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

Forest Agostinelli

Department of Computer Science and Engineering
University of South Carolina, USA
foresta@cse.sc.edu

Abstract

In the context of pathfinding, 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 can be cumbersome. To address this problem, we introduce conflict-driven goal reaching (CDGR), a conflict-driven algorithm for reaching goals specified with non-monotonic logic that refines a search for a goal state based on conflicts encountered during search. Our results show that CDGR 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.

Code — https://github.com/forestagostinelli/SpecGoalNegationAsFailure

Introduction

A pathfinding problem instance is defined by a domain, start state, and goal, where a goal is a set of states considered goal states. The ability to specify goals using expressive languages gives practitioners the ability to describe high-level properties of a goal by building on low-level state properties. This approach can make goal specifications more concise and lead to finding shorter paths compared to specifying goals with less expressive languages, since more expressive languages can lead to more general goal specifications (i.e. a larger set of goal states). Recently, the benefits of an expressive goal specification language have been demonstrated when specifying goals to deep neural networks (DNNs) (Schmidhuber 2015) using first-order logic (Agostinelli, Panta, and Khandelwal 2024b). This method, called 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 an assignment of values to variables that could potentially represent a set of goal states. However, since ASP makes use of negation as failure, 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$_g$ to create conflict-driven goal reaching (CDGR), a conflict-driven branch-and-bound approach that finds shorter paths and often does so much faster than the previous random approach.

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. This is because the number of assignments to consider and number of constraints encoded can be drastically lower using NAF. For example, two equivalent sets of states in the Rubik's cube domain can be specified in the following ways: 1) for all faces, all stickers on that face are different than its center sticker; 2) there does not exist a face that has a sticker that matches its center sticker. The set of minimal assignments of colors to stickers that makes the first specification true consists of all combinations of colors on each face that do not match the center sticker, which is of size $5^{8^6} \approx 3.6 \times 10^{33}$ (5 non-center colors per face, 8 non-center stickers, 6 faces), many of which represent impossible configurations. On the other hand, under NAF semantics, the second specification need only attempt to prove its un-negated statement, which is that there exists a sticker on a face that matches the center sticker and, if it fails to prove it, then NAF semantics dictate that it is false and, therefore, the specification is proven to be true. Each assignment in the set of minimal assignments that makes the un-negated state true only needs one sticker assigned to a color and there are $1*8*6 = 48$ (1 center color per face, 8 non-center stickers, 6 faces) minimal assignments that make it true. Furthermore, the number of constraints encoded in the first specification is higher since it requires the assignment of 48 stickers, which may have constraints that interact with one another, such as what color combinations are allowed on cubelets. As a result, a logic program will need more computation for the first specification compared to the second specification.

Assignments found via NAF can be very general. In fact, for the previous example, the empty assignment makes the second specification true since one cannot prove there is a sticker on a face that matches the center sticker. This gen-

eral assignment represents all states in the state space, many of which are not goal states. However, we can learn why a state is not a goal state by finding *conflicts* and refining these general assignments based on these found conflicts. In this context, conflicts are minimal assignments such that no state that is a superset of that assignment can be a goal state. To accomplish this with NAF, given a state that is not a goal state, we can find a minimal assignment that makes the un-negated statement true. Since the unnegated statement is already true given a state that is not a goal state, we can iteratively unassign variables present in the state and check if the resulting partial assignment makes the un-negated statement true. A minimal assignment that makes the un-negated statement true is a conflict. We can also find such conflicts for specifications that do not use NAF by iteratively unassigning variables and attempting to solve for an assignment that is a superset of the resulting partial assignment that makes the specification true. If no such superset exists for a minimal assignment, then this is also a conflict. However, for disjunctive logic programs, checking is $co\text{-}NP$-complete (Eiter and Gottlob 1993; Dantsin et al. 2001) while solving is NP^{NP}-complete (Eiter and Gottlob 1995). Therefore, searching for conflicts can be easier in practice with NAF.

In this work, we build on conflict-driven clause learning approaches for solving boolean satisfiability problems (Prosser 1993; Marques-Silva and Sakallah 1999) to introduce CDGR, which finds a path to a given specification that makes use of NAF. CDGR first finds a path to assignments based on a given specification and, if the terminal state is not a goal state, finds conflicts based on the terminal state and specification. CDGR then searches for new assignments that ensure these conflicts will not occur again. CDGR also utilizes branch-and-bound to take path cost into account. Furthermore, CDGR is independent of the underlying pathfinding algorithm used to find paths to assignments. For example, one may use A* search (Hart, Nilsson, and Raphael 1968) with a learned DNN heuristic function, as is done in DeepCubeA$_g$, or may use A* search with heuristics derived from PDDL, as is done in the fast downward planner (Helmert 2006). In this work, we use the former since it has been shown to more consistently find paths when compared to the latter on domains such as the Rubik's cube and 24-puzzle (Agostinelli, Panta, and Khandelwal 2024b; Muppasani et al. 2024). We will also show that, without NAF, though assignments found are more specific, many of the assignments may be impossible to reach (i.e. represent a set of states of size zero) or represents sets of states that are far away from the start state. Our theoretical work will characterize the probability of sampling an assignment that represents a closest goal state when NAF is not used and our empirical results will show that goals can be found much faster and via significantly shorter paths when using NAF and CDGR.

Preliminaries

Pathfinding

A pathfinding problem is defined as a weighted directed graph (Pohl 1970), where nodes represent states, edges represent actions that transition between states, weights on the edges represent transition costs, a given state represents the start state, and a given set of states represents the goal. The cost of a path is the sum of transition costs. The transitions can be represented as a function, T, where there exists an edge connecting states s and s' if and only if $s' = T(s, a)$ for some action, a. The transition cost function, $c^a(s)$, is the cost of taking action a in state s.

States and Sets of States

Similar to previous work on planning with axioms (Ivankovic and Haslum 2015), a state in a pathfinding problem is defined as assignments to 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 of variable assignments $\{.., x_i = v_i, ...\}$, where, if $x_i = v_i$ is in the assignment then the variable, x_i, is assigned to value, v_i, and $v_i \in D(x_i)$. Otherwise, x_i is unassigned. An assignment is a partial assignment if and only if at least 1 state variable is 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 a partial or complete assignment, where a complete assignment always represents a set of states of size one. The number of variables 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 (which we also refer to as the empty assignment) 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:

$$H_1; ...; H_l \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{1}$$

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. For rules with no literals in the body (i.e. "facts"), the body is taken to always be true. For rules with no literals in the head (i.e. "headless" rules), 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. A stable model of an answer set program is a set of ground atoms, M, that is equivalent to the set of ground atoms derivable from the reduct (Marek and Truszczyński 1999) of the ground program of Π with respect to M. An answer set program can have multiple stable models.

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 2024b) 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 partial or complete assignments. 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 ϕ^- is the 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), where random start states are generated, a random walk of length t steps is taken, and the final state in that random walk, s_t, is used to generate a target assignment by randomly removing assignments from s_t. For each randomly generated start state, t is generated from a random uniform distribution between 0 and a given maximum number of steps.

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 $s \in \alpha(\Pi)$.

Definition 3 (Reachable Assignment)**.** An assignment, A, is reachable from a given state, s_0, if and only if there exists some state, $s \in \mathcal{S}_A$, such that s is reachable from s_0. Otherwise, the assignment is unreachable.

Reaching Goals Given a specification, Π, an assignment, A_1, is sampled from $\alpha(\Pi)$. A* search 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 case 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

In this section, with Theorem 1, we will show that, for monotonic specifications, all candidate states are goal states. Therefore, we do not have to handle the case where a path to an assignment is found, but the terminal state is not a goal state, as we may have to for non-monotonic specifications. However, with Theorem 2, we will show that, for monotonic specifications, the number of samples needed to have a probability of 0.5 of sampling an assignment that contains a goal state that is closest to the start state becomes impractically high as the number of possible assignments grows, thus giving us a theoretical motivation for exploring goal reaching algorithms that exploit non-monotonic specifications, such as CDGR.

A specification can be either monotonic or non-monotonic, where the definition of a non-monotonic specification is simply the negation of the definition of a monotonic specification.

Definition 4 (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 $A_2 \in \alpha(\Pi)$.

Definition 5 (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 $\mathcal{S}_\Pi$.

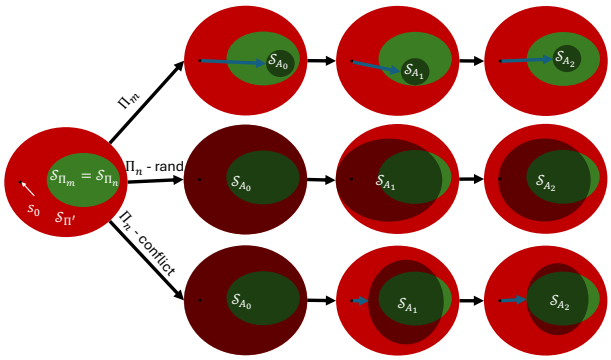

Figure 1: A visualization of searching for a goal with a monotonic specification, Π_m, and a non-monotonic specification, Π_n. The set of goal states is shown in green and denoted by $\mathcal{S}_{\Pi_m}$, which is equal to $\mathcal{S}_{\Pi_n}$. The set of non-goal states is shown in red and denoted 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. The conflict-driven approach takes why the terminal state is not a goal state into account during specialization to ensure the next terminal state found will not have the same conflicts.

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

Proof. S_Π is the set of all goal states and, from Definition 2, we know that all goal states are in $\alpha(\Pi)$. Therefore, from Definition 1, by substituting in a goal state for both s and A, we know that all goal states must be candidate states and, thus, 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 for all states, $s \in \mathcal{S}_{\alpha(\Pi)}$, 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 4, we can see that $s \in \alpha(\Pi)$ by substituting in 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 $\mathcal{S}_\Pi = \mathcal{S}_{\alpha(\Pi)}$. □

Definition 6 (Goal assignment)**.** An assignment A is a goal assignment if and only if $\mathcal{S}_A \subseteq \mathcal{S}_\Pi$.

Definition 7 (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 closest goal state and does not contain any other non-goal states that are as close or closer than a closest goal state, where a closest goal state is a goal state with the lowest path cost from s_0 .

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. Now, we can characterize the probability of sampling $A \in \alpha(\Pi)$ that is a closest assignment.

Theorem 2. Let Π be a monotonic specification, let s_0 be a given start state, let p be the number of closest assignments in $\alpha(\Pi)$ (where $p \geq 1$), 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 without replacement $A \in \alpha(\Pi)$ such that A is a closest assignment within k samples (where $1 \leq k \leq (n+1)$) is $1 - \prod_{i=1}^{k}(1 - \frac{p}{p+n-(i-1)})$.

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 $\frac{p}{p+n-(i-1)}$. We subtract $i - 1$ 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 $(1 - \frac{p}{p+n-(i-1)})$, the probability of not sampling a closest assignment within the first k iterations is $\prod_{i=1}^{k}(1 - \frac{p}{p+n-(i-1)})$, and we can obtain the probability of sampling a closest assignment within the first k iterations by subtracting this quantity from 1, which is $1 - \prod_{i=1}^{k}(1 - \frac{p}{p+n-(i-1)})$. □

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 $5^{8^6} - 1 \approx 3.6 \times 10^{33}$. In this case, we would need on the order of $\approx 1.8 \times 10^{33}$ samples to exceed a probability of 0.5 of sampling a closest assignment. To seek more efficient alternatives, we turn to NAF and CDGR.

Conflict-Driven Goal Reaching

Finding Conflicts and Specializing Assignments

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 $s \notin \alpha(\Pi)$. 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 $A_1 \subset A_c$ since we are going to specialize A_1.

Given a conflict, we 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. If there is disjunction present in the goal specification, then it may be possible that

there exists a superset of A_c in $\alpha(\Pi)$. Therefore, we can also use the specialization operator shown in Equation 6.

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

$$\begin{aligned} \min_{A_c} \quad & |A_c| \\ \text{s.t.} \quad & s \in \mathcal{S}_{A_c} \\ & A_1 \subset A_c \\ & A_c \notin \alpha(\Pi) \end{aligned} \tag{4}$$

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

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

Our implementation uses the clingo (Gebser et al. 2014, 2022) answer set solver. To implement the specialization operators, 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 assignments 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 assignment, $x_i = v_i$, that is in A_c, but not in A_1, and ensure x_i cannot be assigned to v_i when searching for a new assignment, A_2, so that the conflict, A_c, cannot be present. In other words, for all states, $s \in \mathcal{S}_{A_2}$, $x_i = v_i \notin s$. Using classical negation (Gelfond and Lifschitz 1991), we ensure that an atom, $\texttt{k}$, which represents $x_i = v_i$, and its classically negated counterpart, $\texttt{-k}$, cannot both be true at the same time. We, thus, require that clingo return stable models that contain $\texttt{-k}$. 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 predicates that build on atoms in K also have their classically negated counterparts defined.

Branch and Bound Search

To reach a goal given by a goal specification, Π, from a given start state, s_0, CDGR uses a branch and bound (Land and Doig 1960) search to attempt to find a closest state in the set of states represented by Π. The upper bound represents the lowest cost path found to a goal state. In this search algorithm, a node represents an assignment and a conflict. The lower bound of a node is estimated using the cost of a path from s_0 to the assignment associated with that node [1]. CDGR is independent of the pathfinding algorithm used as long as the pathfinding algorithm returns a path, the terminal state, and the path cost, if a path is found.

A priority queue of nodes is maintained, where the priority of a node is given according to the size of its assignment and ties are broken according to the estimated 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, n_{pop}, is removed from the priority queue and specializations of its associated assignment, $n_{\text{pop}}.A$, with respect to its associated conflict are obtained. If $n_{\text{pop}}.A$ 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 specializations 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.

For each specialization, A, a pathfinding algorithm is used to find a path from s_0 to A. Since some of the specializations may be unreachable, we limit the number of iterations the pathfinding algorithm is given to find a path. If a path to A is found, the path cost is less than the upper bound, and the terminal state is not a goal state, then a conflict, A_c, is found based on the terminal state and the specification. A new node is then created with A and A_c, its lower bound is set to the path cost, and it is pushed to the priority queue. Otherwise, if the aforementioned condition is not met, we "ban" A, so that no specializations of A will be sampled from $\alpha(\Pi)$. This is similar to nogood recording in constraint satisfaction (Dechter 1986). We ban A because, if the aforementioned condition is not met, either a path to A is not found or a path to A is found via a path cost greater than or equal to the upper bound and/or the terminal state is a goal state. We ban A if a path is not found because, in practice, if a path is not found to A, then a path is usually not found to specializations of A. We ban A if a path is found to A, because, although DNNs are not guaranteed to find a shortest path, in practice, a path to A is at least as cheap as a path to specializations of A. Finally, after banning A, if a path is found to a goal state via a path cost less than the upper bound, the upper bound is set to the path cost and the best path is set to the path found. The algorithm is outlined in Algorithm 1. The priority queue assigns higher priority to lower values. The size of the initial assignment of None is

[1] Note that, since we use a DNN heuristic function, which is not guaranteed to be admissible, to find paths, this lower bound may be incorrect. However, there is ongoing research on admissibility and DNNs (Agostinelli et al. 2021; Li et al. 2022). Furthermore, we also use weighted A* search (Pohl 1970), which does not guarantee optimality.

Algorithm 1: Conflict-Driven Goal Reaching

Input: Specification Π, pathfinding algorithm path, 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
      $path_i, s_t, g$ = path(s_0, A)
      if ($path_i$ is not None) and ($g < ub$) and ($s_t \notin \alpha(\Pi)$) **then**
        $A_c \leftarrow$ conflict(s_t, Π)
        push NODE($A = A, A_c = A_c, lb = g, i = 0$) to q with priority $(-|A|, 0, g)$
      else
        ban A
        if ($path_i$ is not None) and ($g < ub$) and ($s_t \in \alpha(\Pi)$) **then**
          $ub \leftarrow g$; path $\leftarrow path_i$
          remove n from q and ban $n.A$ if $n.lb >= ub$
        end if
      end if
    end if
  end for
end while
return path

taken to be zero. The pathfinding algorithm returns a path, terminal state, and path cost, if a path is found. If a path is not found, then these values are None. A visualization of this algorithm is shown in Figure 1.

Properties of Conflict-Driven Goal Reaching

Since unreachable assignments can be sampled, the pathfinding algorithm used to find paths to assignments must limit its computation in order to not waste time searching for paths that do not exist. However, this limit could come prematurely in cases where a path does exist. Therefore, the pathfinding algorithm used by CDGR is not complete. By extension, CDGR is not complete and is also not optimal since, by failing to find a path, it can also fail to find an optimal path. We leave modifications to CDGR that obtain theoretical guarantees to future work.

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. To define classically negated atoms, for the Rubik's cube, we use one clause that says that a color cannot be at a given index if it holds that a different color is at that index. For the 24-puzzle, we use three clauses that say 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. For the Rubik's cube, the constraints encoded include that stickers are on cubelets (the smaller cubes that compose the Rubik's cube) and a cubelet cannot have multiple colors from the same face or opposite faces on it (i.e. a cubelet with two white stickers or a white and yellow sticker does not exist). For the 24-puzzle, the constraints encoded include that a tile cannot be in more than one location.

We randomly sample 100 start states for our test data. We test on goals expressed both with and without NAF. We label the goals as follows: <domain name>:Π^i_m or <domain name>:Π^i_n. Where Π^i_m and Π^i_n represent the same set of goal states for a given domain and Π^i_m is a monotonic specification (without NAF) and Π^i_n is a non-monotonic specification (with NAF). The goals we test are:

- 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.
- RC:Π^2_m: For all faces, all stickers on that face are different than its center sticker.
- RC:Π^2_n: There does not exist a face that has a sticker that matches its 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 sums of all rows are even.
- 24p:Π^2_n: There does not exist a row whose sum is odd.

The trained heuristic functions we use are the same ones trained in DeepCubeA$_g$ (Agostinelli, Panta, and Khandelwal 2024b) which were obtained from the corresponding public GitHub (Agostinelli, Panta, and Khandelwal 2024a). 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 set the number of specializations per iteration, B, to 10. We perform batch weighted A* search (Agostinelli et al. 2019) to find paths to assignments with a batch size of 100 and weight of 0.2 on the path cost for a maximum of 100 iterations. 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 patience parameter to 5 for all of our experiments. 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, 3, and 4.

The results in Table 1 show reaching goals with CDGR for goals expressed using NAF significantly decreases the path cost for both the random specialization operator and the conflict-driven specialization operator. In particular, for RC:Π^1_m, the average path cost is 11.5, it solves 70% of instances, and does so with an average of 564.9 seconds. However, for RC:Π^1_n with conflict-driven search, the average path cost is 1.3 (an 88% decrease), it solves 100% of instances, and does so with an average of 6.0 seconds (a 98% decrease). When NAF is used, the conflict-driven specialization operator is faster than the random specialization operator in all cases but one and finds shorter paths when compared to the random specialization operator in all cases. For RC:Π^2_m, no states were solved since clingo took between 1,000 and 3,000 seconds to sample a single assignment from the monotonic specification.

(a) Start (b) RC:Π^1_m (c) RC:Π^1_n w/ conf

Figure 2: The path cost is 12 without NAF and 1 with NAF and conflict-driven search.

(a) Start (b) RC:Π^2_n w/ conf

Figure 3: The goal could not be reached without NAF. The path cost is 16 with NAF and conflict-driven search.

12	22	6	9	5
7	1	19	2	17
16	13	4	20	21
11	15	10	3	8
	14	18	24	23

17	10	20	5	22
1	6	14	15	16
12	13	23		8
11	3	9	4	7
18	19	2	21	24

12	22	6	9	5
7	1	19	2	17
16	13	4	20	21
11	15	10		8
14	18	24	3	23

(a) Start (b) 24p:Π^2_m (c) 24p:Π^2_n w/ conf

Figure 4: The path cost is 93 without NAF and 4 with NAF and conflict-driven search.

Discussion

The success of CDGR can be seen in how much time it took to sample an assignment with the specialization operator, how often goals represented by sampled assignments were reached, and the amount of time it took to find a path, as shown in Table 1. The monotonic specifications took much more time sampling assignments than the non-monotonic specifications. For example, RC:Π^1_m takes an average of 12.8 seconds while RC:Π^1_n took an average of 0.1 seconds. This shows that it can be computationally convenient to specify when the goal is not true using NAF instead of specifying when the goal is true. This computational burden is also demonstrated in the Rubik's cube example in the Introduction.

The sampled assignments are harder to reach for monotonic specifications. This is because, unless all relevant domain-specific constraints relevant to reachability are encoded in the background knowledge, which may not be known to the practitioner, sampled assignments can be unreachable. Furthermore, encoding all relevant constraints may also lead to longer computation times and, thus, slower assignment sampling times. Since the monotonic specifications require more variables to be assigned, the likelihood that it will represent an impossible assignment increases. In the extreme case for RC:Π^2_m and 24p:Π^2_m, all variables must be assigned whereas, for the non-monotonic specification, in the most general case, the empty assignment makes the specification true. This represents the set of all states, many of which are not goal states, while, as shown by Theorem 1, all candidate states are goal states for monotonic specifications. However, the results show that learning conflicts from non-goal states to sample specializations that prevent prior conflicts using CDGR results in superior performance.

The time it takes to find a path to a goal represented by a sampled assignment is much faster for non-monotonic specifications than monotonic ones. This is because non-monotonic specifications can start from very general assignments, such as the empty assignment, add variable assignments to it based on conflicts, and prioritize these resulting assignments based on path cost. On the other hand, monotonic specifications have to sample a much more specific assignment, immediately, and this sampling process is path cost agnostic. As shown in Theorem 2, many assignments may have to be sampled before sampling a closest assignment. This problem is exacerbated by the fact that sampling assignments takes much longer for monotonic specifications.

Future Work

For RC:Π^2_n, the percentage of states solved is 46%, whereas the percentage of states solved for the other non-monotonic specifications is 100% with conflict-driven search. The fact that the average number of iterations for successful searches is up to five times higher than that of other specifications and that the average percentage of reached assignments that did not result in a goal state is up to four times higher than that of other specifications indicates that many specializations were necessary before a goal state was found. Given

Goal	Op	Cost	Solve	Itr	Node	Reach	¬Goal	$\frac{\text{Secs}}{\text{Assgn}}$	$\frac{\text{Secs}}{\text{Path}}$	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**
RC:Π^2_m	-	-	-	-	-	-	-	-	-	-
RC:Π^2_n	Rand	5.3	3	13.7	127.7	80.7	98.4	0.0	2.1	279.8
	Conf	13.7	**46**	22.1	179.7	91.5	98.0	0.1	1.0	227.8
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**	**4.0**	**30.8**	**100.0**	38.7	**0.2**	**0.0**	**6.8**
	Conf	**2.2**	**100**	4.1	32.2	**100.0**	21.5	**0.2**	0.1	7.8
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	16.4	99	10.1	91.5	100.0	85.1	0.1	0.1	26.4
	Conf	**13.5**	**100**	**8.7**	**77.9**	**100.0**	78.1	**0.1**	**0.1**	**21.9**

Table 1: Comparison of monotonic and non-monotonic specifications with random and conflict-driven specialization operators when performing CDGR 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, the average percentage of reached assignments where the terminal state was not a goal state, the average number of seconds it took to sample an assignment with the specialization operator, 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. The averages were computed amongst the instances that were solved. Since the time limit was checked after each iteration of branch-and-bound search, some completed searches may go over the time limit.

that the majority of computation time was taken by finding paths, future work could remedy this by training a network to predict when finding a path is likely to result in finding a goal state, so that pathfinding is performed less during the conflict-driven search. Given that a lower than usual average percentage of sampled assignments were reached, this could be combined with training the heuristic function on random assignments, some of which may be unreachable. The heuristic function should then give a very large cost-to-go for these unreachable assignments, which can then be pruned.

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 CDGR to create a dataset to train a heuristic function that directly encodes the first-order logic specification. This could then be used to better prioritize nodes and even remove the need to sample assignments, altogether.

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 (McDermott 2000). 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; Speck et al. 2019) and domain independent heuristics that take advantage of axioms have been derived (Ivankovic and Haslum 2015). Furthermore, existential quantification can play a significant 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).

Conclusion

NAF allows us to express goals succinctly and has the ability to significantly reduce computational load. However, reaching goals specified with NAF must address non-monotonicity. We address this problem with CDGR, which exploits NAF semantics to quickly find conflicts and specialize assignments based on those conflicts. This conflict-driven approach often results in finding shorter paths in less time when compared to random specializations and also when compared to specifying equivalent goals without NAF.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Award No. 2426622.

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.; McAleer, S.; Shmakov, A.; Fox, R.; Valtorta, M.; Srivastava, B.; and Baldi, P. 2021. Obtaining Approximately Admissible Heuristic Functions through Deep Reinforcement Learning and A* Search. In *International Con-*

ference on Automated Planning and Scheduling - Bridging the Gap Between AI Planning and Reinforcement Learning Workshop.

Agostinelli, F.; Panta, R.; and Khandelwal, V. 2024a. SpecGoal. https://github.com/forestagostinelli/SpecGoal.

Agostinelli, F.; Panta, R.; and Khandelwal, V. 2024b. 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_2^p-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. $\exists$-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.

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

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.

Li, T.; Chen, R.; Mavrin, B.; Sturtevant, N. R.; Nadav, D.; and Felner, A. 2022. Optimal search with neural networks: Challenges and approaches. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 109–117.

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.

Muppasani, B.; Pallagani, V.; Srivastava, B.; and Agostinelli, F. 2024. Comparing Rubik's Cube Solvability in Domain-Independent Planners Using Standard Planning Representations for Insights and Synergy with Upcoming Learning Methods. In *International Conference on Automated Planning and Scheduling - Heuristics and Search for Domain-Independent Planning Workshop*.

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

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.