ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.4, April 2008, pp.869−878
DOI: 10.3724/SP.J.1001.2008.00869
© 2008 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一个带破圈启发方法的回答集编程系统*

沈榆平+, 赵希顺

(中山大学 逻辑与认知研究所,广东 广州 510275)

An Answer Set Programming System with Cycle Breaking Heuristic

SHEN Yu-Ping+, ZHAO Xi-Shun

(Institute of Logic and Cognition, Sun Yat-sen University, Guangzhou 510275, China)

+ Corresponding author: Phn: +86-20-84114036, Fax: +86-20-84110298, E-mail: exp04shy@mail2.sysu.edu.cn

Shen YP, Zhao XS. An answer set programming system with cycle breaking heuristic. *Journal of Software*, 2008,19(4):869−878. http://www.jos.org.cn/1000-9825/19/869.htm

Abstract: Answer set programming (ASP) is a logic programming paradigm under answer set semantics, which can be utilized in the field of non-monotonic reasoning and declarative problem solving, etc. This paper proposes and implements a cycle breaking heuristic and a bottom-restricted look-ahead procedure for ASP, and the resulting system is called LPS. The experimental results show that, relative to other state-of-the-art ASP systems, LPS could efficiently solve logic programs in phase transition hard-job-regions, and these programs are generally considered difficult to compute. In addition, by applying the so-called dynamic variable filtering (DVF) technique, LPS could greatly reduce the search tree size during the computation.
Key words: answer set programming; heuristic; look-ahead; logic program; phase transition

摘 要: 回答集编程(answer set programming,ASP)是一种回答集语义下的逻辑编程范例,可应用于非单调推理,叙述式问题求解等领域.本文为ASP提出并实现了一种破圈启发方法与一种基部限制式前向搜索过程,所得到的系统称为LPS.实验结果显示,相对于其他经典的ASP系统,LPS能够有效地解决处于相变难区域中的逻辑程序,通常这些程序被认为是计算困难的.除此以外,通过使用被称为动态变元过滤(dynamic variable filtering,DVF)的技术,LPS可以在计算过程中极大地缩小搜索树的尺寸.
关键词: 回答集编程;启发方法;前向搜索;逻辑程序;相变
中图法分类号: TP18 文献标识码: A

1 Introduction

Answer set programming (ASP) is a logic programming paradigm under answer set semantics (also called stable model semantics)[1,2]. In ASP, problems are encoded as logic programs, the corresponding answer sets of the logic programs give solutions to the original problems. Many applications like non-monotonic reasoning, reasoning

* Supported by the National Natural Science Foundation of China under Grant Nos.60573011, 10410638 (国家自然科学基金); the MOE Project of China under Grant No.05JJD72040122 (国家教育部基地重大招标项目)

Received 2006-10-31; Accepted 2007-03-08

© 中国科学院软件研究所 http://www.jos.org.cn

about actions, declarative problem solving[3], can be expressed by logic programs, and then solved by an ASP system.

Unfortunately, checking whether a logic program has an answer set is NP-complete[4]. It follows that computing answer sets for logic programs is a computational hard task. In the last decade, much work has been devoted to the implementation of ASP, most of these systems like Smodels, DLV[5,6] are based on backtrack search algorithms, which essentially construct a binary search tree and have to handle exponential search space. For such algorithms, heuristics are crucial, good heuristics can significantly improve system performance.

In this paper we propose so-called *cycle breaking* heuristic for ASP, the idea is motivated by Ref.[7], where new upper bounds for computing answer sets are obtained through analyzing the number of *cycles* of a logic program. In particular, Ref.[7] suggests that a good heuristic should break the most number of *even* cycles. As we shall see later, our heuristic matches the purpose quite good. Another important feature of ASP systems is the use of look-ahead procedures[5,6], the basic idea is to discover inconsistency before selecting a new branching node, and then backtrack as soon as possible, avoiding falling into a deep dead end. It has been shown that look-ahead can greatly speed up the computation, however, it often consumes most of the running time of a system, many restricted look-ahead procedures are then proposed to reduce the running cost. In this paper we present a look-ahead procedure for ASP, which restricts the atoms being looked on *bottoms*. Intuitively, bottoms are the “root” of the dependency graph of a logic program, the truth values of the atoms outside bottoms depend on those inside, therefore restricting the looking over bottoms is considered to be a good idea.

The rest of the paper is organized as follows. Section 2 provides some basic concepts of ASP. Section 3 presents the basic algorithm for computing an answer set. Sections 4 and 5 describe the cycle breaking heuristic and the *bottom-restricted* look-ahead procedure. In Section 6, the experimental results are presented and some related work is briefly discussed in Section 7, we draw conclusions and show future directions in Section 8.

2 Preliminaries

A *logic program* is a finite set of rules of the form: $a \leftarrow b_1,\ldots,b_m,$ *not* $c_1,\ldots,$*not* c_n, where a, b_i's, c_j's are atoms. We call a the *head* of the rule and the other atoms the *body* of the rule. For convenience, let $Head(r)=a$, $Pos(r)=\{b_1,\ldots,b_m\}$ and $Neg(r)=\{c_1,\ldots,c_n\}$. A *literal* is an atom or an atom preceded by *not*. The former is called a positive literal and the latter is called a negative literal. If $Neg(r)=\varnothing$ then r is a *definite* rule. A *definite* logic program P is a set of definite rules, by $Cn(P)$ we refer to its *unique minimal closure*. By $At(P)$ we denote the set of atoms occurring in a logic program P. The *Gelfond-Lifschitz reduct* P^A of P with respect to a set $A \subseteq At(P)$ is obtained by deleting each rule $r \in P$ such that $Neg(r) \cap A \neq \varnothing$, and removing all negative literals in the remaining rules. Note that a Gelfond-Lifschitz reduct must be a definite program. A set of atoms $A \subseteq At(P)$ is said to be an *answer set*[1] of P iff $A=Cn(P^A)$. The well-founded model[9] of a logic program P is a unique ordered pair $I(P)=\langle I(P)^+,I(P)^-\rangle$ in which $I(P)^+$ contains atoms which must be *true* and $I(P)^-$ contains atoms which must be *false*. The truth values of atoms which are neither in $I(P)^+$ nor in $I(P)^-$ are *uncertain*. Well-founded model can be characterized by operator γ_P, which is defined as $\gamma_P(A)=Cn(P^A)$. Note that $A_1 \subseteq A_2$ implies $\gamma_P(A_2) \subseteq \gamma_P(A_1)$, i.e. γ_P is *anti-monotone*. It follows that $\gamma_P^2(A)=\gamma_P(\gamma_P(A))$ is *monotone*, the well-founded model of a program P can be represented as $\langle lfp(\gamma_P^2), gfp(\gamma_P^2)-At(P)\rangle$ in which $lfp(\gamma_P^2)$ is the least fixed-point of γ_P^2 and $gfp(\gamma_P^2)$ the greatest fixed-point of γ_P^2 [10]. Consider a logic program $P_1=\{c \leftarrow not\ b, b \leftarrow not\ c, a \leftarrow\}$, the answer sets of it are $\{a,c\}$ and $\{a,b\}$ while its well-founded model is $\langle\{a\},\varnothing\rangle$. A *simplification*[11] of a logic program P under its well-founded model $I(P)$, denoted by $P\backslash I(P)$, is obtained by deleting each rule $r \in P$ such that either $Head(r) \in I(P)^+$ or $Pos(r) \cap I(P)^- \neq \varnothing$ or $Neg(r) \cap I(P)^+ \neq \varnothing$, and remove all positive literals a where $a \in I(P)^+$ and all negative literals *not* a in which $a \in I(P)^-$ from the bodies of the

© 中国科学院软件研究所 http://www.jos.org.cn

remaining rules.

A *dependency graph* of a logic program P, denoted by $G(P)$, is a directed graph $\langle V,E\rangle$ with labeled edges. Let $V=At(P)$, there is a *positive* (*negative*, respectively) edge from node p to node q if there exists a rule $r\in P$, $Head(r)=q$ and $p\in Pos(r)$ ($p\in Neg(r)$, respectively). A *strongly connected component* of a directed graph $\langle V,E\rangle$ is a set of nodes $V'\subseteq V$ such that for any $u,v\in V'$, there is a path from u to v, and V' is not a proper subset of any such sets. A *bottom*[7] is a strongly connected component S of a directed graph, and there is no other strongly connected component S' such that there is a path from S' to S. Note that a strongly connected component can be computed in linear time[12]. An *odd* (*even*, respectively) *cycle* of a dependency graph is a simple cycle containing an odd (non-zero even, respectively) number of *negative* edges. Both odd and even cycles are called *negative cycles*, simple cycles containing no negative edges are called *positive cycles*.

Odd cycles and even cycles are of great interest in investigating logic programs. It has been shown that a logic program which has no odd cycles[13] (called *call-consistent* program) at least has one answer set, and a logic program that has no even cycles[14] has at most one answer set. Furthermore, in the latter case, if the well-founded model of the logic program is also its answer set, then it is the only one, otherwise, it has no answer sets[15]. Informally speaking, odd cycles eliminate answer sets while even cycles generate answer sets. Based on these results, an algorithm for computing answer sets concerning cycles is proposed in Ref.[7], in next section we shall describe a slightly modified version.

3 Basic Algorithm for Computing Answer Sets

A *signed atom* a^* is an atom a with sign $*\in\{+,-\}$. Define $\neg\neg\neg a^-=a^+$ and $\neg a^+=a^-$. Let σ be a *finite consistent set* of signed atoms, by consistence we mean there is no atom a such that both a^+, a^- are in σ. Define $c^+=\{b^+|b^+\in\sigma\}$, similarly, $\sigma^-=\{b^-/b^-\in\sigma\}$. For a set of signed atoms $\varnothing$, let $|\varnothing|=\{b|b^*\in\varnothing\}$.

Definition 1[11]. Let P be a logic program, and σ a finite consistent set of signed atoms with $|\varnothing|\subseteq At(P)$, let P_σ be the program obtained from P by deleting all rules $r\in P$ which $Pos(r)\cap|\sigma^-|\neq\varnothing$ or $Neg(r)\cap|\sigma^+|\neq\varnothing$, and removing positive literal a from the bodies of the remaining rules if $a\in|\sigma^+|$ or negative literal *not* a if $a\in|\sigma^-|$.

For instance, let P be P_1 mentioned in previous section, $\sigma=\{b^+,c^-\}$, we have $P_\sigma=\{b\leftarrow,a\leftarrow\}$. Intuitively, $|\sigma^+|$($|\sigma^-|$, respectively) stands for atoms assumed to be true (false, respectively). Furthermore, consider the dependency graph $G(P_\sigma)$, it is a subgraph of $G(P)$ and there are no edges going out from atoms in $|\sigma|$, in some sense they break the outgoing edges, we call these atoms *breaking nodes*.

Proposition 2[11]. Let P be a logic program, and σ a finite consistent set of signed atoms with $|\sigma|\subseteq At(P)$. Suppose S is an answer set of P, $|\sigma^+|\subseteq S$ and $|\sigma^-|\cap S=\varnothing$, then S is an answer of P_σ.

Proposition 2 implies that each answer set of P is an answer set of either $P_{\{a^+\}}$ or $P_{\{a^-\}}$ where $a\in At(P)$, however, the inverse generally does not hold. The proposition below shows an interesting property about bottoms.

Proposition 3[7]. Given a logic program P and its well-founded model $I(P)$, if $Q=P\backslash I(P)$ is not empty, then every bottom of Q must have a pair of nodes a and b, such that there is a negative edge from a to b.

Proposition 3 shows the existence of a negative cycle in each bottom of the well-founded simplification of a program, which plays an important role in the proof of theorem 4. Having introduced the above definitions and propositions, we present our basic algorithm $Ans(P)$ in Fig.1, where $ComputeAns(P_\sigma)$ performs binary search, $Bottoms(Q)$ returns the set of all bottoms of logic program Q and $I(P_\sigma)$ returns the well-founded model of P_σ. The soundness of the algorithm is guaranteed by the theorem below:

Theorem 4[7]. A logic program P has an answer set if and only if $Ans(P)$ returns True.

Proof: (Sketched) $\Rightarrow$. By induction on k, the number of odd and even cycles in $G(P)$. For $k=0$, it is well-known

© 中国科学院软件研究所 http://www.jos.org.cn

P has an unique answer set $I(P)^+$. For $k>0$, trivial if $I(P)^+$ is the answer set of P. Suppose $I(P)^+$ is not an answer set for P, pick a breaking node a from $\cup Bottoms(Q)$ on a negative cycle (such a node always exists). Note that an answer set of P is either an answer set of $P_{\{a^+\}}$ or $P_{\{a^-\}}$ and both of them have less negative cycles than k, by induction assumption, $ComputeAns(P_{\{a^+\}})$ ($ComputeAns(P_{\{a^-\}})$, respectively) returns true if $P_{\{a^+\}}$ ($P_{\{a^-\}}$, respectively) has an answer set which is also an answer set of $P_\varnothing$ (i.e. P). Therefore if P has an answer set then $Ans(P)$ returns true. $\Leftarrow$. Trivial. □

It has been shown in Ref.[7] that if a program P has at most k even cycles, then P has at most 2^k answer sets, and can be computed in $2^{2k}O(n^k)$ time, where n is the size of the given program. Moreover, restricting the number of odd cycles cannot reduce the complexity. These facts imply that $ComputeAns(P_\sigma)$ should break the most number of even cycles when selecting a breaking node, such that the two resulting logic programs have less even cycles and then easier to compute. In $ComputeAns(P_\sigma)$, we pick breaking nodes from bottoms which at least break one negative cycle, unfortunately, whether it breaks an even cycle is an NP-complete problem[7], we shall propose a heuristic for this problem in next section.

Procedure *Ans*(P)
Input: A logic program P;
Output: True if P has an answer set otherwise False.
1. **return** $ComputeAns(P_\varnothing)$;

Procedure $ComputeAns(P_\sigma)$
Input: A logic program P_σ;
Output: True if P_σ has an answer set that is also an answer set of $P_\varnothing$ otherwise False.
1. **if** $I(P_\sigma)^+$ is an answer set of P_σ **then**
2. **if** $I(P_\sigma)^+$ is an answer of $P_\varnothing$ **then return** True **else return** False
3. **end if**
4. $Q:=P_\sigma\backslash I(P_\sigma)$;
5. Pick $a\in\cup Bottoms(Q)$ on a negative cycle;
6. **if** $ComputeAns(P_{\sigma\cup\{a^+\}})$ **then return** True **else return** $ComputeAns(P_{\sigma\cup\{a^-\}})$

Fig.1 Basic algorithm for computing an answer set

4 Cycle Breaking Heuristic

From the above sections, we know that choosing a node that breaks many even cycles can significantly speed up the computation. However, it is very difficult to pick such a node since even deciding whether it breaks one even cycle is NP-complete. A compromise is to prefer breaking negative cycles, no matter which kind of cycles (odd or even) they are. In addition, observe that in a directed graph, a node that has more degrees (occurrences) would probably break more cycles, similar idea has been adopted by the famous MOMS heuristic[16,17], which prefers picking atoms in shorter open clauses. Based on the above motivations, we propose our heuristic as follows.

Define the *length* of a rule r to be the number of *unassigned* literals in its body with respect to well-founded semantics, denoted by $L(r)$. Let r be a rule containing some unassigned atoms in program P (i.e. open rules), for each unassigned atom $a\in At(P)$ we define $w_1(a)=\sum_{head(r)=a}\alpha^{-L(r)}$, $w_2(a)=\sum_{a\in Pos(r)}\alpha^{-L(r)}$, $w_3(a)=\sum_{a\in Neg(r)}\alpha^{-L(r)}$.

These functions calculate the weights contributed by different occurrences of atom a, more precisely, they calculate the weights of a when it is a head, positive literal and negative literal respectively. The value of α is empirically set to 5, means that every 5 occurrences of an atom in a length $i+1$ rule are counted as 1 occurrence in rule which has length i, this value is obtained by a large number of experimenting, and adopted by many researchers[16−18]. The evaluation function $w(a)$ is defined as: $w(a)=w_1(a)+w_2(a)+\beta\cdot w_3(a)$ where β is an empirically

© 中国科学院软件研究所 http://www.jos.org.cn

good value used to emphasize negative edges since we prefer breaking negative cycles, this value could be modified when dealing different logic programs. In this paper we use a fixed value β=1.3. LPS selects a breaking node a such that $w(a)$ is the greatest, moreover, if $w_2(a)>w_3(a)$ then it first branches a, otherwise it first branches *not* a. Alternating branching order is employed by most ASP systems and has been proved quite useful in some instances.

This function is applied on *bottoms*, since picking a node from a bottom at least breaks one cycle (since bottoms are strongly connected components), furthermore, since there exists at least one negative cycle in each bottom of the well-founded reduction of a program, so the heuristic is expected to efficiently break negative cycles and therefore greatly reduce the depth of the search tree.

For example, let $P_2=\{a\leftarrow not\ b, b\leftarrow not\ a, c\leftarrow a, c\leftarrow b\}$, its dependency graph is shown in Fig.2, it is easy to see it has one bottom $\{a,b\}$. In the binary search if we first pick c then the truth values of a,b are still not determined, however, if we first choose an atom from the bottom $\{a,b\}$, then an answer set is immediately derived. Note that in the latter case, the only even cycle of the program is broke when we choose a or b according to our heuristic.

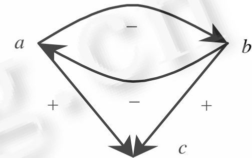

Fig.2 Dependency graph of P_2

5 Bottom-Restricted Look-Aheads

The essential of look-ahead[5,6] is to discover a dead end as early as possible, thus avoid large scale backtracking. Simply speaking, before choosing a new node in the binary search, look-ahead procedure assumes *true* and then *false* for an *unassigned* atom and performs consistency checking, if both checkings are inconsistent, the search backtracks since there is a contradiction, else if exactly one of the checking is consistent then it picks the atom with the consistent value, otherwise the procedure looks at next unassigned atom. As mentioned before, look-ahead procedure is computational expensive, since for each unassigned atom the consistency checking is performed twice. In LPS look-ahead is applied on bottoms instead of all unassigned atoms, thus called *bottom-restricted* look-ahead. Intuitively, bottoms are the “root” of the dependency graph of a program, the truth values of the atoms outside bottoms depend on those inside. A large program may have small bottoms, therefore looking on bottoms is believed strong enough to discover many contradictions in advance.

We implemented two versions of bottom-restricted look-ahead in LPS: one adopts so-called *static variable filtering* (SVF) while another one adopts *dynamic variable filtering* (DVF). These two techniques are originally proposed in SAT[19], where they appear to have different performances. Roughly speaking, look-ahead with SVF at most looks every atom once and then return to the binary search, however, if an atom is picked during the looking, then the current partial model is changed and new information about the contradiction may be inferred, so look-ahead with DVF looks unassigned atoms repeatedly until no contradictions could be discovered.

Bottom-restricted look-ahead with DVF is shown in Fig.3 and the SVF version could be obtained by removing the **Do-Until** loop. The function $Conflict(P,\langle|\sigma^+|,|\sigma^-|\rangle)$ in Fig.2 is from Ref.[5], it returns true when P has no answer set S such that $|\sigma^+|\subseteq S$ and $|\sigma^-|\subseteq At(P)\backslash S$, the mechanism behind it is to expand the current partial model $\langle|\sigma^+|,|\sigma^-|\rangle$ by some sophisticated inference rules and see whether $|\sigma^+|\cap|\sigma^-|\neq\varnothing$, if so then the current partial model is consistent, details about this function could be found in Ref.[5]. In Fig.4, we present the improved basic algorithm for LPS, which integrates the bottom-restricted look-ahead and the cycle breaking heuristic.

To see how look-ahead procedure speeds up the computing, consider a program $P_3=\{a_1\leftarrow not\ b_1, b_1\leftarrow not\ a_1,\ldots a_n\leftarrow not\ b_n, b_n\leftarrow not\ a_n, c\leftarrow not\ c\}$. P_3 has no answer sets and n+1 bottoms (Fig.5), without look-ahead procedure, in the worst case the algorithm may has to go through 2^n partial models before it is sure that there is no

© 中国科学院软件研究所 http://www.jos.org.cn

answer sets, however, if we enable look-ahead procedure, the algorithm can return false at one step since conflict checking is performed on each bottom thus the rule $c\leftarrow not\ c$ is discovered to be unsatisfiable for any partial model. In this extreme example, the look-ahead procedure saves exponential time.

Procedure *Look-ahead*(P_σ)
Input: A logic program P_σ;
Output: A logic program $P_{\sigma'}$ if no dead end is discovered otherwise False.

1. **do**
2. $\sigma' := \sigma$; $Q := P_\sigma \backslash I(P_\sigma)$;
3. **for each** $a \in \cup Bottoms(Q)$
4. **if** $Conflict(P_\varnothing, \langle |\sigma^+| \cup \{a\}, |\sigma^-| \rangle)$ **&&**
5. $Conflict(P_\varnothing, \langle |\sigma^+|, |\sigma^-| \cup \{a\} \rangle)$ **then**
6. **return** False;
7. **else if** $Conflict(P_\varnothing, \langle |\sigma^+| \cup \{a\}, |\sigma^-| \rangle)$ **then**
8. $\sigma := \sigma \cup \{a^-\}$;
9. **else if** $Conflict(P_\varnothing, \langle |\sigma^+|, |\sigma^-| \cup \{a\} \rangle)$ **then**
10. $\sigma := \sigma \cup \{a^+\}$;
13. **end if**
14. **end for**
15. **until** $\sigma' = \sigma$
16. **return** P_σ

Fig.3 Bottom-Restricted look-ahead with DVF

Procedure *ComputeAns*(P_σ)
Input: A logic program P_σ;
Output: True if P_σ has an answer set that is also an answer set of $P_\varnothing$ otherwise False.

1. **if** *Look*-ahead(P_σ)=*False* **then return** False
2. $P_\sigma := Look\text{-}ahead(P_\sigma)$;
3. **if** $I(P_\sigma)^+$ is an answer set of P_σ **then**
4. **if** $I(P_\sigma)^+$ is an answer of $P_\varnothing$ **then return** True **else return** False
5. **end if**
6. $Q := P_\sigma \backslash I(P_\sigma)$;
7. Pick $a \in \cup Bottoms(Q)$ such that $w(a)$ is the greatest
8. **If** $w_2(a) > w_3(a)$ **then** $b := a^+$ **else** $b := a^-$
9. **if** $ComputeAns(P_{\sigma \cup \{b\}})$ **then return** True **else return** $ComputeAns(P_{\sigma \cup \{\neg b\}})$

Fig.4 Basic algorithm integrated with cycle breaking heuristic and bottom-restricted look-ahead

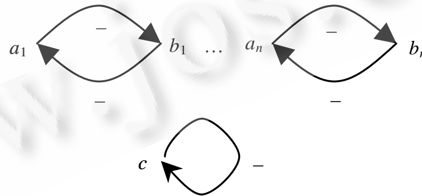

Fig.5 Dependency graph of P_3

6 Experimental Results

In this section we compare LPS with ASP systems Smodels, DLV and Nomore++[20]. The platform is a Celeron4 1.7GHz PC with 256M memory, running Fedora 2.4.22. LPS is compiled by *g*++ version 3.3.2 with optimization parameter $-O3$. In all tables, times are in second, tree means the search tree size, i.e. the number of the breaking nodes. We first experiment randomly generated benchmarks[21]. Let $k\text{-}LP(N,L)$ be the classes of programs that have N atoms and L rules with fixed length k, the authors of Ref.[21] discovered so-called hard-job-regions:

classes with L/N=4~6, i.e. these programs are generally difficult to solve. The following experiments are done with 3-LP(200,L) classes, which are considered to be moderate, i.e. they would not be too easy to distinguish the systems and too hard to compute.

The first benchmark consists of 282 programs from 11 classes where L/N=3~8 with a step of 0.5, following the “easy-hard-easy” pattern. Results are described in Table 1, LPS (SVF) outperforms other solvers in the regions of L/N=4.5~7 which fully cover the hard-job-regions, in easy-job-regions, Smodels is the best one. Note that LPS (DVF) has the smallest search tree size during the experiment (DLV provides no information about the search tree) and the running times are quite close to LPS (SVF).

Table 1 Experimental results on “easy-hard-easy” pattern

L/N	#Num	Smodels		LPS (SVF)		LPS (DVF)		Nomore++		DLV	
		Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
3	25	**2.51**	706	19.71	6181	15.58	**386**	26.41	387	68.85	–
3.5	24	**29.54**	7007	100.13	21415	88.97	**1552**	144.11	1725	748.37	–
4	24	**115.27**	22086	191.18	33332	174.74	**2495**	344.13	3282	1599.78	–
4.5	23	348.81	46266	**237.89**	31411	257.73	**2476**	548.36	4080	2100.10	–
5	27	257.70	35223	**207.74**	22820	218.33	**1808**	551.25	3424	1832.41	–
5.5	26	234.63	24454	**187.55**	16902	241.01	**1333**	586.01	2951	1420.56	–
6	26	144.04	13275	**118.27**	8878	144.07	**747**	415.85	1766	630.84	–
6.5	24	74.95	5825	**66.42**	4301	92.33	**371**	273.92	980	268.54	–
7	27	35.78	2548	**35.74**	1947	49.70	**182**	140.81	439	107.20	–
7.5	28	**16.11**	1088	23.98	1116	34.43	**110**	92.10	254	55.40	–
8	28	**8.5**	540	16.93	677	22.58	**72**	56.51	145	33.73	–

From the above results, it is natural to conjecture that LPS may work more efficient in hard-job-regions, so we select three hard classes with L/N=4.5, 5 and 5.5, for each class we experiment 1000 programs, the top 10 hard programs (according to Smodels) are shown in Tables 2~4 respectively. As we expected, LPS completely outperforms other systems on these top hard programs, where LPS (SVF) solved 24 out of 30 programs with the minimal times, and LPS (DVF) outperforms LPS (SVF) on the remaining 6 programs, and still, LPS (DVF) possesses the smallest search tree sizes.

We analyzed some hard programs and found that they contain complicated bottoms, it follows that the cycles occurring in these components are quite complex. They seem to be the reason why these instances are so difficult to solve. In some sense, the experimental results support our conjecture that LPS performs better than other systems in hard-job-regions. Furthermore, LPS (DVF) can efficiently prune the search space, thus has the minimal search tree sizes, this is mainly because LPS (DVF) chooses most breaking nodes during look-ahead instead of binary search.

Table 2 Top 10 out of 1000 hard programs, k=3, N=200, L/N=4.5

Instance No.	Smodels		LPS (SVF)		LPS (DVF)		Nomore++		DLV	
	Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
095	1122.85	165489	**212.30**	37953	314.33	**3042**	937.07	7246	1625.67	–
303	798.74	162475	**384.66**	49971	523.74	**4190**	951.59	7226	>3600	–
198	778.73	129200	**167.66**	22098	187.73	**1783**	350.82	2700	1938.20	–
703	635.92	113574	**129.63**	17838	136.12	**1430**	270.96	2230	627.89	–
536	598.45	53235	**394.81**	47904	398.52	**3683**	966.46	7747	>3600	–
400	561.42	71972	**392.93**	59896	499.34	**4507**	1348.54	10671	2262.15	–
540	516.70	51230	148.94	18566	**122.32**	**1470**	280.82	2121	588.04	–
695	493.15	58198	208.11	26933	**192.13**	**1741**	289.39	2178	2359.9	–
700	426.55	64440	**340.29**	50340	382.31	**3712**	914.39	6741	2390.54	–
592	367.32	62212	**175.79**	21790	206.08	**1790**	376.80	2890	1544.44	–

Table 3 Top 10 out of 1000 hard programs, k=3, N=200, L/N=5

Instance	Smodels		LPS (SVF)		LPS (DVF)		Nomore++		DLV	
No.	Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
174	779.34	93711	**156.29**	17549	195.98	**1449**	801.88	4250	2107.70	–
263	774.74	101593	**286.10**	34440	430.23	**3063**	648.61	3877	2748.91	–
021	691.43	103662	**164.72**	19392	209.54	**1599**	626.59	4250	1563.71	–
574	653.86	96238	139.36	16483	**128.51**	**1192**	388.87	2461	993.45	–
953	619.80	56523	207.76	27351	**205.26**	**2250**	694.94	3976	720.61	–
444	604.15	50463	**126.99**	14833	160.70	**1616**	562.09	3771	1525.78	–
529	603.35	78678	**277.26**	34310	445.43	**2331**	693.48	4306	1559.89	–
028	587.69	70425	**376.70**	41145	448.02	**3265**	891.20	5478	1748.90	–
196	553.17	72574	**100.78**	12306	119.61	**1066**	578.18	3659	1328.63	–
076	535.12	56781	**266.77**	33033	354.12	**2766**	584.74	3619	1557.45	–

Table 4 Top 10 out 1000 hard programs, k=3, N=200, L/N=5.5

Instance	Smodels		LPS (SVF)		LPS (DVF)		Nomore++		DLV	
No.	Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
255	817.47	63135	**351.13**	40027	500.73	**3060**	970.41	4858	>3600	–
214	683.31	73163	**143.70**	13261	188.16	**1135**	618.79	2857	1051.08	–
396	649.78	64759	**70.84**	7171	86.96	**871**	802.62	4133	1826.96	–
567	624.98	58082	**236.67**	24210	311.78	**1987**	1275.24	7038	2091.13	–
534	546.12	61502	202.66	17859	**74.31**	**670**	214.27	1106	235.80	–
233	439.80	39295	**108.60**	8480	152.77	**794**	558.04	2857	1393.03	–
409	408.06	41100	**236.06**	24718	329.77	**2117**	875.90	4941	1939.24	–
929	400.39	50450	**280.71**	27233	342.84	**2078**	617.15	3428	1107.26	–
855	379.38	34575	189.74	16482	**147.48**	**1343**	530.13	2990	651.68	–
182	348.43	20305	**124.17**	13879	195.94	**1190**	681.01	3406	566.30	–

Table 5 presents experimental results on some real-world benchmarks. Roughly speaking, the performance of LPS is close to other systems, though it is not the most efficient one in terms of running time. This is reasonable, since ASP is NP-Complete, the cycle breaking heuristic and the bottom-restricted look-ahead would not be efficient for all benchmarks. Moreover, Smodels is highly optimized for real-world applications, it is not surprising that it is the best one in Table 5. A good result is that, LPS (DVF) still appear more efficient than other systems in terms of tree sizes.

Table 5 Experimental results on bounded model checking

Instance	#Atom	#Rule	Smodels		LPS (SVF)		LPS (DVF)		Nomore++		DLV	
			Time	Tree	Time	Tree	Time	Tree	Time	Tree	Time	Tree
DP-8	691	1112	**0.16**	**9**	0.47	26	3.82	**9**	0.97	12	4.42	–
DP-10	1103	1790	**2.12**	296	12.77	339	16.73	**89**	14.31	116	28.43	–
DP-12	1611	2628	**347.97**	103557	429.36	82456	903.77	**9931**	565.47	32073	677.33	–
Elavator-1	1103	1596	**0.34**	**17**	2.41	55	5.30	19	3.32	15	3.54	–
Elavator-2	3195	4465	**4.58**	38	26.39	72	37.12	**22**	21.11	35	43.75	–
Elavator-3	7824	10660	**149.14**	130	221.23	166	241.18	**59**	198.43	107	565.9	–
Elavator-4	6437	8957	1164.34	1384	1613.22	1928	1825.12	**279**	**907.11**	796	>3600	–
Hartstone-50	854	1138	**2.17**	142	4.45	173	5.03	**77**	3.92	103	5.63	–
Hartstone-75	1254	1663	**10.94**	192	15.77	203	23.54	**99**	16.68	188	34.55	–
Hartstone-100	1654	2188	**26.25**	242	35.34	277	38.91	**66**	21.11	139	58.13	–

7 Related Work

In this section we briefly discuss features of other ASP system. Smodels uses several sophisticated inference rules for characterizing answer sets, during the binary search, it chooses an atom which maximizes the current partial model and restricts look-ahead by removing some unassigned atoms if they have been derived during consistency checking. DLV is designed and optimized for disjunctive logic programs[6,22], this may explain why DLV appears not so good during the above experiments since all programs are not disjunctive. Several heuristics are

© 中国科学院软件研究所 http://www.jos.org.cn

incorporated into DLV[18], one of these heuristics looks close to ours, however, the most important difference is that the heuristics in DLV do not concern the heads at all, while in our heuristic the heads do contribute to the evaluation function. For restricting the look-ahead, DLV defines so-called look-ahead equivalent literals, i.e. if two literals are look-ahead equivalent, then it is sufficient to only look one of them thus avoiding unnecessary looking. Nomore++[20] is an operator-based system, it adopts several semantics operators to compute answer sets and treats heads and bodies as equitably computational objects. It follows that its look-ahead not only perform on atoms but also on bodies, Ref.[20] shows that this hybrid look-ahead may save exponentially many choices on some instances. The heuristics in Nomore++ are operator-based and runtime configurable, by combing different semantics operators with choice operator, Nomore++ could enable several heuristics during the computation. It is worth to mention some SAT-based ASP systems like[23], which compute answer sets by SAT solvers. Generally speaking, SAT-based ASP systems are more efficient than the systems shown above, however, SAT-based ASP systems do not concern any ASP heuristics or implementation techniques at all, thus beyond the scope of this paper.

8 Conclusions and Future Directions

We present an ASP system called LPS, which employs a bottom-restricted look-ahead procedure together with an effective cycle breaking heuristic. The experimental results show that in phase transition hard-job-regions, LPS generally performs better than other ASP systems, i.e. LPS is good at solving random hard programs to some extent. Further, LPS with DVF (dynamic variable filtering) appears to be an efficient ASP system in terms of search tree size, this means DVF is theoretically a good approach for reducing search space.

It is worth to point out that random hard phenomena get more attention in recent years, they are believed to have strong connections with cryptography[24] (hard random propositional formulas or logic programs could be considered as one-way functions). Since LPS is good at solving random hard instances, it could be applied to the area of cryptography in addition to real-world applications. Our future work will focus on improving LPS, some advanced techniques like conflict recording[25], non-chronological backtracking[26] will be added, and various heuristics for real-world applications will also be studied in LPS.

References:

[1] Gelfond M, Lifschitz V. The stable model semantics for logic programming. In: Robert K, Kenneth B, eds. Proc. of the 5th Int'l Conf. on Logic Programming. Cambridge: MIT Press, 1988. 1070−1080.

[2] Niemela I. Logic programs with stable model semantics as a constraint programming paradigm. Annals of Math. and Artificial Intelligence, 1999,25(3-4):241−273.

[3] Baral C. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge: Cambridge University Press, 2003.

[4] Dantsin E, Eiter T, Gottlob G, Voronkov A. Complexity and expressive power of logic programming. ACM Computing Surveys, 2001,33(3):374−425.

[5] Simons P, Niemela I, Soininen T. Extending and implementing the stable model semantics. Artificial Intelligence, 2002,138(1-2): 181−234.

[6] Eiter T, Faber W, Leone N, Pfeifer G. Declarative problem-solving using the DLV system. In: Logic-Based Artificial Intelligence. Massachusetts: Kluwer Academic Publishers, 2000. 79−103.

[7] Lin FZ, Zhao XS. On odd and even cycles in normal logic programs. In: Deborah M, George F, eds. Proc. of the 19th National Conf. on Artificial Intelligence. 2004. 80−85.

[8] Faber W, Leone N, Pfeifer G. Optimizing the computation of heuristics for answer set programming systems. In: Thomas E, Wolfgang F, Miroslaw T, eds. Proc of the 6th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning. Berlin: Springer-Verlag, 2001. 295−308.

© 中国科学院软件研究所 http://www.jos.org.cn

[9] Van Gelder A, Ross KA, Schlipf JS. The well-founded semantics for general logic programs. Journal of the ACM, 1991,38(3):620−650.

[10] Baral CR, Subrahmanian VS. Dualities between alternative semantics for logic programming and nomonotonic reasoning. Journal of Automated Reasoning, 1993,10(2):399−420.

[11] Cholewinski P, Truszczynski M. External problems in logic programming and stable model computation. Journal of Logic Programming, 1999,38(2):219−242.

[12] Cormen TH, Leiserson CE, Riverst RL, Stein C. Introduction to Algorithms. Cambridge: MIT Press, 2001.

[13] Sato T. Completed logic programs and their consistency. Journal of Logic Programming, 1990,9(1):33−44.

[14] You JH, Yuan LY. A three-valued semantics for deductive database and logic programs. Journal of Computer and System Science, 1994,49(2):334−361.

[15] Zhao JC. A study of answer set programming [MS. Thesis]. Hong Kong: Hong Kong University of Science and Technology, 2002.

[16] Freeman JW. Improvements to propositional satisfiability search algorithms [Ph.D. Thesis]. University of Pennsylvania, 1995.

[17] Li CM, Anbulagan. Heuristics based on unit progagation for satisfiability problems. In: Proc. of the 15th Int'l Jonit Conf. on Artificial Intelligence. San Mateo: Morgan Kaufmann Publishers, 1997. 366−371.

[18] Faber W, Leone N, Pfeifer G. Experimenting with heuristics for answer set programming. In: Bernhard N, ed. Proc. of the 17th Intl'l Jonit Conf. on Artificial Intelligence. San Mateo: Morgan Kaufmann Publishers, 2001. 635−640.

[19] Anbulagan. Extending unit propagation look-ahead of DPLL procedure. In: Chengqi Z, Hans WG, Wai-Kiang Y, eds. Proc. of the 8th Pacific Rim Int'l Conf. on Artificial Intelligence. Berlin: Springer-Verlag, 2004. 173−182.

[20] Anger C, Gebser M, Linke T, Neumann A, Schaub T. The Nomore++ system. In: Chitta B, Gianluigi G, Nicola L, Giorgio T, eds. Proc of the 8th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning. Berlin: Springer-Verlag, 2005. 422−426.

[21] Zhao YT, Lin FZ. Answer set programming phase transition: A study on randomly generated programs. In: Catuscia P, ed. Proc. of the 19th Int'l Conf. on Logic Programming. 2003. 239−253.

[22] Wang KW, Zhou LZ, Chen HW. An argumentation-based framework for extended disjunctive logic programs. Journal of Software, 2000,11(3):293−299 (in English with Chinese abstract).

[23] Lin FZ, Zhao YT. ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence, 2004,157(1-2): 115−137.

[24] Cook SA, Mitchell DG. Finding hard instances of the satisfiability problem: A survey. In: Proc. of the DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1997. 35:1−17.

[25] Lynce I, Marques-Silva J. The effect of nogood recording in DPLL-CBJ SAT algorithms. In: Barry O, ed. Proc. of the Int'l Workshop on Constraint Solving and Constraint Logic Programming. 2003. 144−158.

[26] Bayardo Jr RJ, Schrag RC. Using CSP look-back techniques to solve real-world SAT instances. In: Proc. of the 14th National Conf. on Artificial Intelligence. California: AAAI Press, 1997. 203−208.

附中文参考文献:

[22] 王克文,周立柱,陈火旺.扩充析取逻辑程序的争论语义.软件学报,2000,11(3):293−299.

SHEN Yu-Ping was born in 1980. He is a Ph.D. candidate at the Institute of Logic and Cognition, Sun Yat-sen University. His research areas are knowledge representation and reasoning, logic programming and propositional satisfiability.

ZHAO Xi-Shun was born in 1964. He is a professor and doctoral supervisor at the Institute of Logic and Cognition, Sun Yat-sen University. His research areas are computational complexity, non-monotonic reasoning and propositional satisfiability.

© 中国科学院软件研究所 http://www.jos.org.cn