Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Solving Goal Recognition Design Using ASP

Tran Cao Son,[†] Orkunt Sabuncu,[‡] Christian Schulz-Hanke,[‡]
Torsten Schaub,[‡◇] William Yeoh[†]
[†]New Mexico 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 recognition 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.

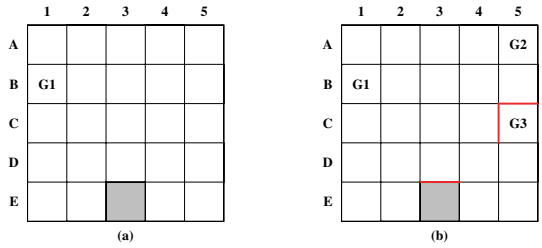

Figure 1: Example Problem

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 $\langle \mathbf{F}, s_0, \mathbf{A}, \mathbf{C}, \mathbf{G} \rangle$, where $\mathbf{F}$ is a set of fluents; s_0 is the start state of the agent; $\mathbf{A}$ is the set of actions; $\mathbf{C} : \mathbf{A} \rightarrow \mathbb{R}$ defines the cost for each action; and $\mathbf{G}$ is a set of goal states. Each action $a \in \mathbf{A} = \langle pre(a), add(a), del(a) \rangle$ is a triplet consisting of the precondition, add, and delete lists, respectively, and all are subsets of $\mathbf{F}$. An action a is applicable in state s if $pre(a) \subseteq s$. If action a is applied in state s, then it results in a new state $s' = (s \setminus del(a)) \cup add(a)$. 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 $g \in \mathbf{G}$. The cost of a plan $\mathbf{C}(\pi) = \sum_i \mathbf{C}(a_i)$ is the sum of the cost of each individual action in the plan. The goal is typically to find a cost-minimal plan $\pi^* = \operatorname{argmin}_\pi \mathbf{C}(\pi)$.

Goal Recognition Design (GRD). A GRD problem (Keren, Gal, and Karpas 2014) is represented as a tuple $\mathcal{P}{=}\langle D, \mathbf{G} \rangle$, where $D{=}\langle \mathbf{F}, s_0, \mathbf{A}, \mathbf{C} \rangle$ captures the domain information and $\mathbf{G}$ 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 $\pi = \langle a_1, \ldots, a_k \rangle$ that is the prefix in cost-minimal plans $\pi^*_{g_1}$ and $\pi^*_{g_2}$ to distinct goals $g_1, g_2 \in \mathbf{G}$.*

Using our example problem of Figure 1(a), a longest sequence of actions that can lead to two distinct goals is $\langle (E3, up), (D3, up), (C3, right), (C4, right) \rangle$, 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 $X \subset \mathbf{A}$, let $\mathcal{P} \ominus X$ be the problem $\hat{\mathcal{P}} = \langle \hat{D}, \mathbf{G} \rangle$ where $\hat{D} = \langle \mathbf{F}, s_0, \mathbf{A} \setminus X, \mathbf{C} \rangle$. The objective in GRD problems is to find a subset of actions $\hat{\mathbf{A}}^* \subset \mathbf{A}$ such that if they are removed from the set of actions $\mathbf{A}$, then the *wcd* of the resulting problem $\hat{\mathcal{P}}$ is minimal. This optimization problem is subject to the requirement that the cost of cost-minimal plans to achieve each goal $g \in \mathbf{G}$ 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 $\hat{\mathbf{A}}^*$, called a* solution to $\mathcal{P}$ wrt *k, such that*

$$\hat{\mathbf{A}}^* = \operatorname*{argmin}_{\hat{\mathbf{A}} \subset \mathbf{A}} wcd(\mathcal{P} \ominus \hat{\mathbf{A}})$$
$$\textit{subject to} \quad \mathbf{C}(\pi^*_g) = \mathbf{C}(\hat{\pi}^*_g) \qquad \forall g \in \mathbf{G}$$
$$|\hat{\mathbf{A}}^*| \leq k$$

where π^*_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 $\mathcal{P} \ominus \hat{\mathbf{A}}^*$.

For example, if $k{=}3$, then blocking the actions $(E3, up), (C4, right), (C5, up)$ in our example problem, where we use pairs (s, a) to denote 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 $(E3, left)$, we know that the agent's goal is G1; (*b*) if the agent executes the action $(E3, right)$, then the goal is G2 or G3; (*c*) if the agent continues with the action $(E4, right)$, its goal must be G3; (*d*) after the agent executes the sequence $\langle (E3, right), (E4, up) \rangle$, it must reveal its goal by either $(D4, right)$ or $(D4, up)$. This implies that the longest sequence of actions that can be executed by the agent before it must reveal its goal is $\langle (E3, right), (E4, up) \rangle$, 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, WCD-BFS and LATEST-SPLIT, to compute the *wcd* of a problem. WCD-BFS uses breadth-first search (BFS) to explore all combinations of paths and prunes subsets of paths that are provably distinctive. LATEST-SPLIT compiles the problem into a set of classical planning problems and solves them using any classical planner. They also introduced two algorithms, EXHAUSTIVE-REDUCE and PRUNED-REDUCE, to reduce the *wcd* of a problem. EXHAUSTIVE-REDUCE uses a variation of BFS to exhaustively search the whole search space in the worst case. PRUNED-REDUCE optimizes EXHAUSTIVE-REDUCE by pruning some portions of the search space.

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

$$c_1 \mid \ldots \mid c_k \leftarrow a_1, \ldots, a_m, \ not\ a_{m+1}, \ldots, \ not\ a_n \quad (1)$$

where $0{\leq}m{\leq}n$, $0{\leq}k$, each a_i or c_j is a literal of a propositional language and $not\, a$ 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 induces a collection of so-called *answer sets*, which are distinguished models of determined by answer sets 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 Soininen 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 in the cardinality constraint. 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's antecedent expands to the conjunction of all instances of $a(X)$

for which the corresponding instance of $b(X)$ holds. Similarly, $2\ \{a(X)\ :\ b(X)\}\ 4$ 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 c_j are expressed as $\#minimize\ \{w_1 : c_1, \ldots, w_n : c_n\}$.

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 $\mathcal{P}{=}\langle D, \mathbf{G}\rangle$ where $D{=}\langle \mathbf{F}, s_0, \mathbf{A}, \mathbf{C}\rangle$ be a GRD problem and k be a positive integer denoting the maximal number of actions that can be blocked for reducing the `wcd` of $\mathcal{P}$

A Saturation-based Meta Encoding. We employ the method of encoding a planning problem by a set of facts to encode D and $\mathbf{G}$. Specifically, (*i*) $\mathbf{F}$ is encoded by a set of atoms of the form $fluent(f)$ for $f \in \mathbf{F}$; (*ii*) $\mathbf{A}$ by $action(a)$ for $a \in \mathbf{A}$; (*iii*) s_0 by $init(l)$ for $l \in s_0$; (*iv*) each $g \in \mathbf{G}$ by $goal(g, l)$ for l as a conjunct in g; (*v*) $pre(a)$ by a set of atoms of the form $exec(a, l)$; (*vi*) $add(a)$ ($del(a)$) by $effect(a, f, id)$ ($effect(a, \neg f, id)$) where id is a unique identifier associated with an effect of a; (*vii*) each condition of a conditional effect by $cond(a, l, id)$ for l as a conjunct in condition for the effect of action a associated with id. 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 Σ_2^P-complete problems. A typical problem in this class is the satisfiability problem for $\exists\forall$-QBF. For instance, in a typical encoding for satisfiability of a $\exists\forall$-QBF the *guess part* uses disjunction to generate all possible truth values for the propositional atoms that are quantified by $\forall$ ($\forall$-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 wrt 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, and Scarcello 2001).

Let $vl(x, y, c)$ denote that c is the common prefix of minimal cost plans of π_x^* and π_y^*. It is easy to see that the following $\exists\forall$-QBF encodes the *wcd* definition

$$\exists x, y, c[vl(x, y, c) \wedge [\forall x', y', c'[vl(x', y', c') \rightarrow |c| \geq |c'|]] \quad (2)$$

where, for the sake of simplicity, we omit some details such as $x, y, x', y' \in \mathbf{G}$, and that c and c' correspond to sequences of actions that are the common prefix of cost-optimal plans π_x^* and π_y^*, $\pi_{x'}^*$ and $\pi_{y'}^*$, respectively.

To compute the *wcd*, we only need to encode the satisfiability of formula (2). As in any ASP encoding for planning, we assume a finite horizon len and use $st(t)$ to represent time steps from 1 to len. For each fluent $f \in \mathbf{F}$, $comp(f, f, \neg f)$ gives its complementary literals. For convenience all fluent literals are encoded by $lit(f)$ and $lit(\neg f)$ for $f \in \mathbf{F}$. The rules for the *guess* and *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 $pacts/1$) 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 $pacts(A)$) 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 ($h(P, F, T) : goal(G, Pos), comp(F, Pos, Neg)$) and similarly all fluents occurring as a negative literal must not hold ($not\ \ h(P, F, T)\ \ :\ \ goal(G, Neg), comp(F, Pos, Neg)$). (12)–(14) compute the common prefix c using the $same/1$ and $follow/1$ predicates. $same(T)$ states the same action occurs at step T concerning the goals x and y and instances of $follow(1) \ldots follow(n)$ in an answer set s.t. $n \leq len$, represent the path followed by the agent from start to step n is a non-distinctive path.

$$\{blocked(A)\} \leftarrow action(A). \quad (3)$$

$$\leftarrow k + 1\{blocked(A)\}. \quad (4)$$

$$pacts(A) \leftarrow action(A), not\ blocked(A). \quad (5)$$

$$1\{s(1, G) : goal(G, \_)\}1 \leftarrow . \quad (6)$$

$$1\{s(2, G) : goal(G, \_)\}1 \leftarrow . \quad (7)$$

$$\leftarrow s(1, G), s(2, G). \quad (8)$$

$$\{o(A, P, T) : pacts(A)\}1 \leftarrow s(P, G), st(T). \quad (9)$$

$$goalat(P, T) \leftarrow s(P, G), st(T), \quad (10)$$
$$not\ h(P, F, T) : goal(G, Neg), comp(F, Pos, Neg);$$
$$h(P, F, T) : goal(G, Pos), comp(F, Pos, Neg).$$

$$\leftarrow s(P, \_), not\ goalat(P, len). \quad (11)$$

$$same(T) \leftarrow o(A, 1, T), o(A, 2, T). \quad (12)$$

$$follow(1) \leftarrow same(1). \quad (13)$$
$$follow(T{+}1) \leftarrow follow(T), same(T{+}1), len{>}T. \quad (14)$$

In the *guess part*, we generate a search space for the ∀-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 $pos(1p)$ and $pos(2p)$). $nas(P,G)$ 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 $o/3$.

$$as(P,G)|nas(P,G) \leftarrow pos(P), goal(G,\_). \quad (15)$$
$$ao(A,P,T)|nao(A,P,T) \leftarrow pos(P), pacts(A), st(T). \quad (16)$$

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.

$$invalid \leftarrow 2\{as(P,G)\}, pos(P). \quad (17)$$
$$invalid \leftarrow pos(P), nas(P,G) : goal(G,\_). \quad (18)$$
$$invalid \leftarrow as(1p,G), as(2p,G). \quad (19)$$
$$invalid \leftarrow 2\{ao(A,P,T)\}, pos(P), st(T). \quad (20)$$
$$invalid \leftarrow as(P,G), goal(G,Pos), comp(F,Pos,Neg), nh(P,F,len). \quad (21)$$
$$invalid \leftarrow as(P,G), goal(G,Neg), comp(F,Pos,Neg), h(P,F,len). \quad (22)$$

Using meta-programming techniques, similar to Gebser, Kaminski, and Schaub (2011), we encode the action domain, in which we obey syntactive 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 $nh/3$) since we cannot use naf-literals such as $not\ h(P,F,T)$ in this context.

$$h(P,F,0) \leftarrow s(P,\_), init(Pos), comp(F,Pos,Neg). \quad (23)$$
$$h(P,F,0) \leftarrow pos(P), init(Pos), comp(F,Pos,Neg). \quad (24)$$
$$nh(P,F,0) \leftarrow pos(P), init(Neg), comp(F,Pos,Neg). \quad (25)$$
$$nh(P,F,0) \leftarrow pos(P), not\ init(Pos), not\ init(Neg), comp(F,Pos,Neg). \quad (26)$$

Next we represent the action preconditions in (27)–(31). While representing the constraint that no action with an unsatisfied precondition can occur in c (cf. (27)–(28)), we can use integrity constraints and naf-literals (since only ∃-variables play a role). However, we have to use $nh/3$ to represent non-executability of an action in c' (in (29)–(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.

$$\leftarrow o(A,P,T), exec(A,Pos), comp(F,Pos,Neg), not\ h(P,F,T). \quad (27)$$
$$\leftarrow o(A,P,T), exec(A,Neg), comp(F,Pos,Neg), h(P,F,T). \quad (28)$$
$$nposs(P,A,T) \leftarrow pos(P), st(T), exec(A,Pos), comp(F,Pos,Neg), nh(P,F,T-1). \quad (29)$$
$$nposs(P,A,T) \leftarrow pos(P), st(T), exec(A,Neg), comp(F,Pos,Neg), h(P,F,T-1). \quad (30)$$
$$invalid \leftarrow ao(A,P,T), nposs(P,A,T). \quad (31)$$
$$c(P,L,T) \leftarrow o(A,P,T), effect(A,L,I), h(P,F,T-1) : cond(A,Pos,I), comp(F,Pos,Neg); not\ h(P,F,T-1) : cond(A,Neg,I), comp(F,Pos,Neg). \quad (32)$$
$$c(P,L,T) \leftarrow ao(A,P,T), effect(A,L,I), h(P,F,T-1) : cond(A,Pos,I), comp(F,Pos,Neg); nh(P,F,T-1) : cond(A,Neg,I), comp(F,Pos,Neg). \quad (33)$$
$$h(P,F,T) \leftarrow c(P,Pos,T), comp(F,Pos,Neg). \quad (34)$$
$$nh(P,F,T) \leftarrow pos(P), c(P,Ne,T), comp(F,Po,Ne). \quad (35)$$

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 ∃-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 ∀-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 wrt a fluent literal and a time point. One case is that at this time point there are no occurrances 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)).

$$nc(P,L,T) \leftarrow pos(P), st(T), lit(L), nao(A,P,T) : effect(A,L,I), pacts(A). \quad (36)$$
$$fail(P,A,L,I,T) \leftarrow ao(A,P,T), effect(A,L,I), cond(A,Pos,I), comp(F,Pos,Neg), nh(P,F,T-1). \quad (37)$$
$$fail(P,A,L,I,T) \leftarrow ao(A,P,T), effect(A,L,I), cond(A,Neg,I), comp(F,Pos,Neg), h(P,F,T-1). \quad (38)$$
$$nc(P,L,T) \leftarrow ao(A,P,T), lit(L), fail(P,A,L,I,T) : effect(A,L,I). \quad (39)$$
$$h(P,F,T) \leftarrow h(P,F,T-1), not\ c(P,Neg,T), st(T), comp(F,Pos,Neg), s(P,\_). \quad (40)$$
$$h(P,F,T) \leftarrow h(P,F,T-1), nc(P,Neg,T), st(T), comp(F,Pos,Neg), pos(P). \quad (41)$$

$$nh(P,F,T) \leftarrow nh(P,F,T-1), nc(P,Pos,T), st(T), \quad (42)$$
$$comp(F,Pos,Neg), pos(P).$$

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 $|c_T| \geq |c'_T|$ holds where c_T or c'_T denotes the common prefixes considering only the paths formed by selected actions from start to step T for selected goals x, y and 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)).

$$nafollow(T) \leftarrow ao(A,P,T), nao(A,P1,T), P \neq P1. \quad (43)$$
$$nafollow(T+1) \leftarrow nafollow(T), len > T. \quad (44)$$
$$geq(0) \leftarrow . \quad (45)$$
$$geq(T) \leftarrow geq(T-1), nafollow(T). \quad (46)$$
$$geq(T) \leftarrow geq(T-1), follow(T). \quad (47)$$
$$goal \leftarrow geq(len). \quad (48)$$

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.

$$sat \leftarrow invalid. \quad (49)$$
$$sat \leftarrow goal. \quad (50)$$
$$\leftarrow not\ sat. \quad (51)$$
$$as(P,G) \leftarrow sat, pos(P), goal(G,\_). \quad (52)$$
$$nas(P,G) \leftarrow sat, pos(P), goal(G,\_). \quad (53)$$
$$ao(A,P,T) \leftarrow sat, pos(P), pacts(A), st(T). \quad (54)$$
$$nao(A,P,T) \leftarrow sat, pos(P), pacts(A), st(T). \quad (55)$$

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).

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

Note that by canceling out rule (4) and adding a second level optimization statement $\#minimize\{1@1, A : blocked(A)\}$ 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 c and c' consider only cost-optimal plans of the selected goals and the condition $\mathbf{C}(\pi^*_g) = \mathbf{C}(\hat{\pi}^*_g)$ for each goal $g \in \mathbf{G}$, 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 π using only unblocked actions for each goal as an abstract $\exists$-atom. In the guess part, we guess another plan π' considering all actions as an abstract $\forall$-atom. The check part checks the condition $|\pi| \leq |\pi'|$ for valid guesses to guarantee that π is an optimal plan. Additionally, since π' 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.

Let Π_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 $\Pi_{\mathcal{P}}$ is the instance program of $\mathcal{P}$. Let len be the maximal length of plans to all the goals in $\mathcal{P}$. The set $\hat{\mathbf{A}}$ of blocked actions is a solution of $\mathcal{P}$ s.t. $|\hat{\mathbf{A}}| \leq k$ and $w = \mathtt{wcd}(\mathcal{P} \ominus \hat{\mathbf{A}})$ is minimal iff $\Pi = \Pi_E \cup \Pi_{\mathcal{P}}$ has an answer set M s.t. $\hat{\mathbf{A}} = \{a \mid blocked(a) \in M\}$ and $w = |\{follow(i) \mid 1 \leq i \leq t, follow(i) \in M\}|$.*

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 and let the branch-and-bound style optimization of the ASP solver handle the outer optimization. To the best of our knowledge, this is the first comprehensive ASP encoding solving minmax/maxmin optimization problems. The elaborated research on this methodology and applying it to other areas are among our future line of research.

A Multi-Shot ASP Encoding. Let $\mathcal{P} = \langle D, \mathbf{G}\rangle$ be a GRD problem, where $D = \langle \mathbf{F}, s_0, \mathbf{A}, \mathbf{C}\rangle$, and k be a positive integer as described earlier. 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*) $\mathtt{wcd}(\mathcal{P})$; 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 $GRD(\langle D, \mathbf{G}\rangle, k, max)$ (or $GRD(.)$).

Lines 3–6 of Alg. 1 compute the optimal cost plan for each $g \in \mathbf{G}$. If some goal has no plan of length at most max then $GRD(.)$ returns *unsolvable*. Otherwise, the optimal cost is used as the bound for $\pi(\mathcal{P})$ (Line 8). $\mathtt{wcd}(\mathcal{P})$ is computed by computing an answer set of $\pi(\mathcal{P})$ with len equals 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 13–17 implement a simple exhaustive search to identify a set of at most k actions that reduce $\mathtt{wcd}(\mathcal{P})$. $GRD(.)$ controls the computation of $\mathtt{wcd}(\mathcal{P})$ and a solution of $\mathcal{P}$ wrt a given k assuming that the maximal length of plans to all goals in $\mathbf{G}$ 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

(a) $k = 1$

Domain	Instances	*wcd* reduction	Runtime (s) PR	ASP1	ASP2	MS
GRID-NAVIGATION	5-14	$9 \rightarrow 9$	12	26	1	1
	19-10	$17 \rightarrow 17$	12	18	1	1
	20-9	$39 \rightarrow 39$	23	406	3	3
	16-11	$4 \rightarrow 4$	11	12	1	1
	16-11	$4 \rightarrow 4$	12	10	1	1
IPC–GRID$^+$	5-5-5	$4 \rightarrow 3$	14	9	1	1
	5-10-10	$11 \rightarrow 11$	194	475	14	11
	10-5-5	$12 \rightarrow 10$	46	36	2	1
	10-10-10	$19 \rightarrow 19$	2,661	1,257	33	30
BLOCK-WORDS	8-20	$10 \rightarrow 10$	946	timeout	64	48
	8-20	$14 \rightarrow 14$	809	timeout	121	71
LOGISTICS	1-2-6-2-2-6	$18 \rightarrow 18$	3,506	timeout	151	228
	1-2-6-2-2-6	$18 \rightarrow 18$	2,499	timeout	135	140
	2-2-6-2-2-6	$17 \rightarrow 17$	3,173	timeout	352	756
	2-2-6-2-4-6	$17 \rightarrow 17$	timeout	timeout	5,377	1,943
	2-2-6-2-6-6	$16 \rightarrow 16$	timeout	timeout	5,166	timeout

(b) $k = 2$

wcd reduction	Runtime (s) PR	ASP1	ASP2	MS
$9 \rightarrow 8$	50	811	2	12
$17 \rightarrow 17$	12	488	1	18
$39 \rightarrow 39$	23	2,980	3	74
$4 \rightarrow 3$	24	147	1	8
$4 \rightarrow 3$	24	63	1	5
$4 \rightarrow 3$	33	62	1	5
$11 \rightarrow 11$	194	10,092	14	362
$12 \rightarrow 10$	92	1,022	2	36
$19 \rightarrow 19$	2,665	timeout	33	2,208
$10 \rightarrow 10$	3,927	timeout	178	938
$14 \rightarrow 14$	3,482	timeout	218	1,015
$18 \rightarrow 18$	3,527	timeout	155	639
$18 \rightarrow 18$	2,496	timeout	137	483
$17 \rightarrow 17$	timeout	timeout	594	1,943
$17 \rightarrow 7$	timeout	timeout	6,752	6,065
$16 \rightarrow 16$	timeout	timeout	5,215	timeout

Table 1: Experimental Results

Algorithm 1 $GRD(\langle D, \mathbf{G}\rangle, k, max)$

Input: a GRD problem $\mathcal{P}{=}\langle D, \mathbf{G}\rangle$ & integers k, max.
Output: $\texttt{wcd}(\mathcal{P})$, and a solution R of $\mathcal{P}$ w.r.t. k and $\texttt{wcd}(\mathcal{P}{\ominus}R)$ or unsolvable if some goal is not achievable.
for each goal g in $\mathbf{G}$ **do**
  compute the minimal length of plan for g
  if plan of length $i \leq max$ exists **then** set $m_g = i$
  else return *unsolvable*
let $\pi_1{=}\pi^*(\mathcal{P}){\cup}\{min\_goal(g, m_g), activate(g)|g{\in}\mathbf{G}\}$
set $len = \max\{m_g \mid g \in \mathbf{G}\}$ in π_1
add the optimization module of $\pi(\mathcal{P})$ to π_1
compute an answer set Y of π_1
let $\texttt{wcd}(\mathcal{P}) = d$ where $wcd(d) \in Y$
compute a set S of actions that can potentially change $\texttt{wcd}(\mathcal{P})$ when they are removed
set $w = \texttt{wcd}(\mathcal{P})$ and $R = \emptyset$
for each set X of at most k actions in S **do**
  let $\pi_2 = \pi_1 \cup \{blocked(a) \mid a \in X\}{\cup}$ the blocking module of $\pi(\mathcal{P})$
  compute an answer set Z of π_2
  if $wcd(d') \in Z$ & $d' < w$ **then** set $w{=}d'$ and $R{=}X$
return $\langle \texttt{wcd}(\mathcal{P}), w, R\rangle$

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

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

$$\#external \quad activate(T) : traj(T). \tag{57}$$
$$\#external \quad min\_goal(T, L) : traj(T), step(L). \tag{58}$$
$$\#external \quad blocked(A) : action(A). \tag{59}$$

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

$\pi(\mathcal{P})$ consists of the following modules:

- *Planning*: A program encoding the domain information D of $\mathcal{P}$ and the rules for generating optimal plan for each $g \in \mathbf{G}$. 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_I and g_J on trajectories $I \neq J$ given a set of plans for the goals in $\mathbf{G}$. It also contains the optimization statement for selecting answer sets containing $\texttt{wcd}(\mathcal{P})$.

$$p\_wcd(0). \tag{60}$$
$$prefix(A, I, J, 1) \leftarrow I \neq J, o(A, I, 1), o(A, J, 1).$$
$$prefix(A, I, J, S{+}1) \leftarrow I \neq J, prefix(A, I, J, S) \tag{61}$$
$$o(A, I, S + 1), o(A, J, S + 1),$$
$$p\_wcd(D) \leftarrow prefix(\_, \_, \_, D). \tag{62}$$
$$wcd(D) \leftarrow D = \#max\,\{D : p\_wcd(D)\}. \tag{63}$$
$$\#maximize\,\{D : wcd(D)\}. \tag{64}$$

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

$$\leftarrow occ(A, T, S), blocked(A). \tag{65}$$

Properties of the Multi-Shot Encoding. The correctness of $GRD(.)$ 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 $\texttt{wcd}(d)$ with d being the wcd of the

problem are considered. Let $\mathcal{P}$ be a GRD problem and $\Pi(\mathcal{P})$ be its multi-shot ASP encoding. We can show:

Proposition 2 *If $GRD(\mathcal{P}, k, max)$ returns (*i*)* unsolvable *then some goal in $\mathcal{P}$ is not achievable; (*ii*) $\langle d, w, R\rangle$ then d=$\texttt{wcd}(\mathcal{P})$, R is a solution of $\mathcal{P}$ wrt k, and w=$\texttt{wcd}(\mathcal{P}\ominus R)$.*

The next property of $\pi(P)$ is used in Lines 12 &14 of Alg. 1.

Proposition 3 *Let S be an answer set of π_1 with $len = \max\{m_g \mid g \in \mathbf{G}\}$ and $X \subseteq \mathbf{A}$ such that $|X| \leq k$. If $X \cap \{a \mid \exists\, o(a,t,s) \in S\} = \emptyset$ then $\texttt{wcd}(\mathcal{P} \ominus X) = \texttt{wcd}(\mathcal{P})$.*

Proposition 3 implies that (*a*) every action selected for blocking (Line 12, Alg. 1) should belong to an answer set of π_1; and (*b*) if there exists some answer set Y of π_1 and X (Line 14, Alg. 1) does not contain some action that occurs in Y then X should not be considered.

Experimental Results

We evaluated our ASP-based algorithms (labeled ASP1 and ASP2 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$^+$, 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, ASP2 performs best in terms of efficiency and scalability, followed by MS, PR, and ASP1. The reason ASP1 performs poorly is that it computes optimal plan lengths for each goal (and uses these in the *wcd* calculations) in one solver call. All these independent subproblems hinder the performance of the solver when combined with the main *wcd* computation. In contrast, ASP2 separates and solves the independent subproblems, which explains its better performance.

MS and PR compute the *wcd* of the initial problem and then systematically search for a set of actions to block in order to minimize the *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, ASP2 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

We investigated 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 GRD, by extending the state-of-the-art GRD solver, as well as to ASP, by increasing the applicability of ASP to other areas. We thus make a first step in bridging the GRD and ASP in an effort towards better cross-fertilization of both.

Acknowledgments. This work was partially funded by DFG (550/9) and by NSF grant HRD-1345232.

[1] http://technion.ac.il/ sarahn/final-benchmarks-icaps-2014/.

References

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems*. Springer. 40–110.

Fikes, R., 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.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of ICLP*.

Gebser, M.; Kaminski, R.; and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11(4-5):821–839.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Geib, C. W., and Steedman, M. 2007. On natural language processing and plan recognition. In *Proceedings of IJCAI*, 1612–1617.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proceedings of ICLP*, 579–597.

Jarvis, P.; Lunt, T. F.; and Myers, K. L. 2005. Identifying terrorist activity with AI plan recognition technology. *AI Magazine* 26(3):73–81.

Kabanza, F.; Bellefeuille, P.; Bisson, F.; Benaskeur, A. R.; and Irandoust, H. 2010. Opponent behaviour recognition for real-time strategy games. In *Proceedings of PAIR*.

Keren, S.; Gal, A.; and Karpas, E. 2014. Goal recognition design. In *Proceedings of ICAPS*.

Leone, N.; Rosati, R.; and Scarcello, F. 2001. Enhancing answer set planning. In *Proceedings of PUII*.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138(1–2):39–54.

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

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3,4):241–273.

Ramírez, M., and Geffner, H. 2011. Goal recognition over pomdps: Inferring the intention of a POMDP agent. In *Proceedings of IJCAI*, 2009–2014.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.