Answer Set Programs with One Incremental Variable

Junjun Deng and Yan Zhang

Abstract. In the past decade, Answer Set Programming (ASP) has emerged as a popular paradigm for declarative problem solving, and a number of answer set solvers have been developed. However, most existing solvers require variables occurring in a logic program to be bounded by some finite domains, which limits their applications in one way or another. In this paper, we introduce answer set programs with one incremental variable to overcome this limitation. Based on existing ASP solving techniques, an approach to solve answer set programs with one incremental variable is proposed, and a prototype solver is then developed based on this approach. By conducting some experiments, our approach is shown to be comparable to *iClingo*'s modular domain description approach in the incremental problem solving setting.

1 Introduction

Answer Set Programming (ASP) [1] is a declarative language for problem solving. Normally a problem is described in an ASP program containing bounded variables, and given an instance of that problem, an ASP grounder will output one propositional ASP program, which is handed to a propositional ASP solver to search for answer sets. This process is intuitive and straightforward for many classical search problems. However, some real-world applications may call for unbounded variables.

The Graph Coloring Problem (GCP) is a classical NP hard problem. Given an undirected graph, the goal is to find the least number of colors such that there exists at least one legal graph coloring scheme which assigns every vertex of the graph a color such that no two adjacent vertices have the same color.

Example 1. If the number of available colors k is provided, we can encode this problem into an ASP program such that each answer set of that program corresponds to a legal coloring scheme.

$$
\begin{aligned}
col(1..k).& \\
color(V,C) &\leftarrow node(V), col(C), not\ ncolor(V,C).\\
ncolor(V,C) &\leftarrow node(V), col(C), col(C1), color(V,C1), C \neq C1.\\
&\leftarrow edge(V1,V2), col(C), color(V1,C), color(V2,C).\ \square
\end{aligned}
$$

However, if k is unknown and to be minimized, then the domain of predicate *col* is unbounded, thus above program is not valid. Another example is the planning

problem, where we search for a sequence of actions to reach the goal state, and the length of that sequence is also not known precisely.

To solve the problems comprising a parameter which reflects its solution size, an incremental approach to both grounding and solving in ASP is proposed in [2]. This approach introduces a *(parametrized) domain description* as a triple (B, P, Q) of logic programs, among which P and Q contain a single parameter k ranging over the natural numbers: *a*) B is meant to describe static knowledge; *b*) $P[k]$ capture knowledge accumulating with increasing k; *c*) $Q[k]$ is specific for each value of k. The goal is to decide if the program $R[k] = B \cup \bigcup_{1 \leq i \leq k} P[i] \cup Q[i]$ has an answer set for some integer k [2].

Based on that domain description concept, an incremental ASP solver called *iClingo* [2] was developed, which can solve problems encoded in modular domain description. However, this incremental ASP approach requires programmers to split the whole encoding into three parts, and ensure that their domain description is modular. In addition, some intuitive encodings are excluded from modular domain description for some problems, e.g. GCP in Example 1.

Our goal is to develop an approach that is not only easy and intuitive for users to encode their problems, but also allows solvers to compute answer sets efficiently. In this paper, we propose a new approach for answer set programming, where programs are armed with one incremental variable. This incremental variable is ranging over the natural numbers, thus can model problems with unbounded variables. Based on the Clark's completion and loop formula [7], we propose a dynamic transformation for answer set programs with one incremental variable. With that we can bypass grounding to propositional rules, and directly construct dynamic transformation formula which is valid even when value of the incremental variable increases, and feed it to a propositional ASP solver repetitively until answer sets are found.

The rest of the paper is organized as follows. In Section 2 we define answer set programs with one incremental variable (ivASP). Then safe ivASP programs are introduced in Section 3. Section 4 presents the dynamic transformation of ivASP programs. Based on that transformation, we then develop a prototype solver for safe ivASP programs, and report its performance in Section 5. Finally we conclude this paper with related works in Section 6.

2 ASP with One Incremental Variable

In this section, we introduce answer set programs with one incremental variable (ivASP). The language of ivASP is defined over an alphabet including the following classes of symbols: (*a*) constants, (*b*) function symbols $\mathcal{F}$ including builtin $+$ and $-$; (*c*) variable symbols $\mathcal{V}$; (*d*) predicate symbols $\mathcal{P}$ including builtin $\leq, \geq$ and $=$; and (*e*) a special variable k ranged over natural numbers, which is called the *incremental variable*. A *term* is inductively defined as follows:

- A variable (including k) is a term.
- A constant is a term.

– If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ are terms.

A term is said to be *ground* if no variable occurs in it. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is a n-ary predicate symbol and each t_i is a term. An atom is ground if all t_i $(1 \leq i \leq n)$ in the atom is ground. A *literal* is either an atom or an atom preceded by the symbol "not".

A logic program is a finite set of rules of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_{m+1}, \ldots, not\ c_n$$

where a is an atom or $\bot$, and b_i, c_i are atoms.

A rule is *ground* if all of its atoms are ground, and a logic program is *ground* if every rule in it is ground. The head atom (a here) of a rule r is denoted as $head(r)$. $\{b_1, \ldots, b_m\}$ is called the positive body, denoted by $pos(r)$. Similarly, $\{not\ c_{m+1}, \ldots, not\ c_n\}$ is called the negative body of a rule, denoted by $neg(r)$, and each $not\ c_i$ is called *negative literal*. All atoms of a rule $atom(r) = head(r) \cup pos(r) \cup neg(r)$.

The *Herbrand Universe* of a language $\mathcal{L}$, denoted by $HU_{\mathcal{L}}$, is the set of all ground terms formed with the functions and constants (no variables) in $\mathcal{L}$. Similarly, the *Herbrand Base* of a language $\mathcal{L}$, denoted by $HB_{\mathcal{L}}$, is the set of all ground atoms formed with predicate $\mathcal{P}$ from $\mathcal{L}$ and terms from $HU_{\mathcal{L}}$.

Let r be a rule in the ivASP language $\mathcal{L}$. The grounding of r in $\mathcal{L}$ on level L where $L \in \mathbf{N}$, denoted by $ground(r, \mathcal{L}, L)$, is the set of all rules obtained from r by all possible substitutions of elements of $HU_{\mathcal{L}} \cup \{1..L\}$ for the variables in r except k, and the substitution of L for the special variable k. For any ivASP logic program Π, we define

$$ground(\Pi, \mathcal{L}, L) = \cup_{r \in \Pi} ground(r, \mathcal{L}, L) \tag{1}$$

and we use $ground(\Pi, L)$ as a shorthand for $ground(\Pi, \mathcal{L}(\Pi), L)$.

A *Herbrand interpretation* of a logic program Π is any subset of its Herbrand Base. A Herbrand interpretation I of Π is said to satisfy a ground rule, if

i) if $a \neq \bot$ then $\{b_1, \ldots, b_m\} \subseteq I \wedge \{c_{m+1}, \ldots, c_n\} \cap I = \emptyset$ implies that $a \in I$;
ii) otherwise $a = \bot$, $\{b_1, \ldots, b_m\} \not\subseteq I \vee \{c_{m+1}, \ldots, c_n\} \cap I \neq \emptyset$

A *Herbrand model* A of a logic program Π is a Herbrand interpretation I of Π such that it satisfies all rules in Π.

Definition 1. *Given a ground logic program Π, for any set S of atoms from Π, let Π^S be the program obtained from Π by deleting*

i) each rule that has a negative literal not b in its body with $b \in S$, and
ii) all negative literals in the bodies of the remaining rules.

The set S is a *stable model* (*answer set*) of Π if S is a minimal Herbrand model of Π^S [6].

Definition 2. *Given an ivASP program Π, a pair of a set of atoms $S \subseteq HB_{\mathcal{L}(\Pi)}$ and an integer L, (S, L) is a* stable model (answer set) *of Π if S is a stable model of ground(Π, L).*

Two ivASP programs are *equivalent* if their answer sets are the same.

Example 2. Following is an ivASP program:

$$p(1..k).$$
$$q(X+1) \leftarrow p(X), not\ p(X+1).$$

The pair $(\{p(1), q(2)\}, 1)$ and $(\{p(1), p(2), q(3)\}, 2)$ are two answer sets of it. □

3 Safe ivASP programs

Given an ivASP program Π, its grounding program may consist of infinite number of rules. To make the problem of finding answer sets feasible, we should consider ivASP programs that is equivalent to ground programs with finite rules. This calls for the definition of safe ivASP programs.

Definition 3. *Given a logic program Π, the* dependency graph $DG(\Pi) = (V_\Pi, E_\Pi)$ *of Π is a graph where each node corresponds to a predicate in Π and there is an edge from node p to node q iff there exists a rule in Π where p is the predicate symbol in head and q is a predicate symbol in the body.*

In a logic program Π, a predicate p *depends on* a predicate q iff there is a path from p to q in the dependency graph $DG(\Pi)$. A predicate p is an *extensional predicate* iff it does not depend on any predicate including itself.

Definition 4. *In an ivASP program Π, a predicate $p \in \Pi$ is a* domain predicate *iff it holds that every path in $DG(\Pi)$ starting from the node corresponding to p is cycle-free.*

Since the domain (extension) of a domain predicate can be computed without search, domain predicates are able to serve as the basis for an efficient grounding.

Given a variable X in a rule of a logic program, it is *range bounded* if there are at least a lower bound atom of the form $X \geq Y$ and an upper bound atom of the form $X \leq Z$ in the positive body where Y and Z are integer constants or range bounded variables.

Definition 5. *A rule r in an ivASP program is* safe *if it holds that: if the head predicate of r is an extensional predicate, then every variable except k occurs in the head is also range bounded; otherwise, every variable except k that occurs in the rule also appears in a positive domain predicate. An ivASP program is* safe *if every rule in it is safe.*

In practice, a rule of the form $p(a..k)$ is commonly used as a shorthand of $p(X) \leftarrow a \leq X, X \leq k$, thus is safe by definition. Example 1 is a safe ivASP program.

In Equation (1), the grounding of an ivASP program instantiates rules based on the Herbrand Universe. This naive grounding method may generate too many unnecessary rules in practice. Given a safe ivASP program Π and an integer L, a more concise ground program $grd(\Pi, L)$ which is equivalent to $ground(\Pi, L)$ can be constructed as follows:

1. Build the dependency graph $DG(\Pi)$, and identify a set of domain predicates $D(\Pi)$. Then predicates in $D(\Pi)$ are sorted topologically from bottom to top in $DG(\Pi)$, resulting an ordered set $D(\Pi) = (p_1, p_2, \dots p_i, \dots p_n)$ such that for any two predicates $p_i, p_j \in D(\Pi)$, if p_i depends on p_j then $i > j$.
2. For any predicate p that is a leaf node in $DG(\Pi)$ (extensional predicate), its ground instances are determined by fact atoms, or instantiated from range bound variables of the program. These ground atoms are added to $grd(\Pi, L)$, and the domain of this predicate, $Dom(p)$ is populated.

$$Dom(p) = \{\bar{c} | p(\bar{c}). \in \Pi\} \cup$$
$$\bigcup \{(c_1, \dots, c_n) | p(\bar{X}) \leftarrow \bigwedge_{X_i \in \bar{X}} lb \leq X_i, X_i \leq ub. \in \Pi, lb_i \leq c_i \leq ub_i (1 \leq i \leq n)\}$$

   where $\bar{c}$ is a tuple of constants, $\bar{X}$ is a tuple of variables and lb and ub are expressions which can be evaluated to integers.
3. For each domain predicate $d \in D(\Pi)$ which is not extensional, compute its domain $Dom(d)$ based on set operations for domains in Table 1.
4. Given the occurrence of a domain predicate of this form $d(X, Y, ...)$, let its variable binding $Bind(d) = \{(X/x, Y/y, ...) | (x, y, ...) \in Dom(d)\}$. For all rules $r \in \Pi$, the variable binding of r, B_r is the natural join of variable bindings of all positive domain predicates, $B_r = \bowtie_{d \in D(\Pi) \cap pos(r)} Bind(d)$. Then some elements in B_r are filtered out if they do not satisfy any relation test in the body of r. Finally instantiations of r is obtained through substitutions of variables to bindings in B_r, and $grd(\Pi, L) = grd(\Pi, L) \cup \{r/\theta | \theta \in B_r, r \in \Pi\}$.

Table 1. Set operations for domains

Rules	*Operations*
$p(X) \leftarrow q(X)$ $p(X) \leftarrow r(X)$	$Dom(p) = Dom(q) \cup Dom(r)$
$p(X) \leftarrow q(X), r(X)$	$Dom(p) = Dom(q) \cap Dom(r)$
$p(X) \leftarrow q(X), not\ r(X)$	$Dom(p) = Dom(q) \setminus Dom(r)$
$p(X, Y) \leftarrow q(X), r(Y)$	$Dom(p) = Dom(q) \times Dom(r)$
$p(X, Y, Z) \leftarrow q(X, Y), r(Y, Z)$	$Dom(p) = Dom(q) \bowtie Dom(r)$

Proposition 1. *Given a safe ivASP program and an integer $L \geq 0$, $grd(\Pi, L)$ is equivalent to $ground(\Pi, L)$.*

4 Dynamic Transformation

In the incremental problem solving setting, one of the most important techniques is computation reuse. To do this, we regard each logic program as a propositional

formula by employing Lin and Zhao's loop formulas [7]. Our main idea is to revise the completion formula and loop formula so that they are valid when the value of the incremental variable increases. To do this, we distinguish propositional clauses in the completion formula and loop formula that may be changed and annotate them with auxiliary propositional variables such that those clauses are satisfied whenever they become invalid.

To define the dynamic transformation, we first need some notations. Suppose Π is a safe ivASP program, a a ground atom, and L a natural number. We define (a) the supporting rules of an atom on level L, $sup_L(a,\Pi) = \{r \in grd(\Pi,L) \mid head(r) = a\}$; (b) the ground rules firstly instantiated on level L, $rule_L(\Pi) = grd(\Pi,L) \setminus \cup_{0\leq l<L} grd(\Pi,l)$; and (c) the atoms firstly instantiated on level L, $atom_L(\Pi) = atom(grd(\Pi,L)) \setminus \cup_{0\leq l<L} atom(grd(\Pi,l))$.

A ground rule r is *cumulative* wrt Π if for any two integers $1 \leq L_1 < L_2$ it holds that $r \in grd(\Pi,L_1)$ implies $r \in grd(\Pi,L_2)$. Let $cm_L(\Pi)$ be the set of rules in $rule_L(\Pi)$ which are cumulative wrt Π, and $ncm_{\leq L}(\Pi)$ be the set of rules $grd(\Pi,L) \setminus \cup_{0\leq l\leq L} cm_l(\Pi)$. A ground atom $a \in atom_L(\Pi)$ is *level restrained* if $sup_l(a,\Pi) = sup_L(a,\Pi)$ for all integers $l > L$. Given a natural number L, let $lr_L(\Pi)$ denotes the sets of all level restrained atoms in $atom_L(\Pi)$, $lr_{\leq L}(\Pi)$ be the set of atoms $\cup_{0\leq l\leq L} lr_l(\Pi)$, and $nlr_{\leq L}(\Pi)$ be the set of atoms $atom(grd\,(\Pi,L)) \setminus lr_{\leq L}(\Pi)$.

Definition 6 (Dynamic Completion Formula). *Given a safe ivASP program Π and an integer $L \geq 0$, the* dynamic completion formula *of Π on level L, denoted by $DCF(\Pi,L)$, is the conjunction of the static part*

$$\bigwedge_{r\in cm_L(\Pi)} \left[\widehat{body}(r) \supset head(r)\right] \wedge \bigwedge_{a\in lr_L(\Pi)} \left[a \supset \bigvee_{r\in sup_L(a,\Pi)} \widehat{body}(r)\right], \quad (2)$$

and the dynamic part

$$\bigwedge_{r\in ncm_{\leq L}(\Pi)} \left[\widehat{body}(r) \wedge k_L \supset head(r)\right] \wedge \bigwedge_{a\in nlr_{\leq L}(\Pi)} \left[a \wedge k_L \supset \bigvee_{r\in sup_L(a,\Pi)} \widehat{body}(r)\right], \quad (3)$$

where $\widehat{body}(r)$ denotes the conjunction of all literals in $body(r)$.

For each natural number L, an auxiliary variable k_L is introduced to annotate precedents of clauses in the dynamic part. When these clauses become invalid, the assignment of false to k_L make them satisfied. As we will see later, this technique is also applied to the dynamic loop formula.

Next, let us define the dynamic loop formula for ivASP programs. Suppose P is a ground logic program. The *positive dependency graph* of P is the directed graph whose vertices are atoms appearing in P and that consists of all edges from p to q such that p and q positively occur in the head and the body of a rule in P respectively. A nonempty set ℓ of atoms occurring in P is called a *loop* of P if for each pair of atoms a and b, there is a path from a to b in the positive dependency graph of P such that each vertex in the path belongs to ℓ.

Definition 7 (Dynamic Loop Formula). *Given an ivASP program Π and a natural number L, the dynamic loop formula, denoted by $DLF(\Pi, L)$, is the conjunction of the static part*

$$\bigwedge_{\ell \in \mathcal{L}_L(\Pi) \wedge \ell \subseteq lr_{\leq L}(\Pi)} \bigwedge_{a \in \ell} [a \supset ES(\ell, grd(\Pi, L))], \tag{4}$$

and the dynamic part

$$\bigwedge_{\ell \in loop_L(\Pi) \wedge \ell \not\subseteq lr_{\leq L}(\Pi)} \bigwedge_{a \in \ell} [a \wedge k_L \supset ES(\ell, grd(\Pi, L))], \tag{5}$$

where $loop_L(\Pi)$ is the set of all loops of $grd(\Pi, L)$, and $\mathcal{L}_L(\Pi) = loop_L(\Pi) \setminus \cup_{0 \leq l < L} loop_l(\Pi)$, $ES(\ell, grd(\Pi, L))$ is the formula $\vee_{a \in \ell} \vee_{r \in sup_L(a,\Pi) \wedge pos(r) \cap \ell = \emptyset} \widehat{body}(r)$.

Theorem 1. *Let Π be a safe ivASP program and L a natural number. Then a set A of ground atoms is an answer set of $grd(\Pi, L)$ iff $A \cup \{k_L\}$ is a model of*

$$\bigwedge_{0 \leq l \leq L} [DCF(\Pi, l) \wedge DLF(\Pi, l)]. \tag{6}$$

Proof. Let φ be the formula obtained from (6) by substituting $\bot$ for each k_l where $0 \leq l < L$ and by substituting $\top$ for k_L. Let $\varphi_1 = \bigwedge_{0 \leq l \leq L} DCF(\Pi, l)$ and $\varphi_2 = \bigwedge_{0 \leq l \leq L} DLF(\Pi, l)$ then $\varphi = \varphi_1 \wedge \varphi_2$.

$$\varphi_1 = \bigwedge_{0 \leq l \leq L} \bigwedge_{r \in cm_l(\Pi)} \left[\widehat{body}(r) \supset head(r)\right] \wedge \bigwedge_{0 \leq l \leq L} \bigwedge_{a \in lr_l(\Pi)} \left[a \supset \bigvee_{r \in sup_l(a,\Pi)} \widehat{body}(r)\right]$$
$$\wedge \bigwedge_{r \in ncm_{\leq L}(\Pi)} \left[\widehat{body}(r) \supset head(r)\right] \wedge \bigwedge_{a \in nlr_{\leq L}(\Pi)} \left[a \supset \bigvee_{r \in sup_L(a,\Pi)} \widehat{body}(r)\right],$$

Note that $grd(\Pi, L) = \bigcup_{0 \leq l \leq L} cm_l(\Pi) \cup ncm_{\leq L}(\Pi)$, and for a level restrained atom a, $sup_l(a, \Pi) = sup_L(a, \Pi)$ as $l < L$, thus conjuncts in φ_1 can be merged, resulting

$$\varphi_1 = \bigwedge_{r \in grd(\Pi, L)} \left[\widehat{body}(r) \supset head(r)\right] \wedge \bigwedge_{a \in atom(grd(\Pi, L))} \left[a \supset \bigvee_{r \in sup_L(a,\Pi)} \widehat{body}(r)\right],$$

which is exactly the Clark's completion of $grd(\Pi, L)$. In addition,

$$\varphi_2 = \bigwedge_{0 \leq l \leq L} \bigwedge_{\ell \in \mathcal{L}_l(\Pi) \wedge \ell \subseteq lr_{\leq l}(\Pi)} \bigwedge_{a \in \ell} [a \supset ES(\ell, grd(\Pi, l))]$$
$$\wedge \bigwedge_{\ell \in loop_L(\Pi) \wedge \ell \not\subseteq lr_{\leq L}(\Pi)} \bigwedge_{a \in \ell} [a \supset ES(\ell, grd(\Pi, L))],$$

Algorithm 1 ivASPSolve

Require: A safe ivASP program Π.
Ensure: Returns an answer set of Π or "no answer set".

```
1: l ← 1
2: Δ ← ∅
3: while l ≤ UB do
4:     P ← Ground(Π, l)
5:     Δ ← Δ ∧ DCF(Π, l)
6:     (res, Ans, Δ′) ← ASPSolve(P, Δ, k_l)
7:     Δ ← Δ′
8:     if res = true then
9:         return Ans
10:    else
11:        l ← l + 1
12:    end if
13: end while
14: return no answer set
```

For level restrained atoms, once they are grounded on a level, their supporting rules do not change on higher levels, so for a loop ℓ whose elements are all level restrained atoms, its external support does not change as well. Therefore $ES(\ell, grd(\Pi, l)) = ES(\ell, grd(\Pi, L))$.

$$\varphi_2 = \bigwedge_{\ell \in loop_L(\Pi)} \bigwedge_{a \in \ell} \left[a \supset ES(\ell, grd(\Pi, L)) \right],$$

which equals to the loop formula of $grd(\Pi, L)$. By Theorem 1 of [7], we then obtain the desired result. □

5 Implementation and Experiment

Theorem 1 provides an approach to solve ivASP programs, as we can determine whether a set of ground atoms is the answer set by checking if it satisfies Equation 6.

Algorithm 1 shows the high level structure of our prototype solver called *ivASP*. After initialization, it repetitively grounds the program, computes the dynamic completion formulas, and then call ASPSolve to do actual searching, until answer sets are found or global configured upper bound UB is reached. Ground is a grounding procedure that outputs rules without variables, and marks each atom as level restrained or not and each rule as cumulative or not. Then dynamic completion formulas can be constructed. Δ is the constraint (clause) array, and stores all the constraints transformed from the input program and constraints learnt during search.

ASPSolve is a conflict-driven constraint learning ASP solving procedure, based on Algorithm 1 in [4]. The differences are: (*a*) the last argument k_l is an assumption that forces k_l be in the answer set. (*b*) for any detected unfounded

set, its dynamic loop formula (instead of normal loop formula) is added to Δ; and (c) if the input ground logic program has a answer set Ans, then it returns $(true, Ans, \Delta)$; otherwise it return $(false, \emptyset, \Delta)$. Each call of ASPSolve would probably add more learnt constraints to Δ, thus constraints learnt from current levels can be utilized on later levels.

To evaluate the performance of our proposed approach to ivASP solving, we compare *ivASP* to *iClingo* in some benchmarks. The first series of instances are from the GCP problem. The encoding of GCP for *ivASP* is like Example 1 with the addition of two meta-statement declaring k as the incremental variable and atoms instantiated from *ncolor* as non-level restrained. By treating GCP as a finite model computation problem, we derive its domain description [5], which is the encoding of GCP for *iClingo*. For the Towers of Hanoi problem, we use the encoding and benchmarks from [2]. The rest of benchmarks are from the FNT (First-order form syntactically non-propositional Non-Theorem) division of the CASC-23 competition, also requiring *iclingo* to solve in at least 2 steps and more than 0.1 second. Similar to *fmc2iasp*, which compiles finite model computation (FMC) problems to *iClingo* programs, we also implement a tool that can convert FMC problems to ivASP programs.

The experiments are all done in a linux desktop with an Intel Core i7-3520M CPU and 4 GB memory. In Table 2, total times of *ivASP* and *iClingo* running each selected instances are listed, with "–" denoting timeout in 500 seconds. The second column n denotes the least value of the incremental variable when answer sets are found.

As a conclusion, when a problem is natural to express by an ivASP program, our approach has advantages in both the language and efficiency, as shown in the GCP instances. In general, the performance of *ivASP* in these benchmarks is comparable to that of *iClingo*, though there indeed some instances *iClingo* shows over performance to *ivASP*. Nevertheless, we need to emphasize that this is mainly due to the shortage of optimization of ivASP at this stage, that we will seriously take into account in our next research.

6 Conclusion and Related Works

In this paper, an incremental variable was introduced to ASP language for incremental problem solving, and a dynamic transformation, which is based on the techniques of loop formulas [7] and conflict-driven learning [3], was then proposed to accelerate the solving of answer sets for the new language. With this transformation, a prototype system, *ivASP*, was developed, and the effectiveness of our system was demonstrated by experiments on some benchmarks.

Another system, *iClingo* [2], was also developed for incremental answer set solving. However, the problems to be solved in this system are required being encoded in three parts, which challenges the programmer on both domain knowledge and programming techniques. Instead our system only requires the problems being encoded in a natural way. Surprisingly, our system is still comparable to their system with regard to computation efficiency.

Table 2. Running time of *ivASP* and *iClingo* in selected benchmarks

instance	n	*ivASP*	*iClingo*
1_FullIns_3	4	0.010	0.000
1_Insertions_4	5	38.270	51.760
2_FullIns_3	5	0.010	0.000
2_Insertions_3	4	0.020	0.010
3_Insertions_3	4	0.170	0.140
queen5_5	5	0.010	0.000
queen6_6	7	1.810	2.460
queen7_7	7	0.060	0.050
queen8_8	9	–	–
queen8_12	12	97.020	183.450
queen9_9	10	–	–
	33	36.810	8.350
	34	27.460	9.450
Towers of Hanoi	36	112.750	21.200
	39	213.750	52.030
	41	393.200	99.070
LCL651+1.010	2	0.090	0.120
LCL651+1.020	2	0.360	0.370
LCL653+1.005	2	0.110	0.090
LCL653+1.015	2	0.310	0.340
LCL655+1.010	2	5.790	3.460
LCL659+1.015	2	270.580	59.690
LCL661+1.001	2	0.140	0.120

instance	n	*ivASP*	*iClingo*
LCL661+1.015	2	0.150	0.170
LCL669+1.015	2	0.180	0.260
LCL671+1.015	2	2.520	2.780
LCL677+1.015	2	0.160	0.200
LCL689+1.010	2	0.150	0.140
LCL689+1.020	2	0.270	0.340
NLP160+1	2	1.360	1.260
NLP161+1	2	2.670	1.370
NLP162+1	2	1.310	1.260
NLP164+1	2	2.090	2.300
NLP165+1	2	2.040	2.370
NLP190+1	2	46.870	22.330
NLP192+1	2	13.360	13.430
NLP193+1	2	21.100	21.960
NLP194+1	2	13.800	13.980
NLP195+1	2	13.220	13.080
NLP196+1	2	20.830	24.880
NLP211+1	7	1.450	0.540
NLP212+1	7	1.480	0.570
NLP213+1	7	1.440	0.560
SYN330+1	8	0.360	0.230
SYN335+1	11	106.950	59.930
SYN519+1	3	0.080	0.170

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York, NY, USA (2003)
2. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental asp solver. In: Logic Programming, pp. 190–205. Springer (2008)
3. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of the 20th international joint conference on Artifical intelligence. pp. 386–392 (2007)
4. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence (2012)
5. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. AI Commun. 24(2), 195–212 (Apr 2011)
6. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference on Logic programming. vol. 161 (1988)
7. Lin, F., Zhao, Y.: Assat: Computing answer sets of a logic program by sat solvers. Artificial Intelligence 157(1), 115–137 (2004)