A Conflict-Driven Approach for Reaching Goals Specified with Negation as Failure

Forest Agostinelli[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

Abstract

First-order logic allows for the expressive specification of goals. Using negation as failure, one can specify what must not be true in a goal state instead of what must be true, which can result in succinct goal specifications while also being computationally advantageous. However, due to non-monotonicity, integration of negation as failure and recent deep reinforcement learning methods that incorporate first-order logic in goal specification can be cumbersome. To address this problem, we create a conflict-driven algorithm for non-monotonic goal specification that refines search for a goal state based on conflicts encountered during search. Our results show that this conflict-driven approach results in significantly shorter paths and can significantly speed up search when compared to not taking conflicts into consideration. Furthermore, our results show that finding paths to goals can be much more efficient when goals are specified with negation as failure instead of without negation as failure.

Introduction

Specifying goals in first-order logic programming languages that make use of non-monotonic reasoning through negation as failure (NAF) (Clark 1977) can often be significantly more computationally convenient than specifying goals without NAF. For example, two equivalent sets of states in the Rubik's cube domain can be specified in the following ways: 1) all stickers on the white face are different than the center sticker; 2) there does not exist a sticker on the white face that matches the center sticker. The set of minimal assignments of colors to stickers that makes the first specification true consists of all combinations of non-white colors on the white face, excluding the color of the center sticker, which is of size $5^8 \approx 3.8 \times 10^5$ (5 non-white colors, 8 non-center stickers). On the other hand, under NAF semantics, the second specification need only attempt to prove its un-negated statement and, if it fails to prove it, then it is assumed to be false and, therefore, the specification is assumed to be true. The set of minimal assignments that makes the un-negated statement of the second specification true is of size 8. As a result, a logic program will need to reason over many more scenarios for the first specification compared to the second specification.

Given a state that does not make the first specification true, identifying conflicts, which, in this context, are minimal assignments of colors to stickers that cause it to not be true, requires us to find a minimal assignment such that no superset of that assignment exists that makes it true. To accomplish this, since the first specification is false given the state, we will have to iteratively unassign stickers and attempt to find an assignment that is a superset of the current partial assignment that makes the first specification true. On the other hand, identifying conflicts for the second specification requires us to find a minimal assignment that makes the un-negated statement true. To accomplish this, since the un-negated statement of the second specification is true given the state, we will have to iteratively unassign stickers and check if the current partial assignment makes the un-negated statement true. For disjunctive logic programs, searching for an assignment is NP^{NP}-complete (Eiter and Gottlob 1995) while checking an assignment is *co*-NP-complete (Eiter and Gottlob 1993; Dantsin et al. 2001). Therefore, searching for conflicts can be easier in practice with goals specified with NAF.

We can use a conflict-driven approach when searching for a path to a goal by first finding a path to a specified assignment that makes the specified goal true. If NAF is used, a state that has these assignments may have other assignments which make the goal specification false, or, in other words, create conflicts. We can then find a conflict and search for a new assignment that is consistent with the goal while ensuring this conflict will not be present. We build on conflict-driven clause learning approaches for solving boolean satisfiability problems (Prosser 1993; Marques-Silva and Sakallah 1999) and create a branch-and-bound algorithm that takes into account that finding a state that makes the goal true is not the only criteria since we would also like to find a shortest path to the goal.

More broadly, the ability to specify goals using expressive languages gives practitioners the ability to describe high-level properties of a goal by abstracting away low-level properties using derived predicates. This approach can make specified goals more concise and lead to finding shorter paths, since more general goals correspond to a larger set of states that are deemed goal states. These benefits have been demonstrated in the planning domain definition language (PDDL) (McDermott 2000) using axioms and domain-independent heuristics (Thiébaux, Hoffmann, and Nebel 2005; Ivankovic and Haslum 2015; Speck et al. 2019). Recently, the benefits of an expressive goal

specification language have been demonstrated when specifying goals to deep neural networks (DNNs) (Schmidhuber 2015) that have been trained to estimate the distance between a state and a set of states, which is represented as a set of ground atoms in first-order logic (Agostinelli, Panta, and Khandelwal 2024). This set of ground atoms can also be seen as representing assignments of values to variables. This method, called DeepCubeA$_g$, was shown to outperform domain-independent planners when searching for paths from states to a set of goal states. DeepCubeA$_g$ built on answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011) to describe goals with an answer set program and used an answer set solver to find assignments that could potentially represent a set of goal states. However, since ASP makes use of NAF, an assignment found by the answer set solver could also represent states that are not goal states. Though this prior work addressed this by proposing to search for specialized assignments, this is done randomly and, as we will show, can be very inefficient. Our work will build on DeepCubeA$_g$ with a conflict-driven approach that finds shorter paths and often does so much faster than the previous random approach. A visualization of this approach is shown in Figure 1.

Preliminaries

States and Sets of States

Similar to previous work on planning with axioms (Ivankovic and Haslum 2015), we model a state as a set of V state variables $\{x_1, ..., x_V\}$. Each state variable, x, has a domain, $D(x)$, of finite size. A variable may be assigned a value from its domain. An assignment, A, is a set $\{.., x_i = v_i, ...\}$, where, if $x_i = v_i$ is in the assignment then x_i is assigned to v_i and $v_i \in D(x_i)$. Otherwise, x_i is unassigned. An assignment is a partial assignment if and only if between 1 and V state variables are unassigned and an assignment is a complete assignment if and only if all variables are assigned. A state is represented as a complete assignment and a set of states is represented as either a partial or complete assignment, where a complete assignment always represents a set of states of size one. The number of variables that have been assigned in assignment, A, is denoted $|A|$.

The set of all states in the state space is denoted $\mathcal{S}$. The set of states represented by assignment, A, is denoted $\mathcal{S}_A$. A state, s, is a member of $\mathcal{S}_A$ if and only if $A \subseteq s$. This definition of sets of states imposes a generality relation (De Raedt 2008) on assignments. That is, A_1 is more general than A_2 if and only if $A_1 \subseteq A_2$, which entails $\mathcal{S}_{A_2} \subseteq \mathcal{S}_{A_1}$. When $A_1 \subseteq A_2$, A_1 is considered a *generalization* of A_2, and A_2 is considered a *specialization* of A_1. When $A_1 \subset A_2$, A_1 is considered a *strict generalization* of A_2, and A_2 is considered a *strict specialization* of A_1. The most general assignment is the empty set and the set of states represented by this assignment is equivalent to $\mathcal{S}$.

Answer Set Programming

Answer set programming (Brewka, Eiter, and Truszczyński 2011) is a form of logic programming where each logic program defines a set of stable models, also known as answer sets, according to the stable model semantics (Gelfond and Lifschitz 1988). A program, Π, is comprised of a set of rules in first-order logic of the form:

$$H_1; ...; H_l \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \quad (1)$$

where H_i, B_i, and C_i are atoms in first-order logic. If the body, which is the conjunction of all B_i and $\neg C_i$, is true, then at least one atom in the head, which is the disjunction of all H_i, must be true. Rules with no literals in the body are called "facts" and their body is taken to always be true. Rules with no literals in the head are called "headless" rules and their head is taken to always be false. In practice, headless rules are used as constraints. An atom is derivable if it is in the head of an implication whose body is true. If there is more than one atom in the head, it is assumed at most one of them are true unless other rules derive more than one of them.

To define what a stable model is, we first must define ground programs and reducts. The ground program of Π, Π_g, is the set of all possible ground rules present in Π. A ground rule is obtained from a rule by substituting all variables in that rule for a ground term appearing in Π. A reduct (Marek and Truszczyński 1999) of a ground program, Π_g, with respect to a set of ground atoms, M, Π_g^M, is obtained by starting with Π_g and deleting all rules that have a negated atom in the body if the atom is in M and then deleting all negated atoms in the body of the remaining rules. A set of ground atoms, M, is a stable model of program, Π, if the set of ground atoms derivable from Π_g^M is exactly M.

Choice rules are allowed by ASP solvers such as clingo (Gebser et al. 2014) and have a conjunction of literals in the body and a set of ground atoms in the head. Zero or more ground atoms in the head may be added to the stable model if the body is true.

DeepCubeA$_g$

DeepCubeA$_g$ (Agostinelli, Panta, and Khandelwal 2024) builds on the DeepCubeA (Agostinelli et al. 2019) algorithm to train a heuristic function that generalizes over states and sets of goal states represented as a partial or complete assignment. ASP is used to specify goals and an answer set solver is used to find stable models that represent sets of goal states.

Training DeepCubeA$_g$ trains a heuristic function parameterized by parameters, ϕ, h_ϕ, to map a state, s, and an assignment, A, to the estimated cost-to-go between s and a closest state in $\mathcal{S}_A$. Therefore, a problem instance with goal, A, is considered solved if the terminal state is s_t and $s_t \in \mathcal{S}_A$. The heuristic function is trained using deep approximate value iteration (DAVI), a version of approximate value iteration (Bertsekas and Tsitsiklis 1996) that uses DNNs as the approximation architecture. The loss function used to train the heuristic function is shown in Equation 2, where $c^a(s)$ is the transition cost when taking action, a, in state, s, T is the transition function that maps a state, s, and action, a, to the state that results from taking action, a, in state, s, and ϕ^-

are parameters of a target network (Mnih et al. 2015) that are periodically updated to ϕ.

$$L(\phi) = (\min_a (c^a(s) + h_{\phi^-}(T(s,a), A)) - h_\phi(s, A))^2 \quad (2)$$

Training data is generated through a method based on hindsight experience replay (Andrychowicz et al. 2017) which generates random start states, takes a random walk of length t steps, where t is between 0 and a given maximum number of steps, and uses the final state in that random walk, s_t, to generate a target assignment by randomly removing assignments from s_t.

Specifying Goals An answer set program, Π, is used to represent a specification. Π consists of background knowledge, a set of rules that have `goal` in the head, a headless rule, `:- not goal`, to ensure `goal` is true in all stable models, and a choice rule with an empty body that contains a set of ground atoms, K, where each atom represents an assignment of a single value to a single state variable. The subset of ground atoms of a stable model, M, of Π, that are in K is denoted M_K. M_K represents an assignment where all variables not assigned a value by some ground atom in M_K are taken to be unassigned. The set of all possible assignments that can be obtained from a program, Π, is denoted $\alpha(\Pi)$.

Definition 1 (Candidate state)**.** A state, s, is a candidate state with respect to a specification, Π, if and only if, there exists some $A \in \alpha(\Pi)$ such that $s \in \mathcal{S}_A$.

Definition 2 (Goal state)**.** A state, s, is a goal state with respect to a specification, Π, if and only if $s \in \alpha(\Pi)$.

Reaching Goals Given a specification, Π, an assignment, A_1, is sampled from $\alpha(\Pi)$. A* search (Hart, Nilsson, and Raphael 1968) with the trained heuristic is performed to find a path from s to A_1. If a path is found then s_t is a candidate state, where s_t is the terminal state along the path to A_1 and $s_t \in \mathcal{S}_{A_1}$. However, since ASP can make use of NAF, it is possible that s_t can be a candidate state, but not a goal state. DeepCubeA$_g$ addresses this by randomly sampling another assignment, A_2, according to the constrained optimization expression in Equation 3. This is known as a specialization operator (De Raedt 2008) because it produces A_2 such that A_2 is a strict specialization of A_1 (if such an A_2 exists). The specialization operator seeks to minimize the size of $|A_2|$ so that it is as general as possible and, thus, represents the largest set of states. The specialization is done in hopes that it will also reduce the number of non-goal states represented by the assignment. However, this process does not take into account why s_t is not a goal state and, thus, can be very inefficient. A visualization of this approach is shown in Figure 1.

Theoretical Properties of Goal Specifications

A specification can be either monotonic or non-monotonic.

Definition 3 (Monotonic specification)**.** A specification, Π, is a monotonic specification if and only if for all assignments, A_1 and A_2, if $A_1 \in \alpha(\Pi)$ and $A_1 \subseteq A_2$, then $A_2 \in \alpha(\Pi)$.

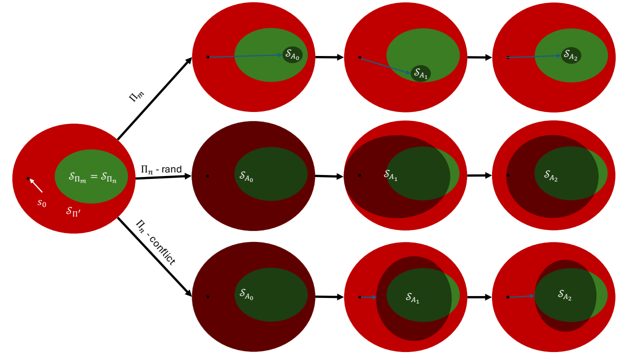


Figure 1: A visualization of searching for a goal with a monotonic specification, Π_m, and a non-monotonic specification, Π_n. The set of goal states is shown in green and denoted by $\mathcal{S}_{\Pi_m}$, which is equal to $\mathcal{S}_{\Pi_n}$. The set of non-goal states is shown in red and denoted by $\mathcal{S}_{\Pi'}$. The black dot represents the start state and is denoted by s_0. The set of states represented by an assignment is shown in transparent black and is denoted $\mathcal{S}_{A_i}$. The blue arrow represents a path to an assignment. The monotonic specification does not need to be specialized since all candidate states are goal states. On the other hand, the non-monotonic specification can produce assignments that can be specialized randomly or with a conflict-driven approach. This conflict-driven approach can lead to finding shorter paths with less time.

Definition 4 (Non-monotonic specification)**.** A specification, Π, is a non-monotonic specification if and only if there exists assignments, A_1 and A_2, such that $A_1 \in \alpha(\Pi)$, $A_1 \subseteq A_2$, and $A_2 \notin \alpha(\Pi)$.

For a specification, Π, we denote the set of all candidate states as $\mathcal{S}_{\alpha(\Pi)}$ and the set of all goal states as $\mathcal{S}_\Pi$.

Lemma 1. For all specifications, Π, $\mathcal{S}_\Pi \subseteq \mathcal{S}_{\alpha(\Pi)}$.

Proof. Since S_Π is the set of all goal states and, from Definition 2, we know that all goal states are in $\alpha(\Pi)$. Therefore, all goal states must be in $S_{\alpha(\Pi)}$. Therefore, $\mathcal{S}_\Pi \subseteq \mathcal{S}_{\alpha(\Pi)}$. □

Theorem 1. For all specifications, Π, if Π is monotonic, then $\mathcal{S}_\Pi = \mathcal{S}_{\alpha(\Pi)}$.

Proof. From Definition 1, we know that if there exists some state, $s \in \mathcal{S}_{\alpha(\Pi)}$, then there exists some assignment, A, such that $A \in \alpha(\Pi)$ and $s \in \mathcal{S}_A$, which means $A \subseteq s$. Since Π is monotonic, from Definition 3, we can see that $s \in \alpha(\Pi)$ by substituting A for A_1 and s for A_2. From Definition 2, we know that if $s \in \alpha(\Pi)$, then $s \in \mathcal{S}_\Pi$. Therefore $\mathcal{S}_{\alpha(\Pi)} \subseteq \mathcal{S}_\Pi$. From Lemma 1 we know that $\mathcal{S}_\Pi \subseteq \mathcal{S}_{\alpha(\Pi)}$. Therefore, it must be the case that $\mathcal{S}_\Pi = \mathcal{S}_{\alpha(\Pi)}$. □

From Theorem 1, we see the set of candidate states is equivalent to the set of goal states for a monotonic specification, Π. Therefore, for all $A \in \alpha(\Pi)$, $\mathcal{S}_A$ are goal assignments (see Definition 5). Now, we can characterize the probability of finding a shortest path to a closest goal state from a given starting state. To accomplish this, we need to

characterize the probability of sampling $A \in \alpha(\Pi)$ that is a closest assignment (see Definition 6).

Definition 5 (Goal assignment)**.** An assignment A is a goal assignment if and only if $\mathcal{S}_A \subseteq \mathcal{S}_\Pi$.

Definition 6 (Closest assignment)**.** An assignment A is a closest assignment relative to some start state s_0 if and only if $\mathcal{S}_A$ contains a goal state closest to s_0 and does not contain any other non-goal states that are closer than the closest goal state.

Theorem 2. Let Π be a monotonic specification, let s_0 be a given starting state, let p the number of closest assignments in $\alpha(\Pi)$, n be the number of all other assignments in $\alpha(\Pi)$, and assume a uniform random procedure to sample from $\alpha(\Pi)$. The probability of sampling $A \in \alpha(\Pi)$ such that A is a closest assignment within $1 \leq k \leq n$ samples without replacement is $1 - \prod_{i=1}^{k}(1 - \frac{p}{p+n-i})$.

Proof. Since Π is monotonic, from Theorem 1 we know that $\mathcal{S}_\Pi = \mathcal{S}_{\alpha(\Pi)}$. Therefore, for any $A \in \alpha(\Pi)$, A is a goal assignment. Therefore, if A contains a closest goal state then it must be a closest assignment since it does not contain any non-goal states. Therefore, there are p closest assignments, $n + p$ total assignments, and the probability of finding a closest assignment in sample i is $\frac{p}{p+n-i}$. We subtract i from the total assignments to account for the fact we are sampling without replacement. Therefore, the probability of not sampling a closest assignment in sample i is $(1 - \frac{p}{p+n-i})$, the probability of not sampling a closest assignment over k iterations is $\prod_{i=1}^{k}(1 - \frac{p}{p+n-i})$, and we can obtain the probability of sampling a closest assignment over k iterations by subtracting this quantity from 1, which is $1 - \prod_{i=1}^{k}(1 - \frac{p}{p+n-i})$. □

Using Theorem 2, we can see that the number of samples we will need to sample a closest assignment can be quite large. Building on the Rubik's cube example specification, if we created a monotonic specification where all stickers on the white face must be different than the center sticker, it could be the case that p is as small as one, in which case n is $5^8 - 1 \approx 3.8 \times 10^5$. In this case, we would need approximately 1.95×10^5 samples to exceed a probability of 0.5 of sampling a closest assignment. To seek more efficient alternatives, we turn to negation as failure and conflict-driven search.

Conflict-Driven Search

Finding Conflicts

In the case that, for a given specification, Π, there exists an assignment $A_1 \in \alpha(\Pi)$ and a state, s, such that $s \in \mathcal{S}_{A_1}$, but $s \notin \alpha(\Pi)$, there must be variables assigned in s that are unassigned in A_1 that cause `goal` to no longer be true. We refer to such assignments as conflicts. We can find these conflicts by finding an assignment, A_c, according to the constrained optimization expression in Equation 4. This expression requires $A \subset A_c$ since we are going to specialize A. We can then create a specialization operator that finds A_2 such that the intersection of $\mathcal{S}_{A_2}$ and $\mathcal{S}_{A_c}$ is empty. This ensures that any states found on a path to A_2 will not contain the same conflict as A_c. This specialization operator is shown in Equation 5. However, in the case where there are multiple goal clauses, there could exist another clause that is true for some specialization of A_c. Therefore, we can also use the specialization operator shown in Equation 6.

$$\begin{aligned} \min_{A_2} \quad & |A_2| \\ \text{s.t.} \quad & A_1 \subset A_2 \\ & A_2 \in \alpha(\Pi) \end{aligned} \tag{3}$$

$$\begin{aligned} \min_{A_c} \quad & |A_c| \\ \text{s.t.} \quad & s \in \mathcal{S}_{A_c} \\ & A \subset A_c \\ & A_c \notin \alpha(\Pi) \end{aligned} \tag{4}$$

$$\begin{aligned} \min_{A_2} \quad & |A_2| \\ \text{s.t.} \quad & A_1 \subset A_2 \\ & \mathcal{S}_{A_2} \cap \mathcal{S}_{A_c} = \emptyset \\ & A_2 \in \alpha(\Pi) \end{aligned} \tag{5}$$

$$\begin{aligned} \min_{A_2} \quad & |A_2| \\ \text{s.t.} \quad & A_c \subset A_2 \\ & A_2 \in \alpha(\Pi) \end{aligned} \tag{6}$$

Our implementation uses the clingo (Gebser et al. 2014, 2022) answer set solver. To accomplish the specialization operators shown in Equations 3, 5, and 6, we must be able to find an assignment that is a strict specialization of another assignment. To accomplish this, we require that clingo return stable models that contain the ground atoms that represent the assignment to be specialized and put a size requirement on the minimum number of atoms from set K that must be added to the stable model so that the size of all assignment found are larger than the assignment to be specialized. We perform minimization of the assignment by unassigning variables until no variables can be unassigned without violating the optimization constraints.

For Equation 5, we seek to find a variable, x_i, that is assigned to a value, v_i, in A_c, but is unassigned in A_1, and ensure x_i cannot be assigned to v_i when searching for a new assignment, A_2. We accomplish this by selecting the ground atom, `k`, that represents the assigned value and ensure that, for all states, $s \in \mathcal{S}_{A_2}$, $x_i = v_i \notin s$. Using of classical negation (Gelfond and Lifschitz 1991), we ensure that an atom, `k`, and its classically negated counterpart, `-k`, cannot both be true at the same time. We, thus, require that clingo return stable models that contain `-k`. In order to do this, we define the classically negated version of every atom in K. In practice, this can be done with only a few lines of code and does not require any derived predicates also have their classically negated counterparts be defined.

Branch and Bound Search

To reach a goal given by a goal specification, Π, from a given start state, s_0, we use a branch and bound (Land and Doig 1960) algorithm to attempt to find a closest state in the set of states represented by Π. The upper bound represents the lowest cost path found. Given an assignment, A, the lower bound is approximated by finding a path from s_0 to A. A node represents an assignment and a conflict associated with a state that is a member of the set of states represented by the assignment. The algorithm maintains a priority queue of nodes, where the priority of a node is given according to the size of its assignment and ties are broken according to the computed lower bound. The intuition is that more specific assignments represent a smaller set of states and, thus, the set of states may contain a higher percentage of goal states compared to more general assignments and, thus, a higher chance of reducing the upper bound. However, this may not always be the case (see Future Work).

At each iteration a node is removed from the priority queue and specializations of its associated assignment with respect to its associated conflict are obtained. If its assignment is None, then minimal assignments are sampled randomly from $\alpha(\Pi)$ without any additional constraints. Since the number of specializations can be very large, expansion is done by sampling at most B assignments from $\alpha(\Pi)$ without replacement, where B is given by the user. This adds stochasticity to the algorithm. To ensure that all possible specializations can be obtained, we put a node back in the queue if there are potentially more specializations to be obtained. We decrease its priority according to the number of times the node has been seen.

The algorithm also makes use of the ability to "ban" an assignment, A, by specifying that clingo should not return any stable models that contain the ground atoms that represent A. As a result, no specialization of A will be sampled from $\alpha(\Pi)$. This is similar to nogood recording in constraint satisfaction (Dechter 1986). We ban an assignment, A, if a path to A is not found, if a path to A is found, or if the lower bound for a path to A is greater than or equal to the upper bound. We ban A if a path is found to A because a shortest path to A is at least as cheap as a shortest path to A_2 if $A \subseteq A_2$. The algorithm is outlined in Algorithm 1.

Algorithm 1: Branch and Bound for Reaching Goals

Input: Specification Π, DNN h_ϕ, start s_0, batch size B, specialization op ρ_s
$q \leftarrow []$; seen $\leftarrow \{\}$; $ub \leftarrow inf$; path $\leftarrow None$
push NODE($A = None, A_c = \{\}, lb = 0, i = 0$) to q
while q is not Empty **do**
  $n_{\text{pop}} \leftarrow$ POP(q)
  specializations $\leftarrow$ apply $\rho_s(\Pi, n_{\text{pop}}.A, n_{\text{pop}}.A_c)$ for at most B
  if len(specializations) $==$ B **then**
    push n_{pop} to q with priority $(-|n_{\text{pop}}.A|, n_{\text{pop}}.i + 1, n_{\text{pop}}.lb)$.
  end if
  for A in specializations **do**
    if A not in seen **then**
      add A to seen
      n_t = A*Search(s_0, A, h_ϕ) {returns terminal node}
      if ($n_t.s$ is not None) and ($g(n_t) < ub$) and ($n_t.s \notin \alpha(\Pi)$) **then**
        $A_c \leftarrow$ conflict($n_c.s, \Pi$)
        push NODE($A = A, A_c = A_c, lb = g(n_c), i = 0$) to q with priority $(-|A|, 0, g(n_c))$
      else
        ban A
        if ($n_t.s$ is not None) and ($g(n_t) < ub$) and ($n_t.s \in \alpha(\Pi)$) **then**
          $ub \leftarrow g(n_t)$; path $\leftarrow$ path to n_t
          remove n from q and ban $n.A$ if $n.lb >= ub$
        end if
      end if
    end if
  end for
end while
return path

Experiments

We test our method on both the Rubik's cube and 24-puzzle. For the Rubik's cube, we use the `at_idx` predicate of arity two to represent assignments where `at_idx(col,idx)` holds if and only if the given index has been assigned the given color. For the 24-puzzle, we use the `at_idx` predicate of arity three to represent assignments where `at_idx(tile,row,col)` holds if and only if the given tile is at the given row and column. For the Rubik's cube, we use the following clause to define the classically negated ground atoms:

```
-at_idx(C,I):- at_idx(C2,I),color(C),
color(C2),not C=C2
```

This means that a color cannot be at a given index if it holds that a different color is at that index (":-" denotes implication). For the 24-puzzle, we use the following clauses to define classically negated ground atoms:

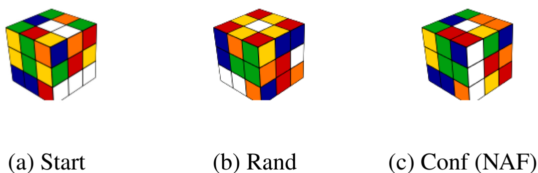
```
-at_idx(X,R,C):- t(X),t(X2),
at_idx(X2,R,C),not X=X2
-at_idx(X,R,C):- row(R),col(C),row(R2),
at_idx(X,R2,_),not R=R2
-at_idx(X,R,C):- row(R),col(C),col(C2),
at_idx(X,_,C2),not C=C2
```

This means that a given tile cannot be at a given row and column if it holds that a different tile is there or if it holds that the given tile is at a different row or column.

Our heuristic function is trained using the same architecture and optimization strategy as Agostinelli, Panta, and Khandelwal (2024). For the branch and bound algorithm, we compare two specialization operators, one that only uses the specialization operator based on a random specialization shown in Equation 3 and a conflict-driven one based on Equations 5 and 6. The conflict-driven specialization operator randomly chooses between Equations 5 and 6 each time it performs a specialization. We perform batch weighted A* search (Agostinelli et al. 2019) to find paths to assignments with a batch size of 100 and weight of 0.6 on the path cost.

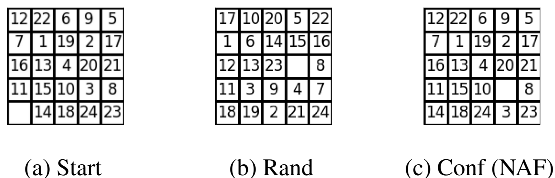
We also add a patience parameter to the algorithm to specify how many iterations the algorithm is willing to wait for the upper bound to improve given that a path has already been found. We set this argument to 5 for all of our experiments.

We randomly sample 100 start states for our test data. The test goal for the Rubik's cube is all stickers on the white face are different than the center sticker. Expressed with NAF it becomes: there does not exist a sticker on the white face that matches the center sticker. For the 24-puzzle there are two test goals: 1) the sum of row 0 is even; 2) the sum of all rows are even. Expressed with NAF they become: 1) it is not true the sum of row 0 is odd; 2) there does not exist a row whose sum is odd. Algorithm 1 is run with a single NVIDIA Tesla V100 GPU for the trained heuristic function and one 2.4 GHz Intel Xeon Platinum CPU, otherwise. We give a time limit of 500 seconds for each test state. Results for our experiments are shown in Table 1. Figures showing goals reached are shown in Figures 2 and 3. Code, data, and specifications are provided in the Supplementary Material.



(a) Start (b) Rand (c) Conf (NAF)

Figure 2: Example of reaching the goal where all stickers on the white face are different than the center sticker. The path cost is 12 without NAF and 1 with NAF and conflict-driven search.



(a) Start (b) Rand (c) Conf (NAF)

Figure 3: Example of reaching the goal where the sum of all rows are even. The path cost is 93 without NAF and 4 with NAF and conflict-driven search.

Discussion

The results in Table 1 show that expressing goals using NAF significantly decreases the path cost for both the random specialization operator and the conflict-driven specialization operator. When NAF is used, the conflict-driven specialization operator is faster and finds shorter paths when compared to the random specialization operator. Furthermore, when looking at the average percentage of assignments that were reached when performing A* search, this is larger in the majority of cases where NAF is used. This is because, all the assignments that make a specification true when not using NAF also make a specification true when using NAF, provided both specifications represent an equivalent set of goal states. However, when using NAF there exists more general additional assignments that also make the NAF specification true. Therefore, A* search can find paths more reliably. On the other hand, all terminal states along the path to assignments are goal states for monotonic specifications (see Theorem 1) and some terminal states may not be goal states for non-monotonic specifications. However, the conflict-driven approach exploits this by determining why the terminal state is not a goal state.

Since the answer set solver is not aware of the start state, it could sample an assignment from $\alpha(\Pi)$, which could be far away from the start state. When using NAF, however, having the ability to iteratively make small changes to the assignment through specialization results in being able to prioritize closer assignments over those further away using its lower bound, whereas, when not using NAF, the assignment is not further refined after the first iteration since it is guaranteed to be a goal assignment. When looking at the time it takes to do a specialization, the answer set solver is much faster in the NAF case. For example, for the Rubik's cube, it takes 12.8 seconds, on average, per specialization without NAF and 0.1 seconds, on average, per specialization with NAF. This shows that, when the set of minimal assignments that make a goal is true is very large, it can be computationally convenient to, instead, specify when the goal is not true using NAF.

Related Work

Expressive specification languages have been of interest to the planning community for expressive goal specification as well as for expressive action precondition specification for declarative planning languages such as PDDL. Axioms in PDDL allow users to define axioms that can be used to derive predicates. It has been shown that axioms are computationally beneficial and can lead to better overall performance (Thiébaux, Hoffmann, and Nebel 2005) . Domain independent heuristics that take advantage of axioms have been derived (Ivankovic and Haslum 2015). Furthermore, existential quantification can play a big role in NAF, like it has done in our experiments. Heuristics that are computed from existentially quantified variables have also been derived for STRIPS (Fikes and Nilsson 1971) planners (Frances and Geffner 2016).

Limitations and Future Work

The priority queue is prioritized by the size of the assignment and ties are broken according to their lower bounds. While this could lead to finding goals quickly, it could also lead to spending time on specializations of assignments that do not decrease the upper bound. To remedy this, the algorithm could be modified to alternate between prioritizing size first and prioritizing lower bound first. This could obtain a better trade-off between the two priorities and result in exploring more diverse areas of the search space. Furthermore, future work could use Algorithm 1 to create a dataset to train a heuristic function that directly encodes the first-order logic specification. This could then be used to better

Goal	Op	Cost	Solve	Itr	Node	Reach	¬Goal	$\frac{\text{Secs}}{\text{Spec}}$	$\frac{\text{Secs}}{\text{Path}}$	Secs
RC:Π^1_m	-	11.5	70	**3.3**	**33.4**	7.7	**0.0**	12.8	7.5	564.9
RC:Π^1_n	Rand	1.7	99	7.2	63.0	87.8	69.1	**0.1**	1.0	95.5
	Conf	**1.3**	**100**	5.4	36.3	**99.3**	52.4	**0.1**	**0.1**	**6.0**
24p:Π^1_m	-	24.6	**100**	9.2	92.4	**100.0**	**0.0**	**0.2**	0.2	42.5
24p:Π^1_n	Rand	3.2	**100**	4.3	33.6	**100.0**	38.7	**0.2**	**0.0**	**6.6**
	Conf	**2.5**	**100**	**4.1**	**31.6**	**100.0**	22.1	**0.2**	**0.0**	**6.6**
24p:Π^2_m	-	83.7	**100**	9.2	91.9	50.4	**0.0**	0.9	1.8	250.2
24p:Π^2_n	Rand	17.1	**100**	10.2	92.1	100.0	85.5	**0.1**	**0.1**	21.7
	Conf	**12.9**	**100**	**8.7**	**77.1**	**100.0**	79.7	**0.1**	**0.1**	**17.1**

Table 1: Comparison of monotonic and non-monotonic specifications with random and conflict-driven specialization operators for non-monotonic specifications. Comparisons are along the dimensions of average path cost, percentage solved, average number of iterations, average number of nodes generated, the average percentage of specified assignments reached with A* search, the average percentage of reached assignments that were not goal states, the average number of seconds it took to do a single specialization, the average number of seconds it took to find a single path (whether or not it was successful), and the average number of overall seconds it took to find a solution. RC:Π^1_m: All stickers on the white face are different than the center sticker. RC:Π^1_n: There does not exist a sticker on the white face that matches the center sticker. 24p:Π^1_m: The sum of row 0 is even. 24p:Π^1_n: It is not true the sum of row 0 is odd. 24p:Π^2_m: The sum of all rows is even. 24p:Π^2_n: There does not exist a row whose sum is odd.

prioritize nodes or even remove the need to do branch-and-bound search altogether.

Conclusion

NAF allows us to express goals succinctly and has the ability to significantly reduce computational load. However, the integration of NAF with solvers based on DNNs still has significant room for improvement. We address this problem with an algorithm that exploits NAF semantics to quickly find conflicts and specialize assignments based on these conflicts. This conflict-driven approach results in finding significantly shorter paths in significantly less time when compared to random specialization and also when compared to specifying equivalent goals without NAF.

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.; Panta, R.; and Khandelwal, V. 2024. Specifying goals to deep neural networks with answer set programming. In *34th International Conference on Automated Planning and Scheduling*.

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.

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.

Clark, K. L. 1977. Negation as failure. In *Logic and data bases*, 293–322. Springer.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3): 374–425.

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

Dechter, R. 1986. Learning while searching in constraint-satisfaction problems.

Eiter, T.; and Gottlob, G. 1993. Propositional circumscription and extended closed-world reasoning are Π^p_2-complete. *Theoretical Computer Science*, 114(2): 231–245.

Eiter, T.; and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15: 289–323.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Frances, G.; and Geffner, H. 2016. ∃-strips: existential quantification in planning and constraint satisfaction. In *Kambhampati S, editor. Twenty-Fifth International Joint Conference on Artificial Intelligence; 2016 Jul 9-15; New York, NY. Palo Alto (CA): AAAI Press/IJCAI; 2016. p. 3082-8.* IJCAI & AAAI Press.

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.

Gelfond, M.; and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9: 365–385.

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.

Ivankovic, F.; and Haslum, P. 2015. Optimal planning with axioms. In *Proceedings of the 24th International Conference on Artificial Intelligence*, 1580–1586.

Land, A. H.; and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3): 497–520.

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.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5): 506–521.

McDermott, D. M. 2000. The 1998 AI planning systems competition. *AI magazine*, 21(2): 35–35.

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.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3): 268–299.

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

Speck, D.; Geißer, F.; Mattmüller, R.; and Torralba, Á. 2019. Symbolic planning with axioms. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 464–472.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *Artificial Intelligence*, 168(1-2): 38–69.