

Answer Set Programming and Its Applications in Planning and Multi-agent Systems

Tran Cao Son^(✉)

Department of Computer Science, New Mexico State University,
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

Abstract. The paper presents some applications in planning and multi-agent systems of answer set programming. It highlights the benefits of answer set programming based techniques in these applications. It also describes a class of multi-agent planning problems that is challenging to answer set programming.

Keywords: Answer set programming · Planning · Multi-agent system

1 Introduction

The invention of answer set programming (ASP) [18, 20] and the development of efficient answer set solvers such as `smodels` [24], `dlv` [5], and `clingo` [8] enable the use of logic programming under answer set semantics in several practical applications [6]. The fundamental idea of ASP is to represent solutions to a problem by answer sets of a logic program. That is, to solve a problem, one first represents it as a logic program whose answer sets correspond one-to-one to its solutions; next, to find a solution, one computes an answer set of that program and extracts the solution from the answer set.

Formally, a logic program Π is a set of rules of the form

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

where $0 \leq m \leq n$, $0 \leq k$, each a_i or c_j is a literal of a propositional language¹ and *not* represents *default negation*. Both the head and the body can be empty. When the head is empty, the rule is called a *constraint*. When the body is empty, the rule is called a *fact*. The semantics of a program Π is defined by a set of *answer sets* [10]. An answer set is a distinguished model of Π that satisfies all the rules of Π and is minimal and well-supported.

To increase the expressiveness of logic programs and simplify its use in applications, the language has been extended with several features such as *weight constraints* or *choice atoms* [24], or *aggregates* [7, 21, 25]. Standard syntax for these extensions has been proposed and adopted in most state-of-the-art ASP-solvers such as `clingo` and `dlv`.

In recent years, attempts have been made to consider continuously changing logic programs or external atoms. For example, the system `clingo` enables the multi-shot model as oppose to the traditional single-shot model. In this model, ASP programs are

¹ Rules with variables are viewed as a shorthand for the set of their ground instances.

extended with Python procedures that control the answer set solving process along with the evolving logic programs. This feature provides an effective way for the application of ASP in a number of applications that were difficult to deal with previously.

This paper describes the application of ASP in planning in the presence of incomplete information and sensing actions (Sect. 2), in goal recognition design (Sect. 3), and in various settings of multi-agent planning (Sect. 4). It highlights the advantage of ASP in these researches and, when possible, identifies the challenging issues faced by ASP.

2 Planning with Incomplete Information and Sensing Actions

Answer set planning was one of the earliest applications of answer set programming [3, 16, 32]. The logic program encoding proposed in these papers are suitable for classical planning problems with complete information about the initial state and deterministic actions. In a series of work, we applied ASP to *conformant planning* and *conditional planning* (e.g., [30, 31, 34, 35]). The former refers to planning with incomplete information about the initial state whose solutions are action sequences that achieve the goal from any possible initial state (and hence, the terms *conformant planning*). The latter refers to planning with incomplete information and sensing actions whose solutions often contain *branches* in the form of conditional statements (e.g., **if-then-else** or **case-statement**) that leads to the terms *conditional planning*.

Conditional planning is computationally harder than conformant planning which, in turn, is computationally harder than classical planning. When actions are deterministic and the plan’s length is polynomially bounded by the size of the problem, the complexity of conditional and conformant planning are PSPACE-complete and Σ_2^P -complete, respectively, [1]. As such, there are problems that has conditional plan as solution but does not have conformant plan as solution. The following example highlights this issue.

Example 1 (From [34]). Consider a security window with a lock that can be in one of the three states *opened*, *closed*² or *locked*³. When the window is closed or opened, pushing it *up* or *down* will *open* or *close* it respectively. When the window is closed or locked, flipping the lock will lock or close it respectively.

Suppose that a robot needs to make sure that the window is locked and initially, the robot knows that the window is not open (but whether it is locked or closed is unknown).

No conformant plan can achieve the goal. Instead, the robot needs a conditional plan consisting of the following steps: (1) checks the window to determine the window’s status; if the window is closed, (2.a) locks the window; otherwise (i.e., the window is already locked), (2.b) does nothing. \square

The proposed ASP-based systems for conditional and conformant planning in [30, 31, 34, 35] show that ASP-based planners performed well comparing to state-of-the-art planning systems of the same kind in several domains. Their performance can be attributed to the following key ideas:

² The window is closed and unlocked.

³ The window is closed and locked.

- The use of an input language that allows for the representation and reasoning with static causal laws (a.k.a. axiom or domain constraints). It should be noted that the original specification of the Planning Domain Description Language (PDDL) – a language frequently used for the specification of planning problems by the planning community – includes axioms⁴ which correspond to non-recursive static causal laws in our terminology [11]. However, the conformant planning benchmarks designed by the planning community do not use static causal laws.
- The employment of an approximation semantics that reduces the computational complexity of planning with incomplete information to **NP**-complete.

The next example highlights the advantage of directly dealing with static causal laws.

Example 2 (Dominos Domain [31]). Suppose that we have n dominos standing on a line in such a way that if one domino falls then the domino on its right also falls. There is also a ball hanging close to the leftmost domino. Swinging the ball will cause the leftmost domino to fall. Initially, the ball stays still and whether or not the dominos are standing is unknown. The goal is to have the rightmost domino to fall. Obviously, swinging the ball is the only plan to achieve this goal, no matter how big n is.

The problem can be easily expressed by a theory with a set of objects $1, \dots, n$ denoting the dominos from left to right and a single action *swing* that causes $down_1$ (the leftmost domino falls) to be true, and $n - 1$ axioms (state constraints) $down_i \Rightarrow down_{i+1}$ representing the fact that $down_{i+1}$ is true if $down_i$ is true. The goal is to have $down_n$ become true.

State constraints are usually dealt with by compiling them away. According to the compilation suggested in [33], for each axiom $down_i \Rightarrow down_{i+1}$, we introduce a new action e_i whose effect is $down_{i+1}$ and whose precondition is $down_i$. Clearly, under this compilation, the plan to achieve the goal is the sequence of actions $[swing, e_1, \dots, e_{n-1}]$.

The main problem with this compilation is that the plan length increases with the number of objects. Even when it is only linear to the size of the original problem, it proves to be challenging for planners following this approach. Most planners have problem when plan length is greater than 500 (i.e., more than 500 dominos). \square

The input language is the action language \mathcal{A}^c (in [30, 31, 35]) and \mathcal{A}_K^c (in [34]). Since \mathcal{A}_K^c is an extension of \mathcal{A}^c with sensing actions, we summarize the features of \mathcal{A}_K^c below. An action theory in \mathcal{A}_K^c is a collection of statements of the following forms:

$$\begin{array}{llll}
 \text{initially}(l) & (2) & \text{causes}(a, l, \phi) & (4) \\
 \text{executable_if}(a, \psi) & (3) & \text{if}(l, \varphi) & (5) \\
 & & \text{determines}(a, \theta) & (6)
 \end{array}$$

where a is an action, l is a fluent literal, and $\psi, \phi, \varphi, \theta$ are sets of literals⁵. (2) says that l holds in the initial situation. (3) says that a is executable in any situation in which ψ holds (the precise meaning of *hold* will be given later). (4) represents a conditional effect of an action. It says that performing a in a situation in which ϕ holds causes l to hold in the successor situation. (5), called a *static causal law*, states that l holds in any situation in which φ holds. (6) states that the values of literals in θ , sometimes referred to as *sensed-literals*, will be known after a is executed.

⁴ In our view, static causal laws can be used to represent relationships between fluents and thus could be considered as axioms in PDDL.

⁵ A set of literals is interpreted as the conjunction of its members. \emptyset denotes *true*.

The complete semantics of \mathcal{A}_K^c can be found in [2]. It defines a transition function Φ over pairs of actions and sets of belief states. The approximation semantics employed in the systems in [30, 31, 34, 35] defines a transition function Φ_a over pairs of actions and approximation states where an approximation state is a set of consistent literals satisfying the static causal laws. Φ_a can be defined in set theoretical terms [31, 34] or by logic program rules [35]. Its precise definition can be found in the aforementioned papers. Φ_a can be used for conformant planning in ASP, in the same way that the transition function of the action language is used for planning as described in [16].

Given a planning problem instance $\mathcal{P} = (\mathcal{D}, \mathcal{I}, \mathcal{G})$, where \mathcal{D} is a set of statements of the forms (3)–(6), \mathcal{I} is a set of statements of the form (2), and \mathcal{G} is a fluent formula. Let k be an integer denoting the maximal length of the desirable solutions. We solve \mathcal{P} by translating it into a logic program $\pi_k(\mathcal{P})$ such that each answer set of $\pi_k(\mathcal{P})$ corresponds to a solution of at most k actions. Besides atoms defining the actions, fluents, literals, etc., $\pi_k(\mathcal{P})$ uses the following main predicates:

- $holds(L, T)$: literal L holds at step T .
- $poss(A, T)$: action A is executable at T .
- $occ(A, T)$: action A occurs at T .
- $pc(L, T)$: literal L may change at $T + 1$.
- $goal(T)$: the goal is satisfied at T .

The rules for encoding direct effects of actions in $\pi_k(\mathcal{P})$ are similar to the rules used for classical planning:

$$holds(L, T + 1) \leftarrow occ(A, T), causes(A, L, \varphi), holds(\varphi, T).$$

The difference with classical planning lies in the rules defining $pc(L, T)$ and the rule encoding of the inertial axiom:

$$holds(L, T + 1) \leftarrow holds(L, T), not pc(\neg L, T + 1).$$

Rules in $\pi_k(\mathcal{P})$ ensure that, for an answer set S of $\pi_k(\mathcal{P})$, if $\delta = \{l \mid holds(l, t) \in S\}$, $\delta' = \{l \mid holds(l, t + 1) \in S\}$, and $occ(a, t) \in S$ then (a) a is executable in δ ; and (b) $\delta' = \Phi_a(a, \delta)$.

As shown in Example 1, conformant plans are insufficient when sensing actions are needed. In this situation, conditional plans are required. Formally, a conditional plan is (i) \square is a conditional plan, denoting the empty plan, i.e., the plan containing no action; (ii) if a is a non-sensing action and p is a conditional plan then $[a; p]$ is a conditional plan; (iii) if a is a sensing action with proposition (6), where $\theta = \{g_1, \dots, g_n\}$, and p_j 's are conditional plans then $[a; \text{cases}(\{g_j \rightarrow p_j\}_{j=1}^n)]$ is a conditional plan; and (iv) Nothing else is a conditional plan.

To encode a conditional planning problem in ASP, we need to accommodate possible cases of a conditional plan. Let us observe that each conditional plan p corresponds to a labeled plan tree T_p defined as below.

- If $p = \square$ then T_p is a tree with a single node.
- If $p = [a]$, where a is a non-sensing action, then T_p is a tree with a single node and this node is labeled with a .
- If $p = [a; q]$, where a is a non-sensing action and q is a non-empty plan, then T_p is a tree whose root is labeled with a and has only one subtree which is T_q . Furthermore, the link between a and T_q 's root is labeled with an empty string.

- If $p = [a; \text{cases}(\{g_j \rightarrow p_j\}_{j=1}^n)]$, where a is a sensing action that determines g_j 's, then T_p is a tree whose root is labeled with a and has n subtrees $\{T_{p_j} \mid j \in \{1, \dots, n\}\}$. For each j , the link from a to the root of T_{p_j} is labeled with g_j .

For instance, Fig. 1 shows the trees for the following four conditional plans in the domain of Example 1:

- (i) $p_1 = [\text{push_down}; \text{flip_lock}]$;
- (ii) $p_2 = \text{check}; \text{cases} \left(\begin{array}{l} \text{open} \rightarrow [] \\ \text{closed} \rightarrow [\text{flip_lock}] \\ \text{locked} \rightarrow [] \end{array} \right)$;
- (iii) $p_3 = \text{check}; \text{cases} \left(\begin{array}{l} \text{open} \rightarrow [\text{push_down}; \text{flip_lock}] \\ \text{closed} \rightarrow [\text{flip_lock}; \text{flip_lock}; \text{flip_lock}] \\ \text{locked} \rightarrow [] \end{array} \right)$; and
- (iv) $p_4 = \text{check}; \text{cases} \left(\begin{array}{l} \text{open} \rightarrow [] \\ \text{closed} \rightarrow p_2 \\ \text{locked} \rightarrow [] \end{array} \right)$

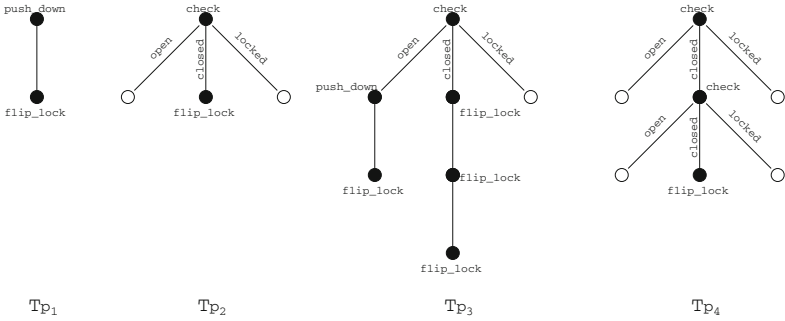


Fig. 1. Sample plan trees

Observe that each trajectory of the plan p corresponds to a path from the root to a leaf of T_p . Furthermore, let α (or the *width* of T_p) be the number of leaves of T_p and β (or the *height* of T_p) be the number of nodes along the longest path from the root to the leaves of T_p . Let w and h be two integers such that $\alpha \leq w$ and $\beta \leq h$ and the leaves of T_p be x_1, \dots, x_α . We map each node y of T_p to a pair of integers $n_y = (t_y, p_y)$, where t_y is the number of nodes along the path from the root to y , and p_y is defined in the following way.

- For each leaf x_i of T_p , p_{x_i} is an arbitrary integer between 1 and w such that (i) there exists a leaf x with $p_x = 1$, and (ii) $i \neq j$ implies $p_{x_i} \neq p_{x_j}$.
- For each interior node y of T_p with children y_1, \dots, y_r , $p_y = \min\{p_{y_1}, \dots, p_{y_r}\}$.

Figure 2 shows some possible mappings for the four trees in Fig. 1. It is easy to see that if $\alpha \leq w$ and $\beta \leq h$ then such a mapping always exists and (1, 1) is always assigned to the root. Furthermore, given a labeled tree T_p whose nodes are numbered according to the above rules, the plan p can easily be reconstructed. This means that

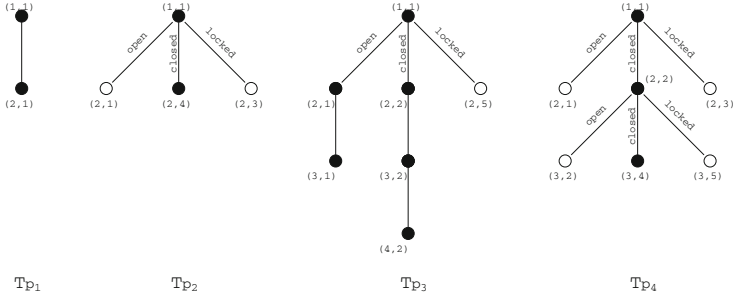


Fig. 2. Possible mappings for T_{p_i} ($i = 1, 2, 3, 4$) with $w = 5$ and $h = 4$

computing a solution p of a planing problem $\mathcal{P} = (\mathcal{D}, \mathcal{I}, \mathcal{G})$ is equivalent to identifying its tree T_p . This property allows for the encoding of \mathcal{P} as a logic program $\pi_{h,w}(\mathcal{P})$ that generates labeled trees corresponding to solutions of \mathcal{P} whose width and height are bounded by w and h , respectively. In addition to the usual predicates defining the actions, fluents, etc. $\pi_{h,w}(\mathcal{P})$ uses the following predicates:

- $holds(L, T, P)$: literal L holds at node (T, P) (i.e., at step T of path P).
- $poss(A, T, P)$: action A is executable at (T, P) .
- $occ(A, T, P)$: action A occurs at (T, P) .
- $pc(L, T, P)$: literal L may change at $(T + 1, P)$.
- $goal(T, P)$: the goal is satisfied at (T, P) .
- $br(G, T, P, P_1)$: there exists a branch from (T, P) to $(T + 1, P_1)$ labeled with G .
- $used(T, P)$: (T, P) belongs to some extended trajectory of the constructed plan.

Observe that most of the predicates used in $\pi_{h,w}(\mathcal{P})$ are similar to those in $\pi_k(\mathcal{P})$ extended with the third parameter encoding branches of a conditional plan, the last two predicates are specific to $\pi_{h,w}(\mathcal{P})$. They encode the cases of the solution. The detail encoding of $\pi_{h,w}(\mathcal{P})$ and its soundness and completeness can be found in [34].

One disadvantage of the proposed approach is the incompleteness of the ASP based planners. To address this issue, we identified completeness condition of the approximation [35]. Saturation and meta-programming techniques (see Sect. 3) could be used for a complete ASP-based planner.

3 Answer Set Programming in Goal Recognition Design

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 [9, 23]. For example, Fig. 3(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 [9, 23].

Goal recognition design (GRD) [12] aims at identifying possible changes to the 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

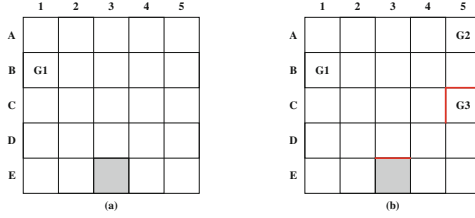


Fig. 3. Example Problem

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 $G3$. In [12], the authors 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. We will next present two ASP-based solutions of the GRD problem. Abusing the notation, we represent a GRD problem \mathcal{P} by the triple $(\mathcal{D}, \mathcal{I}, \mathcal{G})$ with the understanding that \mathcal{G} is the set of possible goals of the agent. By $wcd(\mathcal{P})$, we denote the wcd of \mathcal{P} .

3.1 A Saturation-Based Meta Encoding

The first encoding of the GRD problem in ASP utilizes meta-programming and saturation techniques. 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 [4]. It can be used to encode Σ_2^P -complete problems such as 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 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. It should be noted that the saturation technique puts syntactical restrictions on the program parts by forbidding the use of saturated atoms as default negation literals in a rule or as positive literals in a constraint [4, 15].

As it turns out, the wcd of a problem can be formulated as a $\exists\forall$ -QBF formula as follows. Let $g \in \mathcal{G}$ and π_g^* denote the minimal cost plan achieving g . Let $vl(x, y, c)$ denote that c is the common prefix of minimal cost plans of π_x^* and π_y^* . The wcd definition of \mathcal{P} can be encoded by the following $\exists\forall$ -QBF:

$$\exists x, y, c [vl(x, y, c) \wedge [\forall x', y', c' [vl(x', y', c') \rightarrow |c| \geq |c'|]]] \quad (7)$$

where, for the sake of simplicity, we omit some details such as $x, y, x', y' \in \mathcal{G}$, 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* using the saturation technique, we only need to encode the satisfiability of formula (7). Two possible implementations of the saturation techniques are detailed in [29]; one of them performs exceptionally well against the system developed by the inventor of the GRD problem.

3.2 A Multi-shot ASP Encoding

The second encoding of the GRD problem employs a hybrid implementation made possible by multi-shot ASP. Given a GRD $\mathcal{P} = (\mathcal{D}, \mathcal{I}, \mathcal{G})$, an integer k denoting the maximal number of actions that can be blocked for reducing the *wcd*, and an integer max denoting the maximal length of plans in \mathcal{P} . We develop a multi-shot ASP program $\Pi(\mathcal{P})$ for computing (i) $\mathbf{wcd}(\mathcal{P})$; and (ii) a solution of \mathcal{P} wrt. k (a set of actions that should be blocked) to achieve $\mathbf{wcd}(\mathcal{P})$. Specifically, $\Pi(\mathcal{P})$ implements Algorithm 1 in multi-shot ASP and consists of a logic program $\pi(\mathcal{P})$ and a Python program $GRD(\mathcal{P}, k, max)$.

Algorithm 1. $GRD(\mathcal{P}, k, max)$

```

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

```

The program $\pi(\mathcal{P})$ 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 $g \in \mathcal{G}$. This module is similar to the standard encoding in ASP planning [16] with an extension to allow for the generation of multiple plans for multiple goals at the same time (i.e., similar to that used in conditional planning in Sect. 2).

- *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 \mathcal{G} . It also contains the optimization statement for selecting answer sets containing $\text{wcd}(\mathcal{P})$.
- *Blocking*: A set of rules that interact with the Python program to block actions from the original problem.

The multi-shot ASP implementations of the GRD problem performs reasonably well against the system developed by the inventor of the GRD problem [29].

4 ASP in Multi-agent System

4.1 ASP and Distributed Constraint Optimization Problems

A *distributed constraint optimization problem* (DCOP) is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where: $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of *variables*; $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of finite *domains*, where D_i is the domain of variable x_i ; $\mathcal{F} = \{f_1, \dots, f_m\}$ is a set of *constraints*, where each k_i -ary constraint $f_i : D_{i_1} \times D_{i_2} \times \dots \times D_{i_{k_i}} \mapsto \mathbb{N} \cup \{-\infty, 0\}$ specifies the utility of each combination of values of the variables in its *scope*, $\text{scope}(f_i) = \{x_{i_1}, \dots, x_{i_{k_i}}\}$; $\mathcal{A} = \{a_1, \dots, a_p\}$ is a set of *agents*; and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent.

Figure 4(a,b) shows an example of a DCOP with three agents (see [14]), where each agent a_i controls variable x_i with domain $\{0, 1\}$. Figure 4(a) shows its constraint graph and Fig. 4(b) shows the utility functions, assuming that all of the three constraints have the same function.

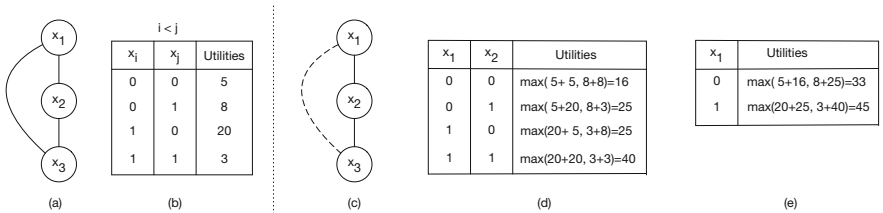


Fig. 4. DCOP graph (a), utility table (b); DPOP graph (c), UTIL-Phase Computation Table (d, e)

A *solution* is a value assignment for all variables and its corresponding utility is the evaluation of all utility functions on such solution. The goal is to find a utility-maximal solution. Solutions of a DCOP can be computed in three steps [22]: (i) constructing a pseudo-tree from the constraint graph (Fig. 4(c) for the example); (ii) UTIL-computation phase: each agent, starting from the leaf of the pseudo-tree, (ii.x) computes the optimal sum of utilities in its subtree for each value combination of variables in the set of variables owned by ancestor agents that are constrained with variables owned by the agents in the subtree (Fig. 4(d) shows the UTIL-computation of the agent a_3) and (ii.xx) sends the maximal value to its parent; and (iii) VALUE-propagation phase: each agent, starting from the root of the pseudo-tree, determines

the optimal value for its variables upon receiving the VALUE message from its parent and sends to its children (Fig. 4(e) shows the VALUE-propagation of the agent a_1) in a VALUE message.

In [14], we presented an ASP-based system, ASP-DPOP, for computing solutions of DCOP. In this system, each agent consists of two modules, an ASP module and a controller. The ASP module is responsible for computing the UTIL and VALUE messages when the agent needs to do so. The controller, written in SICStus[©] Prolog, is responsible for all communications between the agent and other agents. When an agent receives all UTIL messages from its children, the ASP module computes its UTIL-message and the controller sends the message to its parent. When an agent receives its parent’s VALUE message, the ASP module computes its own VALUE message and the controller sends the message to its children. The flexibility and expressiveness of ASP allows ASP-DPOP to work with agents who control multiple variables while state-of-the-art DCOP solvers assume that each agent controls only one variable. ASP-DPOP performs well against state-of-the-art DCOP solvers in several domains and has better performance, both in scalability and efficiency, in domains with hard constraints. The approach has been extended to deal with uncertainty in constraint utilities [13].

4.2 Multi-agent Planning

Multi-agent planning (MAP) is the problem of planning for multiple agents. The presence of multiple agents that can change the environment simultaneously brings about a number of issues:

- can the planning process be done centralized or must it be done distributed?
- what is the protocol for agents to communicate with each other?
- what types of actions are available for the agents (e.g., whether group actions are available? whether knowledge and/or belief changing actions are involved? etc.)?
- what are the representation languages used by individual agents?

For simplicity of the presentation, let us assume that all agents use the same representation language. The answer to the other questions depends on the degree of cooperativeness between agents.

Generally, a MAP for the agents $\{1, \dots, n\}$ can be represented by a tuple $(\mathcal{P}_1, \dots, \mathcal{P}_n)$ where \mathcal{P}_i is a planning problem for agent i extended with information about other agents who can affect the view of the environment locally to i .

- When agents are fully cooperative and planning can be done by one single agent, the encoding for single-agent planning (e.g., in [16]) can be extended to deal with MAP by
 - creating the program $\pi_k(\mathcal{P}_i)$ for \mathcal{P}_i ; and
 - adding constraints to eliminate conflicts that arise due to the (potentially) parallel execution of actions among agents

A prototype of an ASP based MAP system was proposed in [28]. Recently, we extend this prototype to deal with an interesting application the *Multi-Agent Path Finding* (MAPF) problem that deals with teams of agents that need to find collision-free paths from their respective starting locations to their respective goal locations on a graph. This model has attracted a lot of attention due to the success of the autonomous warehouse systems [36]. In these systems (illustrated by Fig. 5), robots (in orange) navigate around a warehouse to pick up inventory pods

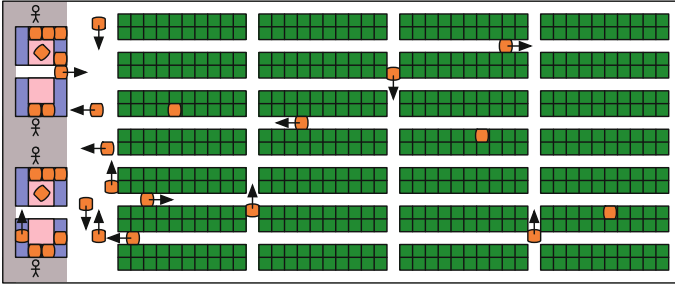


Fig. 5. Layout of an autonomous warehouse system [Wurman *et al.*, 2008] (Color figure online)

from their storage locations (in green) and drop them off at designated inventory stations (in purple) in the warehouse.

As it turns out, the ASP based system does not perform very well in this application comparing to state-of-the-art (e.g., [17]). The interesting part of this problem is that the basic ASP encoding is fairly simple. Yet, the problem quickly becomes *unsolvable* when its size increases.

By adding domain-knowledge to the encoding and decomposing the problem into smaller sub-problems, the scalability of the system improves significantly [19]. For example, it is easy to see that by adding some designated locations to the map, a path can be seen as multiple segments among the designated locations. As such, a path can be generated in multiple steps. In the first step, segments of a path are generated using a simplified map. The final path is then obtained by generating the concrete path for each segment.

- For self-interested agents, solving an MAP requires that agents negotiate with each other and thus an integration of a negotiation framework with MAP will be necessary. As shown in [27], ASP can also be used effectively for the development of negotiation systems. It is worth noticing that any negotiation framework used for this purpose must consider the dynamic of the environment. In [26], we developed an ASP based prototype for planning with negotiation in a dynamic environment. While the underlying encoding for planning does not change, special attentions need to be made to deal with the “effects” of negotiations. We envision that this approach will be necessary for some future extensions of the MAPF problem that might require stronger interactions between agents. For example, an agent might request help from another agent to continue its job if it realizes that its battery will run out before it can complete its job.

5 Conclusions

In this paper, we describe the application of answer set programming in planning with incomplete information and sensing actions, goal recognition design, distributed constraint optimization problem, and various settings of multi-agent planning. We discuss the key techniques that contribute to the good performance of ASP based solutions and present a challenging application for ASP.

Acknowledgement. The author wishes to thank his many collaborators and students for their contributions in the research reported in this paper. He would also like to acknowledge the partial support from various NSF grants.

References

1. Baral, C., Kreinovich, V., Trejo, R.: Computational complexity of planning and approximate planning in the presence of incompleteness. *Artif. Intell.* **122**, 241–267 (2000)
2. Baral, C., McIlraith, S., Son, T.C.: Formulating diagnostic problem solving using an action language with narratives and sensing. In: KR, pp. 311–322 (2000)
3. Dimopoulos, Y., Nebel, B., Koehler, J.: Encoding planning problems in non-monotonic logic programs. In: ECP, pp. 169–181 (1997)
4. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: a primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., Schmidt, R.A. (eds.) *Reasoning Web 2009*. LNCS, vol. 5689, pp. 40–110. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-03754-2_2](https://doi.org/10.1007/978-3-642-03754-2_2)
5. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR system dlw: progress report, comparisons, and benchmarks. In: KR 1998, pp. 406–417 (1998)
6. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. *AI Mag.* **37**(3), 53–68 (2016)
7. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) *JELIA 2004*. LNCS, vol. 3229, pp. 200–212. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-30227-8_19](https://doi.org/10.1007/978-3-540-30227-8_19)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp: a conflict-driven answer set solver*. In: Baral, C., Brewka, G., Schlipf, J. (eds.) *LPNMR 2007*. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007). doi:[10.1007/978-3-540-72200-7_23](https://doi.org/10.1007/978-3-540-72200-7_23)
9. Geffner, H., Bonet, B.: *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers (2013)
10. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: ICLP, pp. 579–597 (1990)
11. Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: *PDDL – the planning domain definition language, version 1.2*. Technical report, CVC TR98003/DCS TR1165, Yale Center for Comp, Vis and Ctrl (1998)
12. Keren, S., Gal, A., Karpas, E.: Goal recognition design. In: ICAPS (2014)
13. Le, T., Fioretto, F., Yeoh, W., Son, T.C., Pontelli, E.: ER-DCOPs: a framework for distributed constraint optimization with uncertainty in constraint utilities. In: AAMAS, pp. 606–614. ACM (2016)
14. Le, T., Son, T.C., Pontelli, E., Yeoh, W.: Solving distributed constraint optimization problems using logic programming. In: AAI, pp. 1174–1181. AAI Press (2015)
15. Leone, N., Rosati, R., Scarcello, F.: Enhancing answer set planning. In: *IJCA Workshop on Planning under Uncertainty and Incomplete Information* (2001)
16. Lifschitz, V.: Answer set programming and plan generation. *Artif. Intell.* **138**(1–2), 39–54 (2002)
17. Ma, H., Koenig, S.: Optimal target assignment and path finding for teams of agents, pp. 1144–1152 (2016)

18. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 375–398 (1999)
19. Nguyen, V.D., Obermeier, P., Son, T.C., Schaub, T., Yeoh, W.: Generalized target assignment and path finding using answer set programming. Technical report, NMSU (2017)
20. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* **25**(3,4), 241–273 (1999)
21. Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: Lifschitz, V., Niemelä, I. (eds.) *LPNMR 2004*. LNCS (LNAI), vol. 2923, pp. 207–219. Springer, Heidelberg (2003). doi:[10.1007/978-3-540-24609-1_19](https://doi.org/10.1007/978-3-540-24609-1_19)
22. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: *IJCAI*, pp. 1413–1420 (2005)
23. Ramírez, M., Geffner, H.: Goal recognition over pomdps: inferring the intention of a POMDP agent. In: *IJCAI*, pp. 2009–2014 (2011)
24. Simons, P., Niemelä, N., Soinen, T.: Extending and implementing the stable model semantics. *Artif. Intell.* **138**(1–2), 181–234 (2002)
25. Son, T.C., Pontelli, E.: A constructive semantic characterization of aggregates in answer set programming. *Theory Pract. Logic Program.* **7**(03), 355–375 (2007)
26. Son, T.C., Pontelli, E., Sakama, C.: Logic programming for multiagent planning with negotiation. In: Hill, P.M., Warren, D.S. (eds.) *ICLP 2009*. LNCS, vol. 5649, pp. 99–114. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-02846-5_13](https://doi.org/10.1007/978-3-642-02846-5_13)
27. Son, T.C., Pontelli, E., Nguyen, N., Sakama, C.: Formalizing negotiations using logic programming. *ACM Trans. Comput. Log.* **15**(2), 12 (2014)
28. Son, T.C., Pontelli, E., Nguyen, N.-H.: Planning for multiagent using ASP-prolog. In: Dix, J., Fisher, M., Novák, P. (eds.) *CLIMA 2009*. LNCS (LNAI), vol. 6214, pp. 1–21. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-16867-3_1](https://doi.org/10.1007/978-3-642-16867-3_1)
29. Son, T.C., Sabuncu, O., Schulz-Hanke, C., Schaub, T., Yeoh, W.: Solving goal recognition design using ASP. In: *AAAI* (2016)
30. Son, T.C., Tu, P.H., Gelfond, M., Morales, R.: An approximation of action theories of \mathcal{AL} and its application to conformant planning. In: *LPNMR*, pp. 172–184 (2005)
31. Son, T.C., Tu, P.H., Gelfond, M., Morales, R.: Conformant planning for domains with constraints – a new approach. In: *AAAI*, pp. 1211–1216 (2005)
32. Subrahmanian, V., Zaniolo, C.: Relating stable models and AI planning domains. In: *ICLP*, pp. 233–247 (1995)
33. Thiebaut, S., Hoffmann, J., Nebel, B.: In defense of PDDL axioms. In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)* (2003)
34. Tu, P., Son, T.C., Baral, C.: Reasoning and planning with sensing actions, incomplete information, and static causal laws using logic programming. *Theory Pract. Logic Program.* **7**, 1–74 (2006)
35. Tu, P., Son, T.C., Gelfond, M., Morales, R.: Approximation of action theories and its application to conformant planning. *Artif. Intell. J.* **175**(1), 79–119 (2011)
36. Wurman, P., D’Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Mag.* **29**(1), 9–20 (2008)