto 2018) algorithm to learn a function, known as an action-value function, $Q(s, a)$, that maps a state s and action a to the estimated cost-to-go when in state s and taking action a. In this setting, we can write $Q(s, a)$ in terms of the transition cost and the cost-to-go from s', which is the state that results from applying action a to state s, to a closest goal state:

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

where $g^a(s, s')$ is the cost to transition from state s to state s' using action a and $h(s')$ is the cost-to-go from state s' to a closest goal state and, also, $h(s')$ is equivalent to $\min_{a'} Q(s', a')$. The optimal action-value function, $q^*(s, a)$, represents the cost of a shortest path when in state s and taking action a. The Q-learning algorithm (Watkins and Dayan 1992) takes a given Q and updates it according to Equation 2, where α is the learning rate.

$$Q(s, a) = Q(s, a) + \alpha(g^a(s, s') + \min_{a'} Q(s', a') - Q(s, a)) \tag{2}$$

In the tabular setting, Q-learning has been shown to converge to q^* as time goes to infinity (Watkins and Dayan 1992). However, for domains with large state spaces, such as the Rubik's cube, we do not have enough memory, or time, to do tabular Q-learning. Therefore, we represent $Q(s, a)$ with a parameterized function $q_\phi(s, a)$ with parameters ϕ. The parameters of the function are trained to minimize the loss function in Equation 3, 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).

$$L(\phi) = (g^a(s, s') + \min_{a'} q_{\phi^-}(s', a') - q_\phi(s, a))^2 \tag{3}$$

The structure of DQNs is typically one where the input is the state, s, and the output is a vector the size of the action space that represents $q_\phi(s, a)$ for every action a. This allows one to compute the sum of the transition cost and cost-to-go for all possible next states of s with a single forward pass through the DQN, which, in turn, allows for faster heuristic search with the Q* search algorithm.

Q* Search

Q* search (Agostinelli et al. 2021) is a heuristic search algorithm inspired by A* search (Hart, Nilsson, and Raphael 1968) that uses DQNs to do heuristic search. Unlike A* search, Q* search does not need to fully expand nodes by applying every action to it and obtaining the resulting state. Instead, it stores tuples of nodes and actions (node_action tuples) in the priority queue. When removing a node_action tuple from the priority queue, Q* search then applies the action to the corresponding node to obtain the current node. The DQN is then applied to the current node to obtain the estimate of the sum of the transition cost and cost-to-go for all of its children. Then, for all actions, a node_action tuple containing the current node and an action is then pushed to the priority queue with its priority set to its path cost plus the corresponding output of the DQN for that action. The only aspect of this algorithm that directly depends on the size of the action space is pushing node_action tuples to the priority queue. Due to this, results have shown that Q* search can be orders of magnitude faster and more memory efficient than A* search while maintaining similar performance in terms of path cost.

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:

$$A \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{4}$$

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. Rules can also be "headless", meaning that there is no logical atom, A, in the head. This can be interpreted as an implication where the consequent is always false. This entails that satisfying the body will result in a contradiction and are used as constraints in practice. Implicitly, headless rules are actually rules 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 actually 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 where all literals in the body are true. If there are no literals in the body, this is taken to mean the body is always true; therefore, the head must also always be 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. We can now see that Π_g^M is 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. It should be noted that Π can have multiple stable models if it contains negation. Furthermore, some ASP solvers, such as clingo, allow 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, which 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 have 1 sticker, edge cubelets have 2 stickers, and corner cubelets have 3 stickers. There are 6 center cubelets, 12 edge cubelets, and 8 corner cubelets. 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 n.d.).

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 action-value function. Therefore, the action-value function now becomes $Q(s, a, \mathcal{G})$, that represents the cost to go from s to a closest state in $\mathcal{G}$ when taking action a. In our implementation, we use a DQN with parameters ϕ whose input is s and $\mathcal{G}$ and whose output is a vector representing $q_\phi(s, a, \mathcal{G})$ for all actions, a. We assume some process to convert $\mathcal{G}$ to a representation suitable for the DQN. To train the DQN, we must first have the ability to sample state and goal pairs. From these pairs, we can then compute the loss using Q-learning.

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 ob-

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

served state, s_t, is then selected to create a goal. To create this goal, we must first translate s_t to a set of ground atoms, $\mathcal{S}_t$, that represents what holds in that state. Since any $\mathcal{S}_t$ that is a superset of a goal, $\mathcal{G}$, also represents a goal, we can simply remove atoms from $\mathcal{S}_t$ to create a goal $\mathcal{G}$ such that $\mathcal{G} \subseteq \mathcal{S}_t$ and; therefore, s_t is a member of the set of goal states.

After obtaining state and goal pairs, we must select an action to update before computing the loss. While we could select the random uniformly selected action that was taken from state s_0 when generating the goal, we would like to prioritize more promising actions over less promising actions during learning to ensure the estimate of the cost-to-go $h(s) = \min_a q_\phi(s, a, \mathcal{G})$ is not biased towards less promising actions. Therefore, we select actions according to a Boltzmann distribution where each action a is selected with probability according to Equation 5, where $|\mathcal{A}|$ is the size of action space and T is the temperature, which we set to $1/3$.

$$\frac{e^{(-q_\phi(s,a,\mathcal{G})/T)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(-q_\phi(s,a',\mathcal{G})/T)}} \tag{5}$$

After action selection, the loss for the DQN is computed according to Equation 6. The parameters of the target network, ϕ^-, are periodically updated to ϕ. This training procedure is outline in Figure 1.

$$L(\phi) = (g^a(s, s') + \min_{a'} q_{\phi^-}(s', a', \mathcal{G}) - q_\phi(s, a, \mathcal{G}))^2 \tag{6}$$

Specifying Goals with Answer Set Programming

We would like to express a goal using a program Π where the union of stable models of Π corresponds to the specified goal. More formally, we have background knowledge B which is a set of rules that describes relevant domain knowledge, we have a goal specification, H, that is a set of rules with the atom $goal$ in the head of arity zero, and we have a model $\mathcal{G}$ that is a set of ground atoms that represents a set of goal states. We seek to find models $\mathcal{G}$ such that $\mathcal{G} \cup \{goal\}$ is a stable model of $B \cup H$. Using clingo (Gebser et al. 2012, 2014) to do ASP, we limit possible stable models to those in which $goal$ is true with the following line, which means it cannot be the case that $goal$ is not true:

```
:- not goal.
```

In our setting, if $\mathcal{G}'$ is a superset of $\mathcal{G}$ and represents at least one *legal* state, then it is the case that $\mathcal{G}' \cup \{goal\}$ is also a stable model of $B \cup H$. Therefore, we must ensure that $B \cup H$ behaves monotonically for legal states. If $B \cup H$ behaves monotonically, then if some fact, such as $goal$, is derived from some set of atoms $\mathcal{G}$, then those facts will not be later retracted given additional facts. However, ASP can behave non-monotonically if negation is present. One solution to this is to not allow for negation in B or H. However, negation in the form of headless rules is often very useful for pruning models, $\mathcal{S}_\emptyset$, that represent illegal states. For example, for the Rubik's cube, it is impossible to have two different stickers be in the same location. Although this leads to non-monotonicity, this is only in the case of illegal states, making it impossible for a search procedure to encounter a state that is a member $\mathcal{S}_\emptyset$ provided it only starts at legal states and takes legal actions. To ensure supersets of some stable model $\mathcal{G}$ can also be stable models, the predicates that define a set of states only appear in the head of conditional literals, which appear in aggregates that allow zero to as many of these atoms as possible. This process implicitly uses disjunction. For example, the following line in clingo allows a program to have zero to as many atoms of `at_idx_cbl(Cbl, Col, I)` as possible given the possible combinations of `Cbl`, `Col`, and `I`:

```
{ at_idx_cbl(Cbl, Col, I) :
cubelet(Cbl), color(Col), index(I) }.
```

An overview of the specification process is shown in Figure 1.

Achieving Goals with Q* Search

Given a stable model that represents a goal, $\mathcal{G}$, we can sample start states and perform Q* search with a DQN that has been trained to estimate the distance between any given state and goal pair to achieve goal states. However, since our answer set program makes use of conditional literals, which implicitly make use of disjunction, there could be more than one stable model. Therefore, to achieve goal states, we first sample N start states, and G stable models. For each start state, we estimate the distance between that state and the G stable models. We then pick the stable model with the lowest estimated distance as the goal for that state. Now that each start state has its own goal, we can do Q* search to find a path from the start state to a state that is a member of the set of goal states. Furthermore, 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 Q* search that removes more multiple nodes from the priority queue at each iteration. In addition, we use weighted Q* search that puts a weight between 0 and 1 on the path cost as, in practice, this leads to shorter solve times and less memory usage.

Experiments

Representation and Training

We investigate goal specification for the Rubik's cube. To specify a set of states, 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 red is at index 12 on the Rubik's cube. `at_idx` is derived from predicate `at_idx_cbl` of arity 3 that holds when the color on a given cubelet is at a given index. This is a conditional literal that can have zero to as many combinations of cubelets, colors, and indices as possible. Below we show how we define colors, cubelets, what color stickers the cubelets have, and `at_idx`:

```
color(w).  color(y). ...  color(g).
white(w). yellow(y). ... green(g).

center_cbl(w_c). center_cbl(y_c). ...
center_cbl(g_c).
```

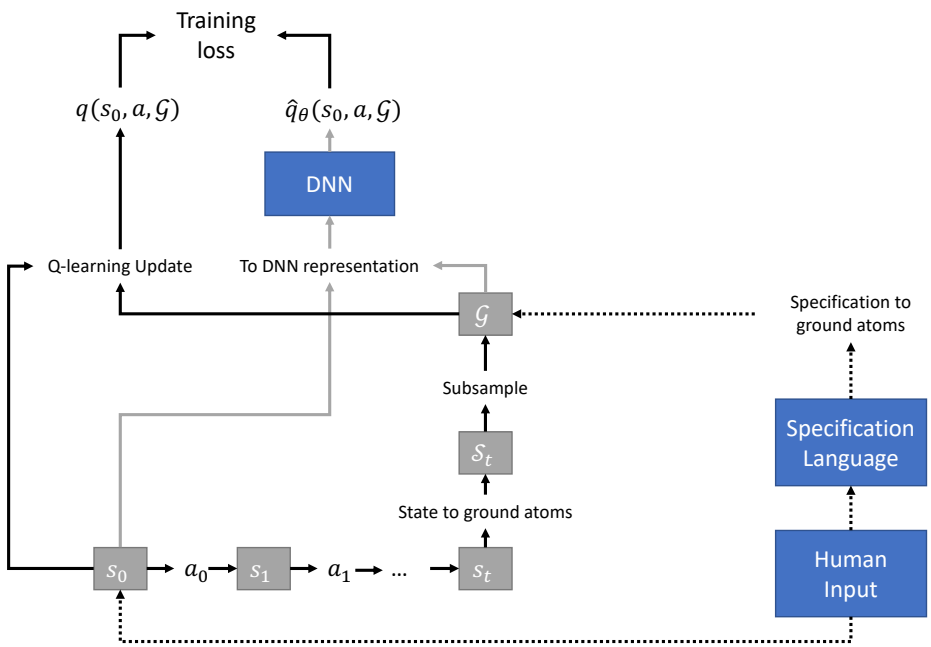


Figure 1: The figure outlines our training procedure as well as the goal specification procedure. Steps involving training are solid black lines, steps involving human goal specification are dashed black lines, and steps common to both are grey lines. The DNN is trained with Q-learning and hindsight experience replay. Pairs of start and goal states are obtained by selecting a random start state, taking t steps where t for each training example is randomly distributed between 0 and T, and converting the state at time step t to a set of logical atoms. A subset of the logical atoms is taken to obtain a goal which represents a set of states. After training, a human then describes a start state and a goal, where the goal is then converted to a set of logical atoms and, subsequently, given to the DNN. Note that, given a method to convert the specification to a set of logical atoms, the specification language is independent of the training procedure.

```
edge_cbl(wo_c). edge_cbl(wg_c). ...
edge_cbl(rb_c).
corner_cbl(wog_c). corner_cbl(wob_c).
... corner_cbl(yrg_c).

cubelet(Cbl) :- center_cbl(Cbl).
cubelet(Cbl) :- edge_cbl(Cbl).
cubelet(Cbl) :- corner_cbl(Cbl).

has_col(w_c, w). has_col(y_c, y).
...has_col(g_c, g).
has_col(wo_c, w). has_col(wo_c, o). ...
has_col(rb_c, b).
has_col(wog_c, w). has_col(wog_c, o).
... has_col(yrg_c, g).

index(0..54).
{ at_idx_cbl(Cbl, Col, I) :
cubelet(Cbl), color(Col), index(I) }.
at_idx(Col, I) :- at_idx_cbl(_, Col,
I).
```

In addition, we define directions (clockwise, counter-clockwise, and opposite), faces, their 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. Some constraints used are shown below:

```
% different stickers from the same
cubelet cannot be on the same face
:- onface(Cbl, Col0, F), onface(Cbl,
Col1, F), dif_col(Col0, Col1).

% cannot have a sticker color from same
cubelet be on more than one face
:- onface(Cbl, Col, F0), onface(Cbl,
Col, F1), dif_face(F0, F1).

% cannot say a color of a cubelet is on
a face if it does not have that color
:- onface(Cbl, Col, _), not
has_col(Cbl, Col).

% cannot have two stickers from same
cbl on opposite faces
:- onface(Cbl, Col0, F0), onface(Cbl,
Col1, F1), face_rel(_, F0, F1, op).
```

```
% cannot have two different colors at
the same index
:- at_idx(Col0, I), at_idx(Col1, I),
dif_col(Col0, Col1).
```

We also make sure the stable model as the fewest number of atoms as possible to obtain the most general stable models possible using the optimization functionality of clingo:

```
count_at_idx(C) :- #count{V0, V1:
at_idx(V0,V1) }=C.
#minimize {C: count_at_idx(C)}.
```

To represent a stable model to the DQN, we use a vector of 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 DQN. The start state is also represented with a one-hot representation, except every sticker is specified, removing the need for the additional value of 6. The architecture of the DQN we use and the optimization procedure is the same as that described in Agostinelli et al. (2021). However, Agostinelli et al. (2021) only trains the DQN for a predetermined goal state. Our training procedure, described in the Methods Section, samples state and goal pairs for training. To randomly generate start states, for each state, we start from the canonical goal state and randomly take between 100 and 200 actions. We train and test using two NVIDIA Tesla V100 32 GB GPUs and 48 2.4 GHz Intel Xeon Platinum CPUs.

Achieving Goals

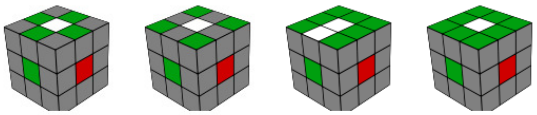
To test our method, we draw from Ferenc (n.d.) to come up with goals that combine different Rubik's cube patterns shown in Figure 2. We also test our method with the canonical solved state for the Rubik's cube where all faces have a uniform color. All patterns are described using clingo. Given the background knowledge, many patterns only require a few lines of code, as shown below.

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

% X
x(F, XCol) :- face(F), color(XCol),
#count{Cbl: corner_cbl(Cbl),
onface(Cbl, XCol, F)} = 4.

% cup
cup(F1, F2, CCol) :- dif_face(F1, F2),
face_col(F1, F1Col), dif_col(F1Col,
CCol), edge_cbl(ECbl), onface(ECbl,
_, F2), onface(ECbl, F1Col, F1),
#count{Cbl: edge_or_corner(Cbl),
onface(Cbl, CCol, F1)} = 7.

% spot
spot(F, BCol) :- color(BCol), face(F),
face_col(F, FCol), dif_col(FCol, BCol),
```

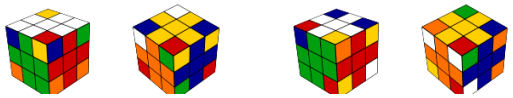


(a) Cross (b) X (c) Cup (d) Spot

Figure 2: Examples of patterns that are combined to create goals.

```
#count{Cbl: onface(Cbl, BCol, F),
edge_or_corner(Cbl)} = 8.

% canonical solved state
face_same(F) :- face_col(F, FCol),
#count{Cbl : onface(Cbl, FCol, F)}=9.
canon_solved :- #count{F :
face_same(F)}=6.
```



(a) Example 1 (b) Example 2

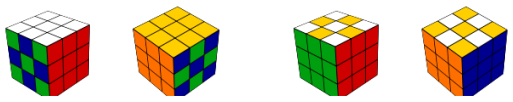
Figure 3: Achieved goal of having a cross on all 6 faces where the center cubelet and cross are the same color.

(a) Example 1 (b) Example 2

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

(a) Example 1 (b) Example 2

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



(a) Example 1 (b) Example 2

Figure 6: Achieved goal of having two checkerboards on opposite faces with all of the other faces the same.

In addition to the canonical goal, we specify four other goals: (1) a goal state where all faces have a cross where the cross is the same color as the center piece; (2) a goal where the red, green, blue, and orange faces have a cup on them (3) a goal where there is a spot adjacent to a cup with the opening of the cup facing the spot; (4) a goal where there are two checkerboard patterns (a cross combined with an X) on opposite faces and all other faces have uniform color. To achieve goals, we sample up to 10 stable models of the answer set program that represent the goal and find a path to 100 goal states by randomly generating start states and using batch weighted Q* search to find a path from these start states to a goal state. We use a batch size of 10,000 and a weight of 0.6 when doing batch weighted Q* search. Each randomly generated start state is a given a budget of 50 iterations. If a goal is not found in that time, then a new start state is generated. Visualizations of achieved goals for the four non-canonical goals are shown in Figures 3, 4, 5, and 6. A table summarizing the time it takes to find stable models, find 100 goals, as well as the average path cost is shown in Table 1.

Table 1: The time it takes to find stable models for each goal, the time it takes to find 100 goal states, and the average path cost from the start states to the goal states.

	Stable Model Time (secs)	Solve Time (secs)	Path Cost
Canon	0.33	625.62	23.82
Cross6	0.35	218.45	11.50
Cup4	11.17	1622.39	24.44
CupSpot	123.04	291.25	14.7
Checkers	0.44	602.03	24.00

Discussion

To illustrate the power of specifying a set of states as a goal instead of pre-determined states, we note that the Cross6 goal contains the canonical goal state in the set of states that it represents. However, finding the canonical goal state takes about three times as long and has a path cost that is about twice as long when compared to the Cross6 goal. This indicates that this method has the potential to allow us to discover more efficient plans as well as to discover new knowledge by achieving *unanticipated* goal states that even humans have not yet considered. For example, in a domain such as chemical synthesis, this could allow practitioners to discover new synthesis routes as well as learn more about chemistry by examining the properties of the unanticipated molecules that meet their specifications.

When examining solve time and path cost in Table 1, the Cross6 goal takes the least amount of time and has the shortest average path cost. The CupSpot goal is also comparable along these same metrics. This could be because Cross6 and CupSpot need to consider fewer stickers than other goals. However, the Cup4 goal takes the longest to achieve out of all the goals even though it also needs to consider fewer stickers than both the canonical goal and the Checkers goal. One indication of the cause of this is that Q* search frequently went over its budget of 50 iterations for the Cup4 goal. This could be because some of the stable models actually represent sets of states that are not reachable. We discuss ways to overcome this in the Future Work Section.

When examining the time it takes to find stable models in Table 1, the CupSpot goal takes the longest out of all the goals. This could be due to having many constraints to consider when finding the stable models. However, dealing with the constraints when solving for the stable models could lead to faster solve times as fewer stable models will represent unreachable goals. It could also be the case that certain constraints could be expressed in a more concise manner.

Future Work

In this work, we investigated the Rubik's cube, which is an environment in which every state is reachable from every other state. However, this may not always be the case in other domains. As a result, not all goals will be reachable from every possible start state. In these cases, the training process could be augmented with mining for "negative" goals (Tian et al. 2021) that could be impossible to reach. The DQN should then give a very high cost-to-go when a goal is not reachable from a given start state. We could then only sample start states whose estimated cost-to-go is below some threshold. Furthermore, one could easily specify goals that only represent impossible states or have some stable models that only represent impossible states. For example, one could specify a Rubik's cube state with two adjacent faces that are entirely white. This is impossible for multiple reasons: 1) There cannot be more than one center cubelet of the same color; 2) Since the faces are adjacent, there would have to be cubelets with two white stickers, which is impossible because no cubelet has two stickers of the same color; 3) There would be more than nine stickers of the same color. While these constraints could be manually added to the program, we also want to strive for a system that *discovers* new constraints that even humans have not yet discovered. Given a specification, one could use a generality relationship, such as entailment or theta subsumption (Plotkin 1972), to find the most general specification that cannot be achieved and add this to its background knowledge as a constraint.

Although we limit the use of negation, allowing for negation as failure may allow for more practical specifications. To overcome the non-monotonicity introduced with negation as failure, instead of just finding stable models for a program that contains $goal$, one could find stable models that contain $goal$ of which no superset exists that is also a stable model but does not contain $goal$. This could be done using an iterative process of first finding a stable model with $goal$, adding all ground atoms except $goal$ to the program, and then searching for a stable model that does not contain $goal$. If one is found, then search for a superset of the stable model that was added to the program that still contains $goal$ and repeat the process. If not, return the stable model that was added to the program.

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 logic language used without having to re-train the DNN. However, this comes with the computational cost of having to solve 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. This could be done for any kind of specification, such as first-order logic or even natural language, given the ability to go from a state to a specification representing a set of states of which that state is a member. For example, the specification given to the heuristic function could be a sentence in first-order logic describing the goal. One could obtain training examples by obtaining a goal state and then search for a first-order logic sentence that represents a set of states of which that goal state is a member. The downside to this approach is that any change in the vocabulary of the specification may require re-training of the DNN.

Conclusion

We have formalized a method for specifying goals to powerful, but opaque, heuristic functions represented by DNNs. This goal specification is done without any need to re-train the DNN for that particular goal. Furthermore, the language used to specify goals only needs to be able to be translated in to 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 have without having to specify any goal state in particular. This method could result in finding unanticipated and potentially novel goal states that allow for the discovery of new knowledge.

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.

Chen, H.-C.; and Wei, J.-D. 2011. Using neural networks for evaluation in heuristic search algorithm. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

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

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

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. n.d. Pretty Rubik´s Cube patterns with algorithms. Accessed March 28, 2023. https://ruwix.com/the-rubiks-cube/rubiks-cube-patterns-algorithms/.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

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

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.

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.

Plotkin, G. 1972. Automatic methods of inductive inference.

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.

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*.

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.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8(3-4): 279–292.

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.