Solving Goal Recognition Design using ASP

Tran Cao Son† and Orkunt Sabuncu‡ and Christian Schulz-Hanke†
and Torsten Schaub‡‡ and William Yeoh†
†New York State University, Las Cruces, NM, USA
‡University of Potsdam, Germany ‡‡INRIA, Rennes, France

Abstract

Goal Recognition Design involves identifying the best ways to modify an underlying environment that agents operate in, typically by making a subset of feasible actions infeasible, so that agents are forced to reveal their goals as early as possible. Thus far, existing work has focused exclusively on imperative classical planning. In this paper, we address the same problem with a different paradigm, namely, declarative approaches based on Answer Set Programming (ASP). Our experimental results show that one of our ASP encodings is more scalable and is significantly faster by up to three orders of magnitude than the current state of the art.

Introduction

Goal recognition, a special form of plan recognition, deals with online problems aiming at identifying the goal of an agent as quickly as possible given its behavior (Geffner and Bonet 2013; Ramírez and Geffner 2011). Goal recognition is relevant in many applications including security (Jarvis, Lunt, and Myers 2005), computer games (Kabanza et al. 2010), and natural language processing (Geib and Steedman 2007). For example, Fig. 1(a) shows an example gridworld application, where the agent starts at cell E3 and can move in any of the four cardinal directions. Its goal is one of three possible ones G1, G2, and G3. The traditional approach has been to find efficient algorithms that observe the trajectory of the agent and predict its actual goal (Geffner and Bonet 2013; Ramírez and Geffner 2011).

Keren, Gal, and Karpas (2014) took an orthogonal approach by proposing to modify the underlying environment in which the agents operate, typically by making a subset of feasible actions infeasible, so that agents are forced to reveal their goals as early as possible. For example, under the assumption that agents follow optimal plans to reach their goal, by making the action that moves the agent from cells E3 to D3 infeasible, the agent is forced to either move left to E2, which would immediately reveal that its goal is G1, or move right to E4, revealing that it is either G2 or G3. They call this the Goal Recognition Design (GRD) problem. It is relevant in many of the same applications that goal recogni-

Figure 1: Example Problem

tion problems are relevant because, typically, the underlying environment can be easily modified.

In Keren, Gal, and Karpas (2014)’s seminal paper, they introduced the notion of worst-case distinctiveness (wcd), as a goodness measure that assesses the ease of performing goal recognition within an environment. The wcd of a problem is the longest sequence of actions an agent can take without revealing its goal. The objective in GRD is then to find a subset of feasible actions to make infeasible such that the resulting wcd is minimized.

Existing algorithms have been designed and developed exclusively using imperative programming techniques, where the algorithms define a control flow, that is, a sequence of commands to be executed. The focus has been on using state-of-the-art planners and pruning techniques in computing and reducing the wcd. In this paper, we are interested in investigating the benefits of using declarative programming techniques to solve GRD problems. Specifically, we propose to use Answer Set Programming (ASP) (Niemelä 1999; Marek and Truszczyński 1999) as the general framework for solving GRD problems.

This paper contributes to both areas of GRD and ASP. Regarding GRD, we demonstrate that using ASP as the general framework provides a number of benefits including the ability to capitalize on (i) ASP’s manifold problem solving techniques (e.g., formulating GRD as a minimization/maximization problem that can be solved by saturation techniques) and (ii) the highly optimized and effective ASP solvers, which results in improved scalability. As regards ASP, the paper shows—by solving GRD—that an appropriate use of the saturation-based encoding coupled with a phase separation can be extremely effective in solving QBF problems. Specifically, it demonstrates that many problem solving techniques developed in ASP can be used directly to outperform specialized imperative method in problems-of-
interest in other communities. Therefore, in this paper, we make a first step of bridging the two areas of GRD and ASP.

**Background**

**Classical Planning.** A classical planning problem (Geffner and Bonet 2013), often formulated in STRIPS (Fikes and Nilsson 1971), is a tuple \( \langle F, s_0, A, C, G \rangle \), where \( F \) is a set of fluents; \( s_0 \) is the start state of the agent; \( A \) is the set of actions; \( C : A \rightarrow \mathbb{R} \) defines the cost for each action; and \( G \) is a set of goal states. Each action \( a \in A = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle \) is a triplet consisting of the precondition, add, and delete lists, respectively, and all are subsets of \( F \). An action \( a \) is applicable in state \( s \) if \( \text{pre}(a) \subseteq s \). If action \( a \) is applied in state \( s \), then it results in a new state \( s' = (s \setminus \text{del}(a)) \cup \text{add}(a) \). A plan \( \pi = \langle a_1, \ldots, a_n \rangle \) is a sequence of actions that brings an agent from the starting state \( s_0 \) to a goal state \( g \in G \). The cost of a plan \( C(\pi) = \sum_i C(a_i) \) is the sum of the cost of each individual action in the plan. The goal is typically to find a cost-minimal plan \( \pi^* = \arg\min_{\pi} C(\pi) \).

**Goal Recognition Design (GRD).** A GRD problem (Keren, Gal, and Karpas 2014) is represented as a tuple \( \mathcal{P} = \langle D, G \rangle \), where \( D = \langle F, s_0, A, C \rangle \) captures the domain information and \( G \) is a set of possible goal states of the agent. The elements in \( D \) are as in classical planning except that every action’s cost is 1.

**Definition 1** Given a GRD problem \( \mathcal{P} \). The worst case distinctiveness (wcd) of a problem \( \mathcal{P} \) is the length of a longest sequence of actions \( \pi = \langle a_1, \ldots, a_n \rangle \) that is the prefix in cost-minimal plans \( \pi_{g_1}^* \) and \( \pi_{g_2}^* \) to distinct goals \( g_1, g_2 \in G \).

Using our example problem of Figure 1(a), a longest sequence of actions that can lead to two distinct goals is \( \langle (E3, \text{up}), (D3, \text{up}), (C3, \text{right}), (C4, \text{right}) \rangle \), where we use pairs \( (s, a) \) to denote that action \( a \) is taken at cell \( s \). This sequence of actions can lead to either goals \( G2 \) or \( G3 \) and is of length 4. Thus, the wcd of the problem is 4.

For a GRD problem \( \mathcal{P} \) and a set of actions \( X \subseteq A \), let \( \mathcal{P} \otimes X \) be the problem \( \mathcal{P} = \langle D, G \rangle \) where \( D = \langle F, s_0, A \setminus X, C \rangle \). The objective in GRD problems is to find a subset of actions \( A^* \subseteq A \) such that if they are removed from the set of actions \( A \), then the wcd of the resulting problem \( \mathcal{P} \) is minimal. This optimization problem is subject to the requirement that the cost of cost-minimal plans to achieve each goal \( g \in G \) is the same before and after removing the subset of actions.

In this paper, we investigate a variant of GRD problems, where we limit the maximum number of actions to remove to a user-defined parameter \( k \). Thus, this variant is equivalent to the original problem when \( k = \infty \).

**Definition 2** Given a GRD problem \( \mathcal{P} \) and an integer \( k \). The \( k \)-reduced GRD problem over \( \mathcal{P} \) is to find a set of actions \( A^* \), called a solution to \( \mathcal{P} \) w.r.t. \( k \), such that

\[
A^* = \arg\min_{\mathcal{A} \subseteq A} \text{wcd}(\mathcal{P} \otimes \mathcal{A}) \quad \text{subject to} \quad \mathcal{A} \subset A, \\
C(\pi_g^*) = C(\hat{\pi}_g^*) \quad \forall g \in G, \\
|A^*| \leq k
\]

where \( \pi_g^* \) is a cost-minimal plan to achieve goal \( g \) in the original problem \( \mathcal{P} \), and \( \hat{\pi}_g^* \) is a cost-minimal plan to achieve goal \( g \) in problem \( \mathcal{P} \otimes \hat{A}^* \).

In our example problem, if \( k = 3 \) then blocking the actions \( (E3, \text{up}), (C4, \text{right}), (C5, \text{up}) \), where \( (s, a) \) denotes that action \( a \) is blocked at cell \( s \), reduces the wcd of the problem to 2. Fig. 1(b) shows the actions blocked. Given that, there are the following cases: (a) if the agent executes the action \( (E3, \text{left}) \), we know that the agent’s goal is \( G1 \); (b) if the agent executes the action \( (E3, \text{right}) \), then the goal is \( G2 \) or \( G3 \); (c) if the agent continues with the action \( (E4, \text{right}) \), its goal must be \( G3 \); (d) after the agent executes the sequence \( ((E3, \text{right}), (E4, \text{up})) \), it must reveal its goal by either \( (D4, \text{right}) \) or \( (D4, \text{up}) \). This implies that the longest sequence of actions that can be executed by the agent before it must reveal its goal is \( ((E3, \text{right}), (E4, \text{up})) \), i.e., the wcd of the resulting problem is 2.

To the best of our knowledge, the only algorithms to compute or reduce the wcd of a problem are the ones introduced by Keren, Gal, and Karpas (2014). They introduced two algorithms, \text{WCD-BFS} and \text{LATEST-SPLIT}, to compute the wcd of a problem. \text{WCD-BFS} uses breadth-first search (BFS) to explore all combinations of paths and prunes subsets of paths that are provably distinctive. \text{LATEST-SPLIT} compiles the problem into a set of classical planning problems and solves them using any classical planner.

They also introduced two algorithms, \text{EXHAUSTIVE-REDUCE} and \text{PRUNED-REDUCE}, to reduce the wcd of a problem. \text{EXHAUSTIVE-REDUCE} uses a variation of BFS to exhaustively search the whole search space in the worst case. \text{PRUNED-REDUCE} optimizes \text{EXHAUSTIVE-REDUCE} by pruning some portions of the search space.

**ASP and Multi-shot ASP.** A logic program II (Gelfond and Lifschitz 1990) is a set of rules of the form

\[
c_1 | \ldots | c_k \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \tag{1}
\]

where \( 0 \leq m \leq n \), \( 0 \leq k \), each \( a_i \) or \( c_j \) is a literal of a propositional language and \( \text{not } a \) represents a default negated literal (or naf-literal) where \( a \) is a literal. When \( k = 0 \) (\( n = 0 \)), (1) is called a constraint (fact). Semantically, a program II induces a collection of so-called answer sets, which are distinguished models of \( \Pi \) determined by answer set semantics; see (Gelfond and Lifschitz 1990) for details.

To facilitate the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints (Simons, Niemelä, and Soiininen 2002). The former are of form \( a : b_1, \ldots, b_m \), the latter can be written as \( s \{ c_1, \ldots, c_n \} \) \( t \) where \( a \) and \( b_i \) are possibly default negated literals, and each \( c_j \) is a conditional literal; \( s \) and \( t \) provide a lower and upper bound on the number of satisfied literals within the cardinality constraints. The practical value of both constructs becomes more apparent when used in conjunction with variables. For instance, a conditional literal of form \( a(X) : b(X) \) in a rule body expands to the conjunction of all instances of \( a(X) \) for which the corresponding instance of \( b(X) \) holds. Similarly,
2 \{a(X) : b(X)\} is true, whenever more than one and less than five instances of \(a(X)\) (subject to \(b(X)\)) are true. Similarly, objective functions minimizing the sum of weights \(w_j\) of conditional literals are expressed as \#\text{minimize} \{w_1 : c_1, \ldots, w_n : c_n\}.

Traditional ASP rests upon a single-shot approach to problem solving, i.e., an ASP solver takes a logic program, computes its answer sets, and exits. Unlike this, recently developed multi-shot ASP solvers provide operative solving processes for dealing with continuously changing logic programs. Such changes can be brought about by unfolding a transition function, sensor data, or more elaborate external data. For controlling such solving processes, the declarative approach of ASP is combined with imperative means. In clingo (Gebser et al. 2014), this is done by augmenting an ASP encoding with Python procedures controlling ASP solving processes along with the corresponding evolving logic programs. The instrumentation includes methods for adding/grounding rules, setting truth values of (external) atoms, computing the answer sets of current program, etc.

**Solution Approaches**

We investigate alternative encodings of the GRD problem in ASP. The first encoding utilizes meta-programming and saturation techniques and the second one employs a hybrid implementation made possible by multi-shot ASP. In what follows, let \(P = \{D, G\}\) where \(D = \{F, s_0, A, C\}\) be a GRD problem and \(k\) be a positive integer.

**A Saturation-based Meta Encoding.** We employ the method of encoding a planning problem by a set of facts to encode \(D\) and \(G\). Specifically, (i) \(F\) is encoded by a set of atoms of the form \(\text{fluent}(f)\) for \(f \in F\); (ii) \(A\) by \(\text{action}(a)\) for \(a \in A\); (iii) \(\text{init}(l)\) for \(l \in s_0\); (iv) each \(g \in G\) by \(\text{goal}(g, l)\) for \(l\) as a conjunct in \(g\); (v) \(\text{pre}(a)\) by a set of atoms of the form \(\text{exec}(a, l)\); (vi) \(\text{add}(a)\) (\(\text{del}(a)\)) by \(\text{effect}(a, f, i)\) (\(\text{effect}(a, -f, i)\)) where \(i\) is a unique identifier associated with an effect of \(a\); (vii) each condition of a conditional effect by \(\text{cond}(a, l, i)\) for \(l\) as a conjunct in condition for the effect of action \(a\) associated with \(i\). We note that our proposed encoding can deal with planning problems in extended STRIPS syntax or action languages.

The saturation technique is an advanced guess and check methodology used in disjunctive ASP to check whether all possible guesses in a problem domain satisfy a certain property (Eiter, Ianni, and Krennwallner 2009). It can be used to encode \(\Sigma^\nu_3\)-complete problems. A typical problem in this class is the satisfiability problem for \(\exists\forall\text{-QBF}\). For instance, in a typical encoding for satisfiability of a \(\exists\forall\text{-QBF}\) the guess part uses disjunction to generate all possible truth values for the propositional atoms that are quantified by \(\forall\) (\(\exists\)-atoms) and the check part checks the satisfiability of the formula for all valuations of the \(\forall\)-atoms (i.e., it checks whether the resulting formula after applying choices made for \(\exists\)-atoms is a tautology or not). To achieve this, the fact that answer sets are minimal w.r.t. the atoms defined by disjunctive rules is utilized. To this end, the saturation part of the program derives (saturates) all atoms defined in the guess part for generating the search space. However, the saturation technique puts syntactical restrictions on the program parts by forbidding the use of saturated atoms as naf-literals in a rule or as positive literals in an integrity constraint (Eiter, Ianni, and Krennwallner 2009; Leone, Rosati, andScarcello 2001).

Let \(vI(x,y,c)\) denote that \(c\) is the common prefix of minimal cost plans of \(\pi_x^+\) and \(\pi^+_y\). It is easy to see that the following \(\exists\forall\text{-QBF}\) encodes the \(\text{wcd}\) definition

\[
\exists x, y, c \forall vI(x,y,c) \land \forall x, y', c' \forall vI(x', y', c') \land \exists \forall I(x, y, c) \geq \exists \forall I(x, y, c')
\]  

(2)

where, for the sake of simplicity, we omit some details such as \(x, y, x', y' \in G\), and that \(c\) and \(c'\) correspond to sequences of actions that are the common prefix of cost-optimal plans \(\pi_x^+\) and \(\pi_y^+\), respectively.

To compute the \(\text{wcd}\), we only need to encode the satisfiability of formula (2). As in any ASP based encoding for planning, we assume a finite horizon \(len\) and use \(st(l)\) to represent time steps from 1 to \(len\). For each fluent \(f \in F\), \(\text{comp}(f, l, -f)\) encodes its complementary literals. For convenience all fluent literals are encoded by \(\text{lit}(f)\) and \(\text{lit}(-f)\) for \(f \in F\). The rules for the \(\text{guess}\) and \(\text{check}\) of formula (2) are described next. In these rules, \(o(A, P, T)\) denotes that action \(A\) occurs at step \(T\) in order to achieve goal \(P\).

**Guess.** The group of rules (6)–(14) choose valid values for \(\exists\)-atoms in (2) while the rules in (3)–(5) block at most \(k\) actions and specify actions (via \(\text{pacts}(s)\)) that could be used in creating plans. (6)–(8) correspond to the choice of \(x \neq y\) in formula (2). (9) chooses among only unblocked actions (via \(\text{pacts}(s)\)) to create action occurrences. (10)–(11) guarantee that the chosen actions satisfy the conditions of goals \(x\) and \(y\). \(h(P,F,T)\) denotes that fluent \(F\) holds at step \(T\) while trying to achieve goal \(P\). Thus, the conditional literals in rule (10) state that all fluents occurring as a positive literal in a corresponding goal must hold \(h(P,F,T) : \text{goal}(G, Pos), \text{comp}(F, Pos, Neg)\) and similarly all fluents occurring as a negative literal must not hold \((\text{not} h(P,F,T)) : \text{goal}(G, Neg), \text{comp}(F, Pos, Neg)\). (12)–(14) compute the common prefix \(c\) using the \(\text{same}/1\) and \(\text{follow}/1\) predicates. \(\text{same}(T)\) states the same action occurs at step \(T\) concerning the goals \(x\) and \(y\) and instances of \(\text{follow}(1) \ldots \text{follow}(n)\) in an answer set \(s.t. n \leq \text{len}\) represent the path followed by the agent from start to step \(n\) is a non-distinguishable path.
In the guess part, we generate a search space for the \( \forall \)-atoms. (15) guesses goals \( x' \) and \( y' \) using disjunction as mentioned in the saturation technique. The usage of as(\( P, G \)) is similar to that of s(\( P, G \)) (1p and 2p as values of \( P \) refer to \( x' \) and \( y' \), respectively, and defined by facts \( \text{pos}(1p) \) and \( \text{pos}(2p) \)). nas(\( P, G \)) represents that \( G \) is not selected for \( P \) (\( x' \) or \( y' \)). (16) guesses \( ao/3 \) to be used for the computation of \( c' \) as in the case of \( o/3 \).

\[
\text{as}(P,G)\text{nas}(P,G) \leftarrow \text{pos}(P), \text{goal}(G,\_). \quad (15)
\]

\[
\text{ao}(A,P,T)\text{nas}(A,P,T) \leftarrow \text{pos}(P), \text{pacts}(A), \text{st}(T). \quad (16)
\]

**Check.** The conditional in formula (2) states that either the guessed values of \( x', y' \) and \( c' \) are invalid or the length of the common prefix \( c \) is greater than or equal to that of \( c' \). The predicates invalid and goal in the following check part of the program represent the former and the latter condition, respectively. For instance, the guess is invalid if for each \( x' \) and \( y' \) more than one goal is selected (rule (17)), no goal is selected (18), or they refer to the same goal (19). Note that we are constrained in the use of naf-literals for atoms defined in the guess part or ones dependant on them. Rule (18), for instance, uses a conditional literal instead of not as(\( P, _ \)) to represent that it is invalid to select no goal for position \( P \). Regarding \( c' \), (20) states that it is invalid if more than one action occurs at any step and for any goal. Rules (21)–(22) represent that if some goal condition of \( x' \) and \( y' \) does not hold at the horizon (len), then the selected actions for \( c' \) do not achieve the related goal and the guess is invalid.

\[
\begin{align*}
\text{invalid} & \leftarrow 2\{\text{as}(P,G)\}, \text{pos}(P). \quad (17) \\
\text{invalid} & \leftarrow \text{pos}(P), \text{nas}(P,G) \leftarrow \text{goal}(G,\_). \quad (18) \\
\text{invalid} & \leftarrow \text{as}(1p,G), \text{as}(2p,G). \quad (19) \\
\text{invalid} & \leftarrow 2\{\text{ao}(A,P,T)\}, \text{pos}(P), \text{st}(T). \quad (20) \\
\text{invalid} & \leftarrow \text{as}(P,G), \text{goal}(G,P). \quad (21) \\
\text{comp}(F,\text{Pos},\text{Neg}), \text{nh}(P,F,\text{len}). & \quad \text{invalid} \leftarrow \text{as}(P,G), \text{goal}(G,\text{Neg}). \quad \text{comp}(F,\text{Pos},\text{Neg}), \text{h}(P,F,\text{len}). \quad (22)
\end{align*}
\]

Using meta-programming techniques, similar to (Gebser, Kaminski, and Schaub 2011), we encode the action domain, in which we obey syntactic restrictions of saturation. Rules (23)–(26) represent the initial state for the planning problem. Related to the guesses \( x', y' \), and \( c' \), the rules (25)–(26) explicitly define conditions when a fluent does not hold (by \( \text{nh}/3 \)) since we cannot use naf-literals such as not \( h(P,F,T) \) in this context.

\[
\begin{align*}
\text{h}(P,F,0) & \leftarrow \text{s}(P,\_), \text{init}(\text{Pos}), \text{comp}(F,\text{Pos},\text{Neg}). \quad (23) \\
\text{h}(P,F,0) & \leftarrow \text{pos}(P), \text{init}(\text{Pos}), \text{comp}(F,\text{Pos},\text{Neg}). \quad (24) \\
\text{nh}(P,F,0) & \leftarrow \text{pos}(P), \text{init}(\text{Neg}), \text{comp}(F,\text{Pos},\text{Neg}). \quad (25) \\
\text{nh}(P,F,0) & \leftarrow \text{pos}(P), \text{not init}(\text{Pos}), \text{not init}(\text{Neg}), \text{comp}(F,\text{Pos},\text{Neg}). \quad (26)
\end{align*}
\]

Next we represent the action preconditions by rules (27)–(31). While representing the constraint that no action with an unsatisfied precondition can occur in \( c \) (rules (27)–(28)), we can use integrity constraint and naf-literal (since only \( \exists \)-variables play a role). However, we have to use \( \text{nh}/3 \) to represent non-executability of an action in \( c' \) (rules (29) and (30)) and generate invalid to eliminate such situations in (31). The action effects are represented by rules (32)–(35). In these rules, \( c(P,L,T) \) denotes that literal \( L \) is caused at step \( T \) for achieving a goal depending on the value of \( P \) (\( x \) and \( y \) are handled by rule (32), \( x' \) and \( y' \) by rule (33)). Similar to the use of conditional literals in rule (10), the ones in rules (32) and (33) check whether all conditions of an effect hold in case it is a conditional effect when deriving a \( c/3 \) atom. Rules (34) and (35) define the positive and negative effects.

\[
\begin{align*}
\text{\text{\textless; (27)}} & \text{\textless; (27)} \\
\text{\textless; (28)} & \text{\textless; (28)} \\
\text{\textless; (29)} & \text{\textless; (29)} \\
\text{\textless; (30)} & \text{\textless; (30)} \\
\text{\textless; (31)} & \text{\textless; (31)} \\
\text{\textless; (32)} & \text{\textless; (32)} \\
\text{\textless; (33)} & \text{\textless; (33)} \\
\text{\textless; (34)} & \text{\textless; (34)} \\
\text{\textless; (35)} & \text{\textless; (35)}
\end{align*}
\]

The law of inertia can be neatly represented in ASP by the use of negation-as-failure. Rule (40) represents it for the fluents related to the \( \exists \)-part of the problem. It basically states that a fluent keeps holding unless its complement is caused (not \( c(P,Neg,T) \)). Inertia for fluents related to the \( \forall \)-part of the problem, however, needs a more labourious representation due to restrictions of the saturation technique. Rules (41) and (42) represent it for positive and negative cases (a fluent holds \( h/3 \) and does not hold \( nh/3 \) using the \( nc/3 \) predicate, which is the dual of \( c/3 \) and states that a fluent literal is not caused at a time point. There are two cases to derive a \( nc \) atom w.r.t. a fluent literal and a time point. One case is that at this time point there are no occurrences of actions having this fluent literal as an effect (rule (36)). The other case is that an action having this literal as a conditional effect occurs at this time point but a condition of the effect fails to hold. (cf. (37)–(39)).
We have already encoded conditions for the predicate \( invalid \). Now, we define the condition \( |c| \geq |c'| \) with the \( goal \) predicate. Rule (43) defines \( nafollow(T) \) that represents the occurrence of different actions at step \( T \) concerning the goals \( x' \) and \( y' \). Rule (44) propagates \( nafollow \) till the horizon \( len \) once the path becomes distinctive. The predicate \( geq(T) \) states that \( |c_r| \geq |c'_r| \) holds where \( c_r \) or \( c'_r \) denotes the common prefixes considering only the paths formed by selected actions from start to step \( T \) for selected goals \( x, y, x', y' \), respectively. This is achieved by propagating \( geq(T - 1) \) to the next step whenever the path concerning \( x' \) and \( y' \) is already distinctive (\( nafollow(T) \) holds, rule (46)) or the path concerning \( x \) and \( y \) is still non-distinctive (\( follow(T) \) holds, rule (47)). Hence, \( |c| \geq |c'| \) holds when \( geq(len) \) holds (rule (48)).

\[
\begin{align*}
nafollow(T) & \leftarrow ao(A, P, T), naf(A, P, T), P \neq P1. \quad (43) \\
nafollow(T + 1) & \leftarrow nafollow(T), len > T. \quad (44) \\
geq(0) & \leftarrow . \\
geq(T) & \leftarrow geq(T - 1), nafollow(T). \quad (46) \\
geq(T) & \leftarrow geq(T - 1), follow(T). \quad (47) \\
goal & \leftarrow geq(len). \quad (48)
\end{align*}
\]

The satisfiability of the QBF in (2) is represented by rules (49) and (50). With constraint (51), we force \( sat \) to be in every answer set, i.e., the QBF should be satisfied. Additionally, the saturation part (52)–(55) forces all atoms defined in the guess part to be true in the case \( sat \) is true.

\[
\begin{align*}
sat & \leftarrow invalid. \quad (49) \\
sat & \leftarrow goal. \quad (50) \\
& \leftarrow not sat. \quad (51) \\
as(P, G) & \leftarrow sat, pos(P), goal(G, \_). \quad (52) \\
nas(P, G) & \leftarrow sat, pos(P), goal(G, \_). \quad (53) \\
ao(A, P, T) & \leftarrow sat, pos(P), pacts(A), st(T). \quad (54) \\
nao(A, P, T) & \leftarrow sat, pos(P), pacts(A), st(T). \quad (55)
\end{align*}
\]

In an answer set, instances of \( follow \) give us the \( wcd \) value of the problem for the selected blocked actions using the saturation technique. We can use optimization statements of \( clingo \) to minimize the \( wcd \) to solve the GRD problem (56).

\[
\#\text{minimize}\{1@2, T : follow(T)\}. \quad (56)
\]

Note that by canceling out rule (4) and adding a second level optimization statement \( \#\text{minimize}\{1@1, A : blocked(A)\} \) to minimize the number of blocked actions, we can solve a more general version of the problem that does not consider the fixed parameter \( k \).

Up to now, we have not mentioned how we represent the common prefixes \( e \) and \( e' \) consider only cost-optimal plans of the selected goals and the condition \( C(x') = C(x') \) for each goal \( y \in G \), i.e., preserving optimal costs of achieving goals. This can also be encoded using the saturation technique in the same way we calculate the \( wcd \) (we omitted this part from the encoding due to space constraints). First, we choose a plan \( \pi \) using only unblocked actions for each goal as an abstract \( \exists \)-atom. In the guess part, we guess another plan \( \pi' \) considering all actions as an abstract \( \forall \)-atom. The check part checks the condition \( |\pi| \leq |\pi'| \) for valid guesses to guarantee that \( \pi \) is an optimal plan. Additionally, since \( \pi' \) considers all actions (blocked or unblocked), we can be sure that optimal costs of achieving goals do not change after blocking. Moreover, the computation in this part can be independently solved as an initial phase for improving the runtime performance. The 2-phase encoding used in the experiments actually does this by first running a basic planning encoding to find actions used in all cost-optimal plans for each goal using the capacity of \( clingo \) for computing brave consequences of a logic program efficiently by linear number of calls to the solver (Gebser et al. 2007).

Let \( \Pi_E \) be the whole saturation based meta encoding including rules (3)–(56).

**Proposition 1** Given a GRD problem \( \mathcal{P} \) and an integer \( k \) where \( \mathcal{P}_w \) is the instance file for \( \mathcal{P} \). Assuming \( lim \) limits the maximal length of plans to all the goals in \( \mathcal{P} \). \( \mathcal{P} \) has a solution of the set \( A \subseteq A \) of blocked actions s.t. \( |A| \leq k \) and \( w = wcd(P \ominus A) \) is minimized iff \( \Pi = \Pi_E \cup \Pi_P \) has an answer set \( M \) s.t. \( A = \{ a \mid \text{blocked}(a) \in M \} \) and \( w = \{ \text{follow}(i) \mid 1 \leq i \leq t, \text{follow}(i) \in M \} \).

It is worthwhile to note that the saturation based encoding of the GRD problem actually follows a general methodology of solving minmax/maxmin optimization problems using disjunctive ASP. The GRD problem is a minmax optimization problem since the \( wcd \) represents the maximum non-distinctive path and the outer optimization minimizes it by blocking actions. As a methodology, we solved the inner optimization using saturation technique and let the branch-and-bound style optimization of the solver \( clingo \) handle the outer optimization. To the best of our knowledge, this is the first comprehensive ASP encoding solving minmax/maxmin optimization problems, which appear in interesting application areas. The elaborated research on this methodology and applying it to other areas are among our future work.

**A Multi-Shot ASP Encoding.** Let \( \mathcal{P} = \langle D, G \rangle \) be a GRD problem, where \( D = \{ F, s_0, A, C \} \), and \( k \) be a positive integer denoting the maximal number of actions that can be blocked for reducing the \( wcd \) of \( \mathcal{P} \). Furthermore, let \( max \) be an integer that denotes the maximal length of plans in \( \mathcal{P} \). We present a multi-shot ASP program \( \Pi(\mathcal{P}) \) for computing (i) \( wcd(\mathcal{P}) \); and (ii) a solution of \( \mathcal{P} \) wrt. \( k \). Specifically, \( \Pi(\mathcal{P}) \) implements Alg. 1 in multi-shot ASP and consists of a logic program \( \pi(\mathcal{P}) \) and a Python program \( \text{GRD(D, G, k, max)} \) (or \( \text{GRD(.)} \)).

Lines 3–6 of Alg. 1 compute the optimal cost plan for each \( g \in G \). If some goal has no plan of length at most \( max \) then \( \text{GRD(.)} \) returns \textit{unsolvable}. Otherwise, the optimal cost is used as the bound for \( \pi(\mathcal{P}) \) (Line 8). \( wcd(\mathcal{P}) \) is computed by computing an answer set of \( \pi(\mathcal{P}) \) with \( len \) equal to the maximal cost of all goals with the optimization module (Lines 7–11). Line 12 identifies the set of actions that can potentially change the \( wcd \) of the problem. Lines 14–17 implement a simple exhaustive search to identify a set of at most \( k \) actions that reduce \( wcd(\mathcal{P}) \). \( \text{GRD(.)} \) controls the computation of \( wcd(\mathcal{P}) \) and a solution of \( \mathcal{P} \) wrt. a given \( k \) assuming that the maximal length of plans to all goals in \( G \) is at most \( max \) by (i) setting the bound of plan cost (\( max \), Line 3–6), (ii) setting the parameter \( len \) of \( \pi(\mathcal{P}) \) (Line 8), and (iii) adding the optimization or blocking action
Algorithm 1 \( GRD((D, G), k, \max) \)

1: **Input:** a GRD problem \( P=(D, G) \) & integers \( k, \max \).
2: **Output:** \( wcd(P) \), and a solution \( R \) of \( P \) w.r.t. \( k \) and \( wcd(P \oplus R) \) or unsolvable if some goal is not achievable.
3: **for** each goal \( g \) in \( G \) **do**
4:  compute the minimal length of plan for \( g \)
5:  if plan of length \( i \leq \max \) exists **then** set \( m_g = i \)
6:  **else return** unsolvable
7:  let \( \pi_1=\pi^*(P) \cup \{\min\text{goal}(g, m_g), \text{activate}(g) | g \in G\} \)
8:  set \( \text{len} = \max\{m_g | g \in G\} \) in \( \pi_1 \)
9:  add the optimization module of \( \pi(P) \) to \( \pi_1 \)
10: compute an answer set \( Y \) of \( \pi_1 \)
11: let \( wcd(P) = d \) where \( wcd(d) \in Y \)
12: compute a set \( S \) of actions that can potentially change \( wcd(P) \) when they are removed
13: set \( w = wcd(P) \) and \( R = \emptyset \)
14: **for** each set \( X \) of at most \( k \) actions in \( S \) **do**
15:  let \( \pi_2 = \pi_1 \cup \{\text{blocked}(a) | a \in X\} \) **\cup** the blocking module of \( \pi(P) \)
16:  compute an answer set \( Z \) of \( \pi_2 \)
17:  if \( wcd(d') \in Z \) & \( d' < w \) **then** set \( w=d' \) and \( R=X \)
18: **return** \( wcd(P), w, R \)

module to the ASP object whose answer sets are computed using clingo. \( \pi^*(P) \) is the planning module of \( \pi(P) \), see below.

\( \pi(P) \) encodes the computation of plans for goals in \( G \), the construction of the longest prefix among plans for the goals, and the removal of actions. Each \( g \in G \) is associated with a trajectory \( t \), an integer between \( 1 \) and \( |G| \). \( \pi(P) \) also uses \( \max \) as \( \text{GRD}(.) \). For communication between \( \pi(P) \) and \( \text{GRD}(.) \), \( \pi(P) \) declares the following external atoms:

\[
\begin{align*}
\#\text{external} & \quad \text{activate}(T) : \text{traj}(T). \quad (57) \\
\#\text{external} & \quad \text{min\_goal}(T, L) : \text{traj}(T), \text{step}(L). \quad (58) \\
\#\text{external} & \quad \text{blocked}(A) : \text{action}(A). \quad (59)
\end{align*}
\]

activate(t) (resp. \( \min\text{goal}(t, l) \)) denotes the active goal (resp. the optimal cost for reaching the \( t \)-th goal) and blocked(a) denotes that the action \( a \) is blocked.

\( \pi(P) \) consists of the following modules:

- **Planning:** A program encoding the domain information \( D \) of \( P \) and the rules for generating optimal plan for each \( g \in G \). This module is similar to the standard encoding in ASP planning (Lifschitz 2002) with an extension to allow for the generation of multiple plans for multiple goals (i.e., \( o(a, t, s) \) is used to denote that action \( a \) occurs at step \( s \) on trajectory \( t \)). To save space, we do not include the set of rules for this module here.

- **Optimization:** A set of rules for determining the longest prefix between two plans of two goals \( g_t \) and \( g_j \) on trajectories \( I \neq J \) given a set of plans for the goals in \( G \). It also contains the optimization statement for selecting answer sets containing \( wcd(P) \).

\[
\begin{align*}
p_{\text{wcd}(0)} & . \quad (60) \\
p_{\text{wcd}(d)} & \leftarrow \text{prefix}(A, I, J, 1, t) \neq I \neq J, o(A, I, J, t), o(A, J, t). \quad (61) \\
p_{\text{wcd}(d)} & \leftarrow \text{prefix}(A, I, J, S+1) \neq I \neq J, \text{prefix}(A, I, J, S), \quad (62) \\
\text{wcd}(D) & \leftarrow D \neq \text{max}\{D : p_{\text{wcd}(D)}\}. \quad (63) \\
\#\text{maximize} & \{D : \text{wcd}(D)\}. \quad (64)
\end{align*}
\]

- **Blocking:** A set of rules that interact with the Python program to block actions from the original problem.

\[
\begin{align*}
\text{occ}(A, T, S) \neq \text{blocked}(A). \quad (65)
\end{align*}
\]

**Properties of the Multi-Shot Encoding.** The correctness of \( \text{GRD}(.) \) follows from the fact that \( \pi(.) \) is correct and that the rules (60)-(64) guarantee that only answer sets containing an atom of the form \( wcd(d) \) with \( d \) being the \( wcd \) of the problem are considered. Let \( P \) be a GRD problem and \( \Pi(P) \) be its multi-shot ASP encoding. We can show:
Proposition 2 If \( GRD(\mathcal{P}, k, \text{max}) \) returns (i) unsolvable then some goal in \( \mathcal{P} \) is not achievable; (ii) \( (d, w, R) \) then \( d = \text{wcd}(\mathcal{P}) \). \( R \) is a solution of \( \mathcal{P} \) wrt. \( k \), and \( w = \text{wcd}(\mathcal{P} \ominus R) \).

The next property of \( \pi(P) \) is used in Lines 16 & 18 of Alg. 1.

Proposition 3 Let \( S \) be an answer set of \( \pi_1 \) with \( \text{len} = \max\{|m_g | g \in G\} \) and \( X \subseteq A \) such that \( |X| \leq k \). If \( X \subseteq \{\exists a(a, t, s) \in S\} = \emptyset \) then \( \text{wcd}(\mathcal{P} \ominus X) = \text{wcd}(\mathcal{P}) \).

Proposition 3 implies that (a) every blocked action (Line 12, Alg. 1) should belong to some answer set of \( \pi_1 \); and (b) if \( \pi_1 \) has an answer set \( Y \) s.t. \( X \) (Line 14) does not contain some action that occurs in \( Y \) then \( X \) should not be considered.

Experimental Results

We evaluated our ASP-based algorithms (labeled SAT-1 and SAT-2 for our 1- and 2-phase saturation-based encoding, respectively, and MS for our multi-shot encoding) against an implementation of the existing PRUNE-REDUCE (labeled PR) algorithm provided by the authors. We also used the same four benchmark domains that they have made publicly available.\(^1\) (1) GRID-Navigation, where each instance is defined by the \( x \)- and \( y \)-dimensions; (2) IPC-GRID\textsuperscript{*}, where each instance is defined by the \( x \)- and \( y \)-dimensions and the number of locks/keys; (3) BLOCKWORDS, where each instance is defined by the number of blocks and words/goals; and (4) LOGISTICS, where each instance is defined by the number of airplanes, airports, locations, cities, trucks, and packages. We set \( k = \{1, 2\} \) as suggested in the benchmarks, conducted our experiments on a 3.60GHz CPU machine with 8GB of RAM, and set a timeout of 5 hours.

Table 1 tabulates the results. In general, SAT-2 performs best in terms of efficiency and scalability, followed by MS, PR, and SAT-1. The reason SAT-1 performs poorly is that it computes optimal plan lengths for each goal (and uses these in the \text{wcd} calculations) in one solver call. All these independent subproblems hinder the performance of the solver when combined with the main \text{wcd} computation. In contrast, SAT-2 separates and solves the independent subproblems, which explains its better performance.

MS and PR compute the \text{wcd} of the initial problem and then systematically search for a set of actions to block in order to minimize the \text{wcd}. Their systematic nature means that the number of calls to the ASP solver (for MS) or the classical planner (for PR) is proportional to the size of possible combinations of blocked actions. In large instances, such as logistics instances, this number can be in the millions. Thus, SAT-2 is better than MS and PR. MS is slightly better than PR because it is able to reduce the number of possible combinations that needs to be considered in the second phase.

Conclusions

In this paper, we investigate declarative approaches, specifically ASP-based algorithms, to solve GRD problems. Our ASP-based algorithms outperform PRUNE-REDUCE, the current state-of-the-art imperative GRD solver, on common GRD benchmarks, thereby contributing to both the GRD community, by extending the state-of-the-art GRD solver, as well as the ASP community, by increasing the applicability of ASP to other areas. This paper thus makes the first step of bridging the two areas of GRD and ASP in an effort towards better cross-fertilization of both areas.

References


\(^1\)http://technion.ac.il/~sarahn/final-benchmarks-icaps-2014/.