Lookahead in Smodels Compared to Local Consistencies in CSP

Jia-Huai You, Guohua Liu, Li Yan Yuan, Curtis Onuczko

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{you,guohua,yuan,onuczko}@cs.ualberta.ca

Abstract. In answer set programming systems like Smodels and some SAT solvers, constraint propagation is carried out by a mechanism called lookahead. The question arises as what is the pruning power of lookahead, and how such pruning power fares in comparison with the consistency techniques in solving CSPs. In this paper, we study the pruning power of lookahead by relating it to local consistencies under two different encodings from CSPs to answer set programs. This leads to an understanding of how the search space is pruned in an answer set solver with lookahead for solving CSPs. On the other hand, lookahead as a general constraint propagation mechanism provides a uniform algorithm for enforcing a variety of local consistencies. We also study the impact on the search efficiency under these encodings.

1 Introduction

Constraint satisfaction problems (CSPs) on the one hand and propositional satisfiability (SAT) and answer set programming (ASP) under the stable model semantics [9] on the other are two competing approaches to constraint programming.

CSPs are typically solved by a systematic backtracking search algorithm, whereas at each choice point consistency of a certain kind is maintained for constraint propagation. Many SAT and answer set solvers are based on the DP procedure (the Davis-Putnam-Logemann-Loveland algorithm) [4], where a main mechanism for space pruning is lookahead [8] - before a guess on a choice point is made, for each atom, if fixing the atom's truth value leads to a contradiction, the atom gets the opposite truth value. In this way, an atom may be propagated from already assigned atoms without going through a search process.

ASP has been advocated as an emerging paradigm of constraint programming for solving a variety of constraint problems, including CSPs [12]. It is therefore important to understand how the search space is pruned in solving CSPs. Such studies can potentially benefit both sides - an effective method in one approach may be adopted by another to improve the search efficiency. The relationship between the two becomes more interesting recently in light of the fact that answer set solvers have been integrated with CSP solvers [7]. One would expect more to come in combining the two in the future.

The research direction of this paper started in [16], where it is shown that lookahead is strictly stronger than arc consistency under Niemelä's encoding with an understanding of where the added pruning power comes from.

In this paper, we extend our investigation in two fronts, one of which is to consider different encodings, and the other is to study how higher level local consistencies for CSPs such as i-consistency may be captured in lookahead. For the first goal we consider two familiar encodings in the SAT literature, the *direct encoding* and *support encoding*. In the direct encoding, disallowed tuples are expressed. We show that the pruning power of lookahead under the direct encoding is the same as the pruning power under Niemelä's encoding where allowed tuples are expressed. The pruning power of lookahead under these two encodings is precisely that of arc consistency enhanced by propagation of unique domain values to variables [16]. However, we will see that these two complementary encodings have different inference powers if tuples of literals are tested in lookahead. For the support encoding, we show that lookahead coincides with a stronger local consistency called *singleton arc consistency* in the literature [2, 5]. This shows that an idea similar to lookahead had been formulated independently for CSP. We also show that by testing tuples instead of individual literals, lookahead can capture higher level consistencies, such as i-consistency and *singleton restricted path consistency* [6]. The possibility of testing n-tuples in lookahead was briefly discussed in Simons' thesis [14].

The work on relating CSP with SAT are relevant here. In [15] several encodings of CSP in SAT are compared. In general, unit propagation is weaker than arc consistency. Kasif [11] on the other hand shows that the support encoding of binary CSPs in SAT can have arc consistency established by unit propagation. Gent in [10] reports experiments that show the support encoding handles hard instances of random binary CSP more effectively than the direct encoding. Kasif's work is further extended to some higher levels and wider range of local consistencies [3]. However, none of these works consider the more powerful mechanism of lookahead in SAT solvers.

The next section defines terminologies for CSPs. Section 3 introduces answer set programming and the lookahead algorithm as defined in Smodels. Section 4 deals with direct encoding while Section 5 treats support encoding. Then, in Section 6 we discuss some relationships between the direct encoding and Niemelä's encoding [12] from CSPs to answer set programs. In Section 7 we show some experimental results that confirm some of our theoretical findings. Section 8 provides a summary.

2 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a triple $\mathcal{A}(X, D, C)$ where $X = \{x_1, \ldots, x_n\}$ is a finite set of variables with respective domains $D = \{D_{x_1}, \ldots, D_{x_n}\}$ listing the possible values for each variable, and C is a finite set of constraints. A constraint $c_{y_1,\ldots,y_k} \in C$ is a subset of the Cartesian product $D_{y_1} \times ... \times D_{y_k}$. We denote by $\mathcal{A}|_{D_x=\{a\}}$ the CSP obtained by restricting the domain D_x to $\{a\}$ in $\mathcal{A}$.

Given a CSP $\mathcal{A}(X, D, C)$, if a tuple (a, b) is in a binary constraint c_{xy}, we say a is a *support* of b w.r.t. c_{xy} and b is a *support* of a w.r.t. c_{xy}. We say x is a *neighboring variable* of y, and vice versa, if there is a constraint c_{xy} or c_{yx} in C.

An *instantiation* is a variable assignment where each variable is assigned a unique value from its domain. An assignment may be partial. A partial instantiation is *consistent* with an n-ary constraint $c \in C$ iff the assignment yields a projection of a tuple in the relation of c. A partial instantiation is consistent iff it is consistent with every constraint. A constraint $c_{y_1,\ldots,y_k} \in C$ is *satisfied* iff the assignment to the variables $y_1, \ldots, y_k$ forms a tuple in the relation of $c_{y_1,\ldots,y_k}$. A *solution* to a CSP is an instantiation of all variables that satisfy all the constraints. We denote by $x \rightarrow a$ that variable x is assigned value a from its domain.

Given a CSP $\mathcal{A}(X, D, C)$, we use n to denote the number of variables in X, e the number of constraints in C, and d the maximum domain size. In this paper we deal only with binary constraints.

A CSP is *i-consistent* iff given any consistent instantiation of any $i - 1$ variables, there exists an instantiation of any ith variable such that the i values taken together satisfy all of the constraints among the i variables. The most popular degrees of consistencies are *arc consistency* when $i = 2$, and *3-consistency* when $i = 3$ (which coincides with *path consistency* for binary constraints).

A local consistency LC is said to be *stronger than* another local consistency LC' if for any CSP in which LC holds so does LC'.

Maintaining (or enforcing) arc consistency on a CSP is a domain reduction process $-$ it removes inconsistent values from the domains of unassigned variables. This is compared with maintaining higher level consistencies where *nogood* tuples are identified. More recently, based on the idea of domain reduction, some additional notions of consistency for domain reduction are introduced [13, 6], among which singleton arc consistency and singleton restricted path consistent are particularly interesting due to their potential in practical applications. We give their definitions below.

Let $\mathcal{A}(X, D, C)$ be a CSP.

– $\mathcal{A}$ is *singleton arc consistent* (SAC) iff $\forall x \in X$, $D_x \neq \emptyset$, and $\forall a \in D_x$, $\mathcal{A}|_{D_x=\{a\}}$ can be made arc consistent with non-empty domains.
– $\mathcal{A}$ is *restricted path consistent* (RPC) iff $\forall x \in X$, D_x has a non-empty arc consistent domain and, $\forall a \in D_x$ and $\forall y \in X$ such that a has a unique support $b \in D_y$, and $\forall z \in X$, $\exists c \in D_z$ such that $(a, c) \in c_{xz}$ and $(b, c) \in c_{yz}$.[1]
– $\mathcal{A}$ is *singleton restricted path consistent* (SRPC) iff $\forall x \in X$, $D_x \neq \emptyset$, and $\forall a \in D_x$, $\mathcal{A}|_{D_x=\{a\}}$ can be made restricted path consistent.

Intuitively, in order to enforce a certain kind of singleton consistency, one restricts each variable to each of its domain values and then enforces that consistency. If enforcing that consistency causes an empty domain, we then know the corresponding value of that variable cannot contribute to any solution and thus should be removed from its domain. One can see that the notion of singleton consistencies bears an idea similar to that of lookahead. In the latter, since we are dealing with Boolean variables, removing a value of a variable simply means to assign the variable with the opposite (truth) value.

Among singleton consistencies, singleton arc consistency is the most extensively studied in the literature of CSP, due to its relatively low complexity.

[1] When a constraint is not explicitly presented in C, it means all the tuples in the corresponding Cartesian product are allowed.

As *singleton* can be prefixed to any consistency, the next computationally realistic consistency to be considered is path consistency. From the definition of path consistency, two variables are required to be instantiated with all possible domain values to see if a conflict exists. To reduce the complexity, restricted path consistency is proposed where only the domain values that satisfy the stated condition in the definition are tested. In the literature, more complex singleton consistencies are considered to be only of theoretical interest [6].

3 Constraint Propagation in Smodels

Smodels implements the stable model semantics for normal logic programs which consist of rules of the form $A \leftarrow B_1, ..., B_m, \texttt{not}\ C_1, ..., \texttt{not}\ C_n.$, where A, B_i and C_i are function-free atoms, and $\texttt{not}\ C_i$ are *default negations* or simply called *not-atoms*. The head A may be the special atom $\perp$, in which case it serves as a constraint. In systems like Smodels, these programs are first instantiated to ground instances for the computation of answer sets.

An *answer set* (also called a *stable model*) is defined over the ground instantiation of a given program [9]. A set of atoms M is an answer set for a program P iff M is the least model of P^M, where P^M is defined as

$$P^M = \{a \leftarrow b_1, ..., b_m \mid a \leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n \in P \text{ and } \\ \forall i \in [1..n], c_i \notin M\}$$

Additional notations: Atoms and not-atoms are both called *literals*. A set of literals is consistent if there is no atom ϕ such that ϕ and $\texttt{not}\ \phi$ are both in the set.

$Atoms(\Phi)$ denotes the set of distinct atoms appearing in Φ (excluding the special atom $\perp$). The expression $not(\texttt{not}\ \phi)$ is identified with ϕ, and $not(\phi)$ is $\texttt{not}\ \phi$. Given a set of literals B, $B^+ = \{\xi \mid \xi \text{ is an atom in } B\}$ and $B^- = \{\xi \mid \texttt{not}\ \xi \in B\}$. Suppose Q is a set of atoms. Then we define $not(Q) = \{\texttt{not}\ \phi \mid \phi \in Q\}$.

Constraint propagation in Smodels is carried out by lookahead (cf. Figure 1 where P is a program and A a set of literals representing a partial truth value assignment). For each unassigned atom ϕ, lookahead assumes a truth value for it (via the function lookahead_once), if that leads to a conflict, then the opposite truth value for ϕ is in any answer set M agreeing with A (meaning $A^+ \subseteq M$ and $A^- \cap M = \emptyset$). Truth values are propagated in lookahead by a function called $expand(P, A)$ (cf. Figure 2), which returns a superset of A, representing the process of propagating the values of the atoms in A to some additional atoms. In lookahead_once, the function $conflict(P, A)$ returns true if $A^+ \cap A^- \neq \emptyset$ and false otherwise.

$Atleast(P, A)$ in the expand function returns a superset of A by repeatedly applying four propagation rules until no new literals can be deduced. Let r be a rule in program P of the form: $r = h \leftarrow a_1, ..., a_n, \texttt{not}\ b_1, ..., \texttt{not}\ b_m$. Define

$$\begin{aligned} min_r(A) &= \{h \mid \{a_1, ..., a_n\} \subseteq A^+, \{b_1, ..., b_m\} \subseteq A^-\} \\ max_r(A) &= \{h \mid \{a_1, ..., a_n\} \bigcap A^- = \emptyset, \\ &\qquad \{b_1, ..., b_m\} \bigcap A^+ = \emptyset\} \end{aligned}$$

The four propagation rules in $Atleast(P, A)$ are:

```
$Function\ lookahead(P, A)$
  $repeat$
  $A' := A$
  $A := lookahead\_once(P, A)$
  $until\ A = A'$
$return\ A.$
```

```
$Function\ lookahead\_once(P, A)$
  $B := Atoms(P) - Atoms(A)$
  $B := B \bigcup not\ (B)$
  $while\ B \neq \emptyset\ do$
      $take\ any\ literal\ \chi \in B$
      $A' := expand(P, A \bigcup \{\chi\})$
      $B := B - A'$
      $if\ conflict(P, A')\ then$
          $return\ expand(P, A \bigcup \{not\ (\chi)\})$
  $end\ while$
$return\ A.$
```

Fig. 1. Function $lookahead(P, A)$

```
$Function\ expand(P, A)$
  $repeat$
  $A' := A$
  $A := Atleast(P, A)$
  $A := A \bigcup \{\texttt{not}\ \phi \mid \phi \in Atoms(P)$ and $\phi \notin Atmost(P, A)\}$
  $until\ A = A'$
$return\ A.$
```

Fig. 2. Function $expand(P, A)$

1. If $r \in P$, then $A := A \bigcup min_r(A)$.
2. If there is an atom a such that for all $r \in P$, $a \notin max_r(A)$, then $A := A \bigcup \{not\ a\}$.
3. If an atom $a \in A$, there is only one $r \in P$ for which $a \in max_r(A)$, and there is a literal x such that $a \notin max_r(A \bigcup \{x\})$, then $A := A \bigcup \{not(x)\}$.
4. If $not\ a \in A$ and there is a literal x s.t. for some $r \in P$, $a \in min_r(A \bigcup \{x\})$, then $A := A \bigcup \{not(x)\}$.

Rule 1 adds the head of a rule to A if the body is true in A. If there is no rule with a as the head whose body is not false w.r.t. A, then a cannot be in any answer set agreeing with A. This is Rule 2. Rule 3 says that if $a \in A$, the only rule with a as the head whose body is not yet false must have its body true in extending A. Rule 4 forces the body of a rule to be false if the head is false in A.

The function $Atmost(P, A)$ is of little relevance here, since the answer set programs that encode CSPs in this paper do not have "positive loops". For these programs, it can be shown that the expand function behaves as if the fifth line in the definition were removed.

4 Constraint Propagation under Direct Encoding

The direct encoding of a CSP by an answer set program consists of two parts. The first part specifies the *uniqueness property* - a variable in CSP is assigned exactly one value from its domain. For each variable $x \in X$ and its domain $D_x = \{a_1, ..., a_d\}$, we use an atom $x(a_j)$ to represent that x gets the value $a_j \in D_x$. Then, for each $j \in [1..d]$ we have a rule

$$x(a_j) \leftarrow \texttt{not}\ x(a_1), ..., \texttt{not}\ x(a_{j-1}), \texttt{not}\ x(a_{j+1}), ..., \texttt{not}\ x(a_d). \tag{1}$$

These rules will be referred to as the *uniqueness encoding*.

In the second part, disallowed tuples in constraints are expressed. For each constraint $c_{xy} \in C$, and for each tuple $(a, b) \notin c_{xy}$, where $a \in D_x$ and $b \in D_y$, we have a *rule of denial*

$$\bot \leftarrow x(a), y(b). \tag{2}$$

The direct encoding of a CSP $\mathcal{A}$ is denoted $P_{dir}(\mathcal{A})$.

The correctness of the direct encoding is obvious. For the complexity, for any CSP $\mathcal{A}$, it can be verified that $O(nd^2 + ed^2)$ bounds the size of $P_{dir}(\mathcal{A})$.

4.1 Propagation Arc Consistency

Given a CSP $\mathcal{A}(X, D, C)$ and a collection Π of pairs $x \rightarrow a$, *unique value propagation* is an operation that generates an extension of Π, which is

$$\begin{aligned} \Pi \cup \{x \rightarrow a \mid\ & c_{xy} \in C \text{ or } c_{yx} \in C, \text{ and } y \rightarrow b \in \Pi \\ & \text{such that } a \text{ is the only value in } D_x \text{ consistent with } b\} \end{aligned}$$

A collection of pairs Π *is closed under (unique value) propagation* if it cannot be extended further by unique value propagation. Π is said to be in *conflict* (or *conflicting*) if there are distinct values a and a' such that for some variable x, $x \rightarrow a, x \rightarrow a' \in \Pi$, otherwise it is non-conflicting.

Definition 1. *A CSP $\mathcal{A}(X, D, C)$ is* propagation arc consistent *(PAC) iff it is arc consistent, and $\forall x \in X$ and $\forall a \in D_x$, the closure of $\{x \rightarrow a\}$ under unique value propagation is non-conflicting.*

It is clear from the definition that PAC is stronger than arc consistency (AC).

Example 1. Consider a CSP with three variables x, y, and z, all with the domain $\{0, 1\}$, and the following constraints:

$$c_{xy} = \{(0, 0), (1, 1)\} \qquad c_{yz} = \{(0, 1), (1, 0)\} \qquad c_{zx} = \{(0, 0), (1, 1)\}$$

It is clear that this CSP is arc consistent. However, it is not PAC since $\{x \rightarrow 0\}$ leads to a conflict under unique value propagation.

In the direct encoding of this CSP, besides the uniqueness encoding, the disallowed tuples are expressed by the following rules:

$$\begin{array}{lll} \bot \leftarrow x(0), y(1). & \bot \leftarrow y(0), z(0). & \bot \leftarrow z(0), x(1). \\ \bot \leftarrow x(1), y(0). & \bot \leftarrow y(1), z(1). & \bot \leftarrow z(1), x(0). \end{array}$$

Now, we have $\texttt{not}\ x(0) \in \mathit{lookahead}(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$, corresponding to 0 being removed from its domain in CSP. This is obtained by assuming $x(0)$, and then by applying the expand function to infer $\texttt{not}\ y(1)$, $y(0)$, $\texttt{not}\ z(0)$, $z(1)$, and $\texttt{not}\ x(0)$, resulting in a conflict.

Under the direct encoding, the pruning power of lookahead is precisely that of PAC.

Theorem 1. *Let $\mathcal{A}(X, D, C)$ be a CSP. A value a is removed from its domain D_x by maintaining PAC on $\mathcal{A}$ iff $\texttt{not}\ x(a) \in lookahead(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$.*

A proof of this theorem is relatively routine but can be lengthy. Roughly, we note that it is known that the result of lookahead is independent of the order in which literals are picked up in lookahead_once [16]. Thus, we only need to show that for any values $a_1, ..., a_k$ that are removed from $D_{x_1}, ..., D_{x_k}$, perspectively, in enforcing PAC, $lookahead(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$ infers $\texttt{not}\ x_1(a_1), ..., \texttt{not}\ x_k(a_k)$, in the same order (and vice versa). Since the effect of value removals is incremental, we can apply a simple induction on k to show that if in lookahead_once $x_i(a_i)$ is picked up, the expand function generates a conflict, assuming all $\texttt{not}\ x_1(a_1), ..., \texttt{not}\ x_k(a_{i-1})$ have already been added by lookahead.

4.2 Extension to i-consistency

Though strictly stronger than AC, it is not a surprise that PAC in general is not as powerful as higher level consistencies.

Example 2. Consider a CSP with three variables x, y, and z, all with the domain $\{0, 1, 2, 3\}$, and three constraints

$$c_{xy} = \{(0,0), (0,1), (1,2), (1,3), (2,0), (2,1), (3,2), (3,3)\}$$
$$c_{xz} = \{(0,0), (0,1), (1,2), (1,3), (2,0), (2,1), (3,2), (3,3)\}$$
$$c_{zy} = \{(0,2), (1,3), (2,0), (3,1), (1,2), (0,3), (3,0), (2,1)\}$$

This CSP is not path consistent: for any pair $(a_i, a_j) \in c_{xy}$ there exists no value $a_k \in D_z$ such that $(a_i, a_k) \in c_{xz}$ and $(a_k, a_j) \in c_{zy}$. However, this CSP is arc consistent. One can also check that for this CSP there is no possibility of unique value propagation. Therefore, it is PAC.

Lookahead may be extended to test tuples of literals instead of single literals [14]. Here we show that i-consistency can be captured by testing $(i-1)$-tuples.

Theorem 2. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any $1 < i \le |X|$, let $\{y_1, ..., y_{i-1}\}$ be any $i-1$ variables in X and y_i be any ith variable. For any consistent instantiation $\Pi = \{y_1 \rightarrow b_1, ..., y_{i-1} \rightarrow b_{i-1}\}$, if y_i has no instantiation consistent with Π, then a conflict will be generated in $expand(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot, y_1(b_1), ..., y_{i-1}(b_{i-1})\})$.*

Note that although this theorem is not stated directly using lookahead, it is clear that what it states is that any inconsistency in enforcing i-consistency can be detected by testing the corresponding $(i-1)$-tuple by lookahead. This implies that if lookahead tests all $(i-1)$-tuples, and the result of lookahead in such testing is independent of any order in which these tuples are tested, then i-consistency is enforced.

Proof. That there is no instantiation of y_i that is consistent with Π implies that for any $v \in D_{y_i}$, there exists $y_j \in \{y_1, ..., y_{i-1}\}$, such that $b_j \in D_{y_j}$ is inconsistent with v. Thus, in $P_{dir}(\mathcal{A})$ there must be a rule of denial $\bot \leftarrow y_j(b_j), y_i(v)$. Since $y_j(b_j)$ is in the input set, $\texttt{not}\ y_i(v)$ can be inferred by Rule 4 of the Atleast function. Since this is the case for any $v \in D_{y_i}$, a conflict will be generated by the uniqueness encoding in $P_{dir}(\mathcal{A})$. □

5 Constraint Propagation under Support Encoding

The support encoding of a CSP by an answer set program consists of two parts. The first part is the uniqueness encoding, the same as in the direct encoding. The difference lies in the second part: for each constraint $c_{xy} \in C$ and for each value $a \in D_x$, let $\{b_1, ...b_k\}$ be the set of the supports for a w.r.t. c_{xy}, the *support rule* for a w.r.t. c_{xy} is

$$\perp \leftarrow x(a), \texttt{not}\ y(b_1), ..., \texttt{not}\ y(b_k). \tag{3}$$

Similarly, for each value b in D_y, let $\{a_1, ...a_{k'}\}$ be the set of the supports for b w.r.t. c_{xy}, the *support rule* for b w.r.t. c_{xy} is

$$\perp \leftarrow y(b), \texttt{not}\ x(a_1), ..., \texttt{not}\ x(a_{k'}). \tag{4}$$

The support encoding of a CSP $\mathcal{A}$ is denoted $P_{sup}(\mathcal{A})$.

Example 3. Let a CSP have two variables x and y with domains $D_x = D_y = \{0, 1\}$, and a constraint $c_{xy} = \{(0, 0), (0, 1), (1, 0)\}$. The support rules for c_{xy} are:

$$\begin{array}{ll} \perp \leftarrow x(0), \texttt{not}\ y(0), \texttt{not}\ y(1). & \perp \leftarrow y(0), \texttt{not}\ x(0), \texttt{not}\ x(1). \\ \perp \leftarrow x(1), \texttt{not}\ y(0). & \perp \leftarrow y(1), \texttt{not}\ x(0). \end{array}$$

It can be seen that the size of the resulting program by support encoding remains to be bounded by $O(nd^2+ed^2)$. Again, the correctness of the support encoding is obvious.

Our goal in this section is to show that, under the support encoding, lookahead coincides with singleton arc consistency (SAC). First, we should place this result in a context, namely the fact that SAC is strictly stronger than propagation arc consistency (PAC) introduced in the last section. It is easy to see that whenever unique value propagation is possible, enforcing AC will carry it out. Thus, SAC is stronger than PAC. Example 4 below shows a CSP which is PAC but not SAC.

Example 4. Consider a CSP with three variables all with domain $D_x = D_y = D_z = \{0, 1, 2, 3\}$, and three constraints

$$\begin{array}{l} c_{xy} = \{(0,0), (0,1), (1,0), (1,1), (2,2), (2,3), (3,2), (3,3)\} \\ c_{yz} = \{(0,0), (0,1), (1,0), (1,1), (2,2), (2,3), (3,2), (3,3)\} \\ c_{zx} = \{(0,2), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (3,1)\} \end{array}$$

This CSP is arc consistent, and there is no unique value propagation; hence it is PAC. But it is not SAC. Suppose we restrict D_x to $\{0\}$. Enforcing AC removes 2 and 3 from D_y as well as from D_z, resulting in 0 being removed from D_x.

We therefore have

Theorem 3. *SAC is strictly stronger than PAC.*

Now, let us use the following example to explain the main technical result of this section.

Example 5. Consider a CSP $\mathcal{A}$ with three variables all with domain $\{0, 1\}$, and three constraints:

$$c_{xy} = \{(0,0),(0,1),(1,1)\} \qquad c_{yz} = \{(0,0),(1,1)\} \qquad c_{zx} = \{(0,1),(1,0)\}$$

Enforcing arc consistency on $\mathcal{A}|_{D_x=\{1\}}$ will eventually remove 1 from D_x. On the other hand, we have $\mathtt{not}\ x(1) \in \mathit{lookahead}(P_{sup}(\mathcal{A}), \{\mathtt{not}\ \bot\})$.

The following lemma is essential in proving the main result.

Lemma 1. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any value $a \in D_x$, a is removed by enforcing arc consistency iff $\mathtt{not}\ x(a) \in \mathit{Atleast}(P_{sup}(\mathcal{A}), \{\mathtt{not}\ \bot\})$.*

The lemma can be proved by an induction on the sequence of value removals. For the base case, for any constraint c_{xy}, if a value $a \in D_x$ does not have a support in D_y, the corresponding support rule in $P_{sup}(\mathcal{A})$ is $\bot \leftarrow x(a)$. By Rule 4 of $\mathit{Atleast}$ we infer $\mathtt{not}\ x(a)$. For the inductive step, suppose a value a in D_x is removed because all of its supports in D_y have already been removed. Then, the corresponding rule in $P_{sup}(\mathcal{A})$ is

$$\bot \leftarrow x(a), \mathtt{not}\ y(b_1), ..., \mathtt{not}\ y(b_k).$$

Again by Rule 4 of the Atleast function, because all $\mathtt{not}\ y(b_i)$, $i \in [1..k]$ are already inferred, we must have $\mathtt{not}\ x(a)$. The argument for the other direction is similar.

This lemma leads to the following theorem.

Theorem 4. *Let $\mathcal{A}(X, D, C)$ be a CSP. A value a is removed from its domain D_x by maintaining SAC on $\mathcal{A}$ iff $\mathtt{not}\ x(a) \in \mathit{lookahead}(P_{sup}(\mathcal{A}), \{\mathtt{not}\ \bot\})$.*

We can further show that by testing pairs of literals, lookahead captures singleton restricted path consistency (SRPC).

Theorem 5. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any constraint $c_{xy} \in C$, if a pair $(a, b) \in c_{xy}$ is identified as* nogood *by maintaining RPC on $\mathcal{A}|_{D_z=\{v\}}$, then either $\mathit{expand}(P_{sup}(\mathcal{A}), \{\mathtt{not}\ \bot, z(v), x(a)\})$ or $\mathit{expand}(P_{sup}(\mathcal{A}), \{\mathtt{not}\ \bot, z(v), y(b)\})$ generates a conflict.*

Note that, in enforcing SRPC, inconsistencies are detected by fixing three values, one for domain restriction and the other two for testing a path. Since one of the latter two is a unique support of the other, it can be propagated in the Atleast function. This is why lookahead only needs to test two values.

6 Relationships

Niemelä proposes to encode a CSP by representing allowed tuples [12]. The resulting program consists of the uniqueness encoding and the rules for allowed tuples: for each constraint c_{xy}, and each pair $(a, b) \in c_{xy}$, we have

$$sat(c_{xy}) \leftarrow x(a), y(b). \tag{5}$$

where $sat(c_{xy})$ is a new atom. We then ask an answer set generator to compute the answer sets that contain all $sat(c_{xy})$ where $c_{xy} \in C$.

It is shown in [16] that under Niemelä's encoding lookahead coincides with PAC. However, the following example shows that Theorem 2 does not hold under Niemelä's encoding, illustrating that these two complementary encodings behave differently when testing tuples of literals.

Example 6. Given three variables and their domains $D_x = D_y = \{0\}$ and $D_z = \{0, 1, 2, 3\}$, and three constraints

$$c_{xy} = \{(0,0)\} \;\; c_{xz} = \{(0,0),(0,1)\} \;\; c_{yz} = \{(0,2),(0,3)\}$$

$\{x \rightarrow 0, y \rightarrow 0\}$ cannot be consistently extended to z. Under the direct encoding, we have rules of denial as follows

$$\bot \leftarrow x(0), z(2). \quad \bot \leftarrow x(0), z(3). \quad \bot \leftarrow y(0), z(0). \quad \bot \leftarrow y(0), z(1).$$

When we test the pair $\langle x(0), y(0)\rangle$, we derive `not` $z(0)$, `not` $z(1)$, `not` $z(2)$, and `not` $z(3)$, from which conflicts are derived by the uniqueness encoding.

However, one can check that no conflict can be derived by the expand function under Niemelä's encoding. The rules for allowed tuples in Niemelä's encoding are:

$$\begin{array}{ll} sat(c_{xy}) \leftarrow x(0), y(0). & \\ sat(c_{xz}) \leftarrow x(0), z(0). & sat(c_{xz}) \leftarrow x(0), z(1). \\ sat(c_{yz}) \leftarrow y(0), z(2). & sat(c_{yz}) \leftarrow y(0), z(3). \end{array}$$

When we test the pair $\langle x(0), y(0)\rangle$, for $sat(c_{xz})$ to be true one of the literals in $\{z(0), z(1)\}$ must be true; for $sat(c_{yz})$ to be true one of the literals in $\{z(2), z(3)\}$ must be true. But their intersection is empty. To arrive at the conclusion of conflict, one has to reason disjunctively along with the uniqueness encoding. But the expand function of Smodels does not perform this type of reasoning.

7 Experiments

The experiments were designed to test the following: the difference in performance of the direct and support encodings from CSP to ASP.

We ran the direct encoding and support encoding on a wide range of problems varying the number of variables, the size of the domains, the number of constraints and the number of no-goods. We found that problems with approximately 10 variables are best for testing, as increasing the number of variables can significantly extend the running time of the program for the more complex problems. We then tested with various domain sizes and made the number of constraints range from 5 to 45. The number of no-good values as a percentage of the constraints is called the tightness, which in our experiments ranged from 5% to 95%.

When measuring the running time of the ASP programs, we only report the time required to run Smodels and omit the time required to ground the problem through *lparse*. We do this because it is possible to directly translate from the CSP instance to a grounded instance without having to use a grounding program. All times were recorded using the *time* command in Linux, and recording the user CPU time.

All experiments were performed by generating random CSP instances using the generator found at [1]. All results reported were performed on an Intel Pentium 4 2.00 Ghz machine with 512 Mb of RAM running Slackware Linux 10.0.

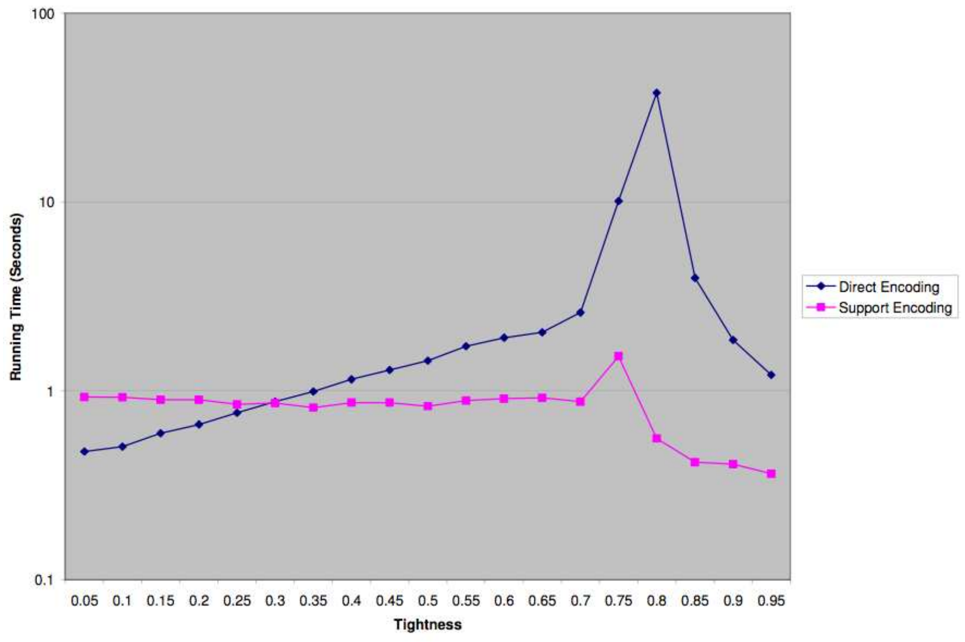

Fig. 3. Run time vs. tightness.

Figure 3 shows the running time (in seconds) of Smodels plotted against the tightness of the problems. The tests were performed on problems with 10 variables, a domain size of 30, and 20 constraints. Each point on the graph represents the average running time of 10 random instances. The direct encoding begins by having a better running time than the support encoding, but is quickly outdone by the support encoding as the tightness of the problems increase. This is especially apparent as the problem set is about to enter its phase transition; the support encoding is greatly outperforming direct encoding.

It is worth noting that Gent performed similar tests between the direct and support encoding of CSP to SAT, using Chaff which does not employ the lookahead mechanism. In his tests a similar shape to the graph was reported. This indicates that the support encoding has a similar improvement in running time over the direct encoding in both the SAT and ASP encodings, with or without lookahead.

We also decided to measure the amount of searching that both the direct and support encodings must do in order to arrive at a solution. This is measured in Smodels via the number of picked atoms reported. This number represents the number of atoms that the lookahead function must pick a value for and then test to see if any conflicts occur. It is a good (though implicit) measure of the amount of search that occurs.

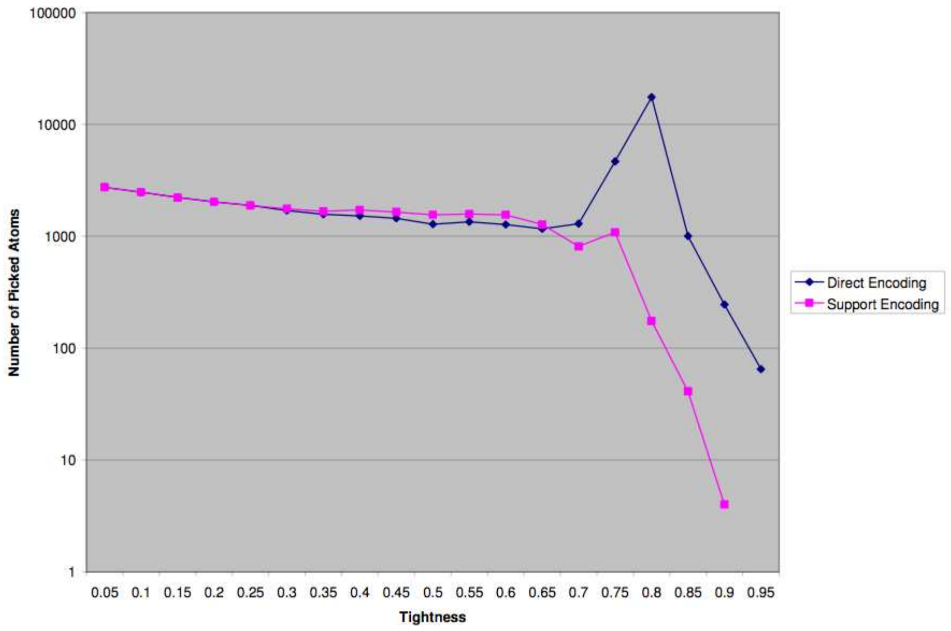

Fig. 4. Picked atoms vs. tightness.

Figure 4 shows the number of picked atoms plotted against the tightness of the problem set. The problem set is identical to the one used for figure 3. For lower values of tightness, the number of picked atoms is almost identical in both the direct and support encoding. Similar to Figure 3, as the problem set begins to enter its phase transition, the number of picked atoms between the two encodings begins to diverge. The support encoding clearly picks less atoms as the problem set becomes tighter.

8 Conclusion

We summarize below the main findings reported in this paper.

(1) Lookahead has the same pruning power under the direct encoding as well as under Niemelä's encoding (Theorem 1 and [16]).

(2) When testing $(i-1)$-tuples, lookahead captures i-consistency under the direct encoding (Theorem 2).

(3) Under the support encoding, lookahead coincides with SAC (Theorem 4).

(4) Lookahead performs more effectively under the support encoding than under the direct encoding (Theorems 1, 4 and 3). Experimentally, the former shows a substantial improvement in running time than the latter on hard random binary CSPs.

(5) When testing pairs of literals, lookahead captures singleton restricted path consistency under the support encoding (Theorem 5).

(6) Lookahead provides an uniform algorithm for maintaining local consistencies of various kinds, in particular enforcing both SAC and PAC is of the complexity $O(n^3d^4 + en^2d^4)$. This is because the number of atoms in these encodings is nd, lookahead may be called at most $O(n^2d^2)$ times, and each time expand takes time linear in the size of the program (which is bounded by $O(nd^2 + ed^2)$). It is interesting to see that this complexity is comparable to that of the algorithm specifically designed for SAC in [5].

Acknowledgment: We thank the referees for their comments, which have been useful in improving the presentation of the paper.

References

1. C. Bessière. Uniform random binary csp generator. Retrieved December 19, 2004, Website `www.lirmm.fr/~bessiere/generator.html`.
2. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. In *Proc. ECAI'04 Workshop on Modeling and Solving Problems with Constraints*, pages 20–29, 2004.
3. C. Bessière, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing*, pages 400–407, 2003.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
5. R. Debruyne and C. Bessière. Some practical filtering techniques. In *Proc. IJCAI'97*, pages 412–417, 1997.
6. R. Debruyne and C. Bessière. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.
7. Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels with CLP and its applications: a simple and effective approach to aggregates in ASP. In *Proc. ICLP'04*, pages 73–89, 2004.
8. J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.
10. Ian Gent. Arc consistency in SAT. In *Proc. ECAI 2003*, pages 121–125, 2002.
11. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, pages 275–286, 1990.
12. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artificial Intelligence*, 25(3-4):241–273, 1999.
13. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proc. CP'00*, pages 353–368, 2000.
14. P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.
15. T. Walsh. CSP vs. SAT. In *Proc. Principles and Practice of Constraint Programming*, pages 441–456, 2000.
16. J. You and G. Hou. Arc consistency + unit propagation = lookahead. In *Proc. ICLP'04*, pages 314–328, 2004.