University of Alberta

RELATING CONSTRAINT PROPAGATION TECHNIQUES IN CSP WITH ANSWER SET PROGRAMMING

by

Guiwen Hou ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95767-5
Our file Notre référence
ISBN: 0-612-95767-5

Acknowledgements

Thanks to my supervisor Prof. You for providing guidance and advice not only on technical aspects but also on effective writing.

I am grateful to the members of my examining committee for their valuable comments, suggestions and time spent reading the thesis. In particular, thanks to Dr. Martin Müller for reading the details of all chapters, and helping me to improve the presentation of this thesis.

Special thanks to all my friends for the various forms of help and encouragement that they have given me.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Contributions 4
1.3 Thesis Layout 4

2 Constraint Satisfaction Problem **5**
2.1 Definitions 5
2.2 Arc-Consistency 6
2.3 Path Consistency 12
2.4 General Search Strategies for CSPs 12

3 Logic Programming with Stable Model Semantics **19**
3.1 Introduction 19
3.2 Stable Model for Normal Logic Program 20
3.3 An Example Program 22
3.4 Constraint Propagation in Smodels 23

4 Arc-Consistency Vs. Lookahead **31**
4.1 Translation from CSPs to Logic Programs 31
4.2 AC^+ Vs. Expand and Lookahead 33

5 Arc-Consistency with Unit Propagation **36**
5.1 Definition 36
5.2 An Algorithm for ACUP 38
5.3 The Correctness of AC4-UP 42
5.4 Complexity 46

6 Further Observations **47**
6.1 ACUP Vs. Arc-Consistency Look Ahead 47
6.2 ACUP Vs. Path Consistency and i Consistency 51

7 Summary and Future Work **53**
7.1 Summary 53
7.2 Future Work 54

List of Figures

2.1 Algorithm of *revise* 7
2.2 Algorithm of AC-1 8
2.3 Algorithm of AC-3 9
2.4 Algorithm of AC-4 10
2.5 A general lookahead algorithm from [2] 14
2.6 SELECT-VALUE subprocedure for forward-checking 16
2.7 SELECT-VALUE subprocedure for arc-consistency look ahead 18

3.1 Algorithm of *smodels* 24
3.2 Function expand(P, A) 24
3.3 Function lookahead(P, A) 29

5.1 An algorithm for ACUP based on AC-4 40
5.2 *unit_propagate** algorithm 41

6.1 SVAC$^+$ subprocedure 50

Chapter 1

Introduction

A constraint problem is a problem where a solution to the problem requires the satisfaction of possibly many constraints (conditions, properties) simultaneously. In general, constraint problems are NP-complete. That is, it is unlikely to design an algorithm that can scale efficiently with the problem size, in all the cases.

Constraint programming (CP) is a programming paradigm that can be used to study and solve constraint problems. In this framework, the end user writes a program for a constraint problem, stating what has to be solved instead of how it is solved. Constraint programming is one of the most exciting developments in programming languages in the last decade. It has been successfully applied for solving many practical problems, such as scheduling and planning.

There are two frameworks for CP that are studied intensively by researchers in artificial intelligence. One is constraint satisfaction problem (CSP), and the other boolean satisfiability (SAT).

A CSP can be expressed by a collection of constraints over a finite set of variables, where each variable has a finite domain. A solution to a CSP is an assignment to each variable of a value from its domain such that all the constraints are satisfied. A CSP may have one, many, or no solutions.

In SAT, a constraint problem is represented by a set of clauses. To find a solution for a SAT instance is to find an assignment for the boolean variables that makes all the clauses satisfied.

In solving a constraint problem, it is common to apply some constraint propagation techniques to prune the search space. *Arc-consistency* is the most popular propagation technique in CSP solvers, while *lookahead* appears to be its counterpart in SAT solvers.

Although CSP solvers and SAT solvers were developed largely in parallel, the relation between arc-consistency and lookahead has been open for some time. Recently, [28] shows that arc-consistency for CSPs is weaker than lookahead in SAT in pruning the search space. A new method called AC^+ is formalized to enhance arc-consistency to achieve the same pruning power.

In this thesis, we review the work given in [28]. We further design an algorithm for AC^+ and perform some further comparisons with other consistency techniques in the literature.

1.1 Motivation

A CSP can be solved by a straightforward method called *generating* and *testing*. First, it generates a complete instantiation of all variables, and then tests whether this instantiation satisfies all the constraints. If it does, then a solution is found, otherwise another instantiation is generated and tested until a solution is found or all instantiations have been generated and tested. Obviously, this algorithm is not very efficient.

The most common algorithm for solving CSPs is based on backtracking. The idea for a backtracking algorithm is to extend a partial solution toward a complete solution consistently. It assigns a value from its domain to a variable, and tests whether this assignment is consistent with the already assigned variables. If it does, then this process continues, otherwise the algorithm chooses an alternative value for this variable. If no value can be assigned to this variable consistently, the algorithm backtracks to the proceeding variable.

A backtracking algorithm can maintain a certain kind of consistency for the underlying CSP in order to prune the search space. Consistency techniques can remove the inconsistent values from the domains in a CSP. As a result, the search space is reduced. A general notion of consistency is called *k-consistency* which requires that any partial solution for any $k-1$ variables be consistently extended to a partial solution with any additional variable [6, 15]. The most popular case of k-consistency is *arc-consistency* (when $k = 2$). Consistency techniques can be applied *before* search, as well as *during* search.

A SAT instance can also be solved by using a backtracking search algorithm. The well known algorithm is Davis-Putnam-Logemann-Lovehand (DPLL) procedure [1]. A common technique for space pruning is *lookahead.* Lookahead tentatively assigns a value for each variable, and checks whether this assignment leads to a contradiction. If it does, lookahead then assigns the opposite value for this variable (cf. [5]). In this way, the variable gets a value propagated from already assigned variables without going through a search process. In order to derive a contradiction, constraint propagation techniques are used. *Unit propagation* is a widely used constraint propagation technique in SAT solvers.

Answer set programming is a newly developed programming paradigm for solving constraint problems, where a constraint problem is represented by a logic program specifying the constraints that must be satisfied. A model for an answer set program corresponds to a solution to the problem being solved. A program may have zero, one or more models. Logic programming based on the stable semantics [9] yields an answer set programming system. Recent implementations of the stable model semantics include Smodels [19, 20, 24] and DLV [4]. Technically, answer set programming can be viewed as a variant of SAT.

Although arc-consistency and lookahead are both used to prune the search space in solving constraint problems, the relation between lookahead in SAT solvers and arc-consistency in CSP solvers remains open for some time. Recently, Walsh [27] compared the impact on achieving arc-consistency on CSPs with unit propagation on SAT problems. Gent [10], in extending an idea from Kasif [14], shows that the "support encoding" of binary CSPs into SAT instances is able to achieve arc-consistency by unit propagation.

In [28], lookahead is related to arc-consistency under some assumptions. Firstly, we fix the encoding from CSPs to answer set programs. Secondly, we choose the *lookahead* algorithm in Smodels for comparison. Finally, we only deal with binary constraints. Under these assumptions, we showed that lookahead is more powerful than arc-consistency in pruning the search space. This insight enables us to identify what is missing in arc-consistency. Unique value propagation, or just *unit propagation*, a terminology borrowed from SAT, is identified to make up the gap. Unit propagation for CSPs assigns a value for a variable if there is a unique value in its domain that is consistent with the partial instantiation. This process continues until no value can be assigned to a variable or a conflict occurs. A new method called AC^+ that combines arc-consistency and node consistency with unit propagation is formalized. AC^+ for CSPs prunes the same space as lookahead in Smodels. We give a worst case time complexity of lookahead for the translated programs from CSPs. Indirectly, a bound for the time complexity of AC^+ is obtained for CSPs.

In this thesis, we extend the work in [28]. Based on the result given in [28], lookahead can be viewed as a providing an algorithm for AC^+ indirectly. As lookahead is a general algorithm, it is worthwhile to investigate whether there exists more efficient algorithms for AC^+. In this thesis, we design an algorithm for a restricted version of AC^+ called *arc-consistency with unit propagation* (ACUP) for CSPs. In ACUP, node consistency is removed, since it can be processed separately. We prove the correctness of this algorithm and give its time complexity. The complexity bound of this algorithm is lower than the one given in [28]. Therefore, the algorithm provided in this thesis is more efficient. Furthermore, we compare the pruning power for ACUP with that of *arc-consistency look ahead* and *path consistency* [2]. We show that for some CSPs, ACUP is stronger than arc-consistency look ahead, while for some others, ACUP is weaker. The same conclusion is applied to the relation between ACUP and path consistency. Although *i-consistency* is very powerful in pruning search space, and has a high time complexity ($O(2^i(ek)^{2i})$ for a brute-force algorithm [2]), we show that for some CSPs it still cannot find the conflict that can be found by ACUP.

1.2 Contributions

The contributions of this thesis are as follows:

1. We perform a review on the constraint propagation techniques between CSPs and answer set programming.
2. An algorithm for ACUP is presented. The correctness of the algorithm is proved, and the complexity of the algorithm is given. The complexity result implies that our algorithm is more efficient than lookahead.
3. We also compare ACUP with consistency techniques in the literature. We show that the pruning power for arc-consistency with unit propagation overlaps with that of arc-consistency look ahead and path consistency. A revised version of the arc-consistency look ahead algorithm is given to achieve the extra pruning power.

1.3 Thesis Layout

This thesis is organized as follows. The next chapter introduces the definitions of CSPs, arc-consistency, and path consistency. We also discuss some general search strategies in the literature. Chapter 3 provides some concepts in logic programming with the stable model semantics, and the algorithm of Smodels. Lookahead is also discussed. In Chapter 4, we review some results in [28]. Firstly, Niemelä's translation is introduced. Under this translation, it has been shown that lookahead is strictly stronger than arc-consistency. Then, in Chapter 5, we introduce the result in [28], showing that arc-consistency plus node consistency with unit propagation (AC^+) prunes exactly the same space as lookahead. Also in this chapter, we present an algorithm for ACUP, an alternative version of AC^+, which only combines arc-consistency with unit propagation. Its correctness is proved and complexity is analyzed. Furthermore, in Chapter 6, we compare ACUP with other consistency techniques in the literature. We show that the pruning power of ACUP neither dominate nor is dominated by path consistency and arc-consistency look ahead (ACLA) in the literature. ACLP is revised to incorporate the idea in ACUP to prune more search space. Chapter 7 concludes our work, and gives some comments on future research.

Chapter 2

Constraint Satisfaction Problem

2.1 Definitions

A constraint satisfaction problem or CSP is a triple $\mathcal{A}(X, D, C)$, where C is a finite set of constrains $\{c_1, c_2, \ldots, c_n\}$ over a finite set of variables $X = \{x_1, x_2, \ldots, x_m\}$ and a domain $D = \{D_{x_1}, D_{x_2}, \ldots, D_{x_m}\}$ that maps each variable $x_i \in X$ to a finite set of values, written D_{x_i}.

A constraint c_i is a relation R_i defined on a subset of variables $S_i \subseteq X$. S_i is called the scope of c_i. If $S_i = \{x_{i_1}, \ldots, x_{i_m}\}$, then R_i is a subset of the Cartesian product $D_{x_{i_1}} \times \ldots \times D_{x_{i_m}}$.

The *arity* of a constraint refers to the size of its scope. A *unary* constraint is defined on a single variable; a *binary* constraint is defined on two variables; an i-ary constraint is defined on i variables.

Instantiation

An instantiation of a set of variables is an assignment of each variable in the set with a value from its domain. Formally, in a CSP $\mathcal{A}(X, D, C)$, an instantiation of a set of variables $\{x_{i_1}, \ldots, x_{i_m}\}$ is a set of pairs $\{\langle x_{i_1}, a_{i_1}\rangle, \ldots, \langle x_{i_m}, a_{i_m}\rangle\}$, where $\{x_{i_1}, \ldots, x_{i_m}\} \subseteq X$, $a_{i_k} \subseteq D_{x_{i_k}}$. We denote by $x \rightarrow a$ or $\langle x, a\rangle$ that variable x is assigned value $a \in D_x$.

Satisfaction of a constraint

A constraint c_i is *satisfied* by an instantiation if and only if the instantiation to variables in the scope of c_i yields a tuple in the relation R_i.

Consistent Partial Instantiation

A partial instantiation is *consistent* with constraint c_i if and only if the assignment yields a projection of a tuple in the relation of R_i. A partial instantiation is consistent if and only if it is consistent with every constraint. A consistent partial instantiation is also called a *partial solution.*

Solution

A *solution* to a CSP $\mathcal{A}(X, D, C)$ is an instantiation of all its variables $X = \{x_1, x_2, \ldots, x_m\}$, where each variable x_i is assigned a value from its domain D_{x_i} such that all the constraints in $C = \{c_1, c_2, \ldots, c_n\}$ are satisfied.

Example 2.1.1 *Let a* CSP *be* $\mathcal{A}(X, D, C)$, *where* $X = \{x, y, z\}$, $D_x = D_y = D_z = \{0, 1, 2\}$, *and* $c_{xy} = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,2\rangle\}, c_{yz} = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,2\rangle\}$.

Then $\{x = 0, y = 1, z = 2\}$ is a solution for $\mathcal{A}$. ■

Two CSPs are said to be *equivalent* if and only if they have the same set of solutions.

2.2 Arc-Consistency

A general notion of consistency is called *k-consistency*, which require a partial solution with an assignment of $k - 1$ variables to be consistently extended to a partial solution with an assignment of an additional variable. When $k = 1$, it is called node consistency; when $k = 2$, it is called arc-consistency; when $k = 3$, it is called path consistency.

Now we give the formal definition of arc-consistency as follows.

Definition 2.2.1 *Given a* CSP $\mathcal{A}(X, D, C)$, *a constraint* $c \in C$ *over* $\{x, y\}$, *where* $x, y \in X$, x *is arc-consistent with respect to* y *over* c *if and only if for every assignment* $x \to a \in D_x$, *there is a corresponding assignment* $y \to b$, *where* $b \in D_y$ *such that* $x \to a$ *and* $y \to b$ *satisfy* c.

A binary constraint whose scope is $\{x, y\}$ is arc-consistent if x is arc-consistent with respect to y and y is arc-consistent with respect to x. A CSP is arc-consistent if all of its constraints are arc-consistent.

Example 2.2.2 *Consider a* CSP *as described by the constraints* $\{x < y,\ y < z,\ z \leq 3\}$, *where* $D_x = D_y = \{1, 2, 3\}$ *and* $D_z = \{2, 3, 4\}$.

Enforcing node consistency reduces D_z to $D'_z = \{2, 3\}$. Enforcing arc-consistency reduces D_x to $D'_x = \{1, 2\}$, as for $x = 3$ there is no value for y such that the constraint is satisfied. Similarly, D_y is reduced to $D'_y = \{2, 3\}$. Working on the constraint $y < z$ for variable y, D'_y is further reduced to 2, which forces

```
revise
Input: two variable $x_i, x_j$ in a certain constraint $c_{ij}$, and their respective domains $D_{x_i}, D_{x_j}$
Output: $D_{x_i}$ such that $x_i$ is arc-consistent with $x_j$
for each $a_i \in D_{x_i}$ do
    if there is no value $a_j \in D_{x_j}$ such that $(a_i, a_j) \in c_{ij}$ then
        remove $a_i$ from $D_{x_i}$;
    end
end
```

Figure 2.1: Algorithm of *revise*

D'_x to become $\{1\}$ by the first constraint, and D'_z to become $\{3\}$ by the last constraint. By now, no further reduction is possible. ■

Many specific algorithms that enforce arc-consistency for binary constraints have been developed (cf. [2, 18]), such as AC-1, AC-3, AC-4. An essential subprocedure used in arc-consistency is *revise* [2] (cf. Figure 2.1). $revise(x_i, x_j)$ tests every value a in D_{x_i} to see if there is a support value in D_{x_j}. If not, then a is removed from domain D_{x_i}.

Since each value in D_{x_i} is tested with each value in D_{x_j}, *revise* has the time complexity $O(k^2)$, where k bounds the domain size.

Algorithm AC-1 given in Figure 2.2 is a brute-force algorithm that enforces arc-consistency of a CSP. It applies *revise* to all variable pairs that are in the scope of the constraints, until no further domain value can be removed.

Let e be the number of constraints, and k a bound on the domain size. Since the loop of part 1 in AC-1 takes $O(ek^2)$ time, and in the worst case, one iteration may cause the deletion of just one value from one domain, we conclude:

Proposition 2.2.3 [2] *The time complexity of AC-1 is $O(e^2k^3)$.*

One can see that AC-1 is a naive algorithm which processes all the constraints even if only a few values are removed in the previous rounds. It can be improved to check only the affected constraints.

In AC-3 (cf. Figure 2.3), a *queue* is established initially to store the pairs of variables in the scopes of all the constraints. In each round, a pair $\langle x_i, x_j \rangle$ is removed from the queue, and the inconsistent values in D_{x_i} are removed by $revise(x_i, x_j)$. If $revise(x_i, x_j)$ causes a removal in D_{x_i}, then all the related

```
AC-1
Input: a constraint $\mathcal{A}(X, D, C)$
Output: a consistent CSP equivalent to $\mathcal{A}$
repeat
1   for each $c_{ij} \in C$ do
        for each $\langle x_i, x_j \rangle$ where $x_i$ and $x_j$ in the scope of $c_{ij}$ do
            $revise(x_i, x_j)$;
            $revise(x_j, x_i)$;
        end
    end
until no domain is changed;
```

Figure 2.2: Algorithm of AC-1

pairs to x_i are added to the queue. This procedure continues until the queue becomes empty.

Example 2.2.4 *Consider a* CSP *that includes three variable x, y, z, where the domains of all variables are $D_x = D_y = D_z = \{0, 1, 2\}$, and the constraints are $c_{xy} = \{\langle 0, 1\rangle, \langle 0, 2\rangle, \langle 1, 2\rangle\}, c_{yz} = \{\langle 0, 1\rangle, \langle 0, 2\rangle, \langle 1, 2\rangle\}, c_{zx} = \{\langle 2, 1\rangle, \langle 1, 2\rangle\}$*

Now we show how AC-3 works on this CSP. Initially, $\langle x, y\rangle, \langle y, z\rangle$, and $\langle z, x\rangle$ are put into the queue. Then $\langle x, y\rangle$ is removed from the queue. Applying $revise(x, y)$, 2 is removed from D_x. Due to $\langle z, x\rangle$ in the queue already, we don't need to add $\langle z, x\rangle$ to the queue again. Then $\langle y, z\rangle$ is removed from the queue. Since 2 is removed from D_y, $\langle x, y\rangle$ is added to the queue again. At this stage, $\langle z, x\rangle$ is removed from the queue. As a result, 0 and 1 are removed from D_z, and $\langle y, z\rangle$ is added to the queue again. In the following round, 2 is removed from D_x, and 1 removed from D_y. From now on, no further domain value can be removed, so no new pair is added to the queue. Therefore, the algorithm stops. Finally, we get $D_x = \{0\}, D_y = \{1\}, D_z = \{2\}$. ■

As each time when *revise* causes a change, there is at least one value that is removed from the domain, so the size of queue is at most $O(ek)$, where k bounds the domain size, and e is the number of constraints. We already know the time complexity of *revise* is $O(k^2)$, we therefore have the following proposition:

Proposition 2.2.5 [2] *The time complexity of AC-3 is $O(ek^3)$.*

```
AC-3
Input: a constraint $\mathcal{A}(X, D, C)$
Output: a consistent CSP equivalent to $\mathcal{A}$
for each $c_{ij} \in C$ do
    for every pair $\langle x_i, x_j \rangle$ with $x_i$ and $x_j$ are in the scope of $c_{ij}$ do
        $queue \leftarrow queue \cup \{\langle x_i, x_j \rangle,\ \langle x_j, x_i \rangle\}$;
    end
end
while $queue \neq \emptyset$ do
    select and delete $\langle x_i, x_j \rangle$ from queue;
    $revise(x_i, x_j)$;
    if $revise(x_i, x_j)$ causes a change in $D_{x_i}$ then
        $queue \leftarrow queue \cup \{\langle x_k, x_i \rangle, k \neq i, k \neq j\}$;
    end
end
```

Figure 2.3: Algorithm of AC-3

In the worst case, checking arc-consistency of a general CSP takes $O(ek^2)$ time (checking each constraint takes $O(k^2)$, and there are e constraints), so there is no algorithm that can have a time complexity lower than $O(ek^2)$.

AC-4 achieves this optimal time complexity by utilizing an efficient structure to store the relations in a CSP. Algorithm AC-4 is given in Figure 2.4.

AC-4 associates each assignment $\langle x_i, a_i \rangle$ with the amount of support from neighboring variable x_j, that is, the number of values in the domain of x_j that are consistent with the assignment $\langle x_i, a_i \rangle$. If variables x and y are in the same scope of a certain constraint, then x is the neighboring variable with respect to y, and y is the neighboring variable with respect to x. A value a_i is then removed from the domain D_i if it has no support from some neighboring variables. AC-4 uses a counter array, *counter*(x_i, a_i, x_j), to store the number of support values from neighboring variables for the assignment of $\langle x_i, a_i \rangle$. An array $S(x_j, a_j)$ is used to store all the values in the other variables supported by $\langle x_j, a_j \rangle$. $\mathcal{L}$ is used to store the unsupported values.

In each step, the algorithm picks up an unsupported value from $\mathcal{L}$, and updates all the affected counters. The counters that become zero as a result are placed in $\mathcal{L}$. This process continues until all the unsupported values are removed.

Example 2.2.6 *Consider a three variable* CSP*: x, y, z with* $D_x = \{1, 2\}, D_y =$

```
AC-4
Input: a CSP $\mathcal{A}(X, D, C)$
Output: a consistent CSP equivalent to $\mathcal{A}$
initialize $S(x_i, a_i)$, $counter(x_i, a_i, x_j)$ from all $c_{ij} \in C$;
for all counters do
    if $counter(x_i, a_i, x_j)$=0 (if $\langle x_i, a_i \rangle$ is unsupported by $x_j$) then
        add $\langle x_i, a_i \rangle$ to $\mathcal{L}$;
    end
end
while $\mathcal{L}$ is not empty do
    choose and remove $\langle x_i, a_i \rangle$ from $\mathcal{L}$, remove $a_i$ from $D_{x_i}$;
    for each $\langle x_j, a_j \rangle \in S(x_i, a_i)$ do
        decrement $counter(x_j, a_j, x_i)$;
        if $counter(x_j, a_j, x_i) = 0$ then
            add $\langle x_j, a_j \rangle$ to $\mathcal{L}$;
        end
    end
end
```

Figure 2.4: Algorithm of AC-4

$\{3, 4\}, D_z = \{0, 1\}$. *There are two constraints:* $c_{xy} = \{\langle 1,3\rangle \langle 1,4\rangle \langle 2,4\rangle\}$, $c_{yz} = \{\langle 3,0\rangle \langle 3,1\rangle\}$.

Initializing the $S(x, a)$ arrays, we have

$$\begin{aligned}
&S(x,1) = \{\langle y,3\rangle, \langle y,4\rangle\}\\
&S(x,2) = \{\langle y,4\rangle\}\\
&S(y,3) = \{\langle x,1\rangle \langle z,0\rangle \langle z,1\rangle\}\\
&S(y,4) = \{\langle x,1\rangle \langle x,2\rangle\}\\
&S(z,0) = \{\langle y,3\rangle\}\\
&S(z,1) = \{\langle y,3\rangle\}
\end{aligned}$$

For counters, we have

$$\begin{aligned}
\mathit{counter}(x,1,y) &= 2\\
\mathit{counter}(x,2,y) &= 1,\\
\mathit{counter}(y,3,x) &= 1,\\
\mathit{counter}(y,3,z) &= 2,\\
\mathit{counter}(y,4,x) &= 2,\\
\mathit{counter}(y,4,z) &= 0,\\
\mathit{counter}(z,0,y) &= 1,\\
\mathit{counter}(z,1,y) &= 1.
\end{aligned}$$

We don't need to add counters between variables that are not directly constrained, such as between x and z. First $\mathcal{L} = \{\langle y,4\rangle\}$ due to $\mathit{counter}(y,4,z) = 0$, which means there is no support for the assignment $y \to 4$. In the first iteration, $\langle y,4\rangle$ is removed from $\mathcal{L}$, and value 4 is removed from domain D_y. Then we decrease affected counters, which are $\mathit{counter}(x,1,y)$ and $\mathit{counter}(x,2,y)$. Due to $\mathit{counter}(x,2,y) = 0$, $\langle x,2\rangle$ is added to $\mathcal{L}$. In the next iteration, $\langle x,2\rangle$ is removed from $\mathcal{L}$, and 2 is removed from D_x. Then we decrease affected counters again, and set $\mathit{counter}(y,4,x) = 1$. Since no new counter becomes zero, no new pair is added to $\mathcal{L}$. Consequently $\mathcal{L}$ remains empty and the algorithm stops. Finally, the domains change to $D_x = \{1\}, D_y = \{3\}, D_z = \{0,1\}$. ■

Note that the number of elements in $\sum_{i,j} S(x_i, a_j)$ is twice the number of tuples in the constraints, also $\sum_{i,j,k} \mathit{counter}(x_i, a_j, x_k)$ is equal to twice the number of tuples in the constraints, which is $O(ek^2)$. So the initialization step that establishes the counters of supports and the pointers to the supports requires $O(ek^2)$ time and space. In the *while* loop of the algorithm, each iteration decreases at least one counter by one, so there are at most $O(ek^2)$ iterations. We therefore conclude [2]:

Proposition 2.2.7 *The time complexity and space complexity of AC-4 is* $O(ek^2)$.

2.3 Path Consistency

Definition 2.3.1 *Given a* CSP $\mathcal{A}(X, D, C)$*, a two variable set* $\langle x_i, x_j \rangle$ *is path consistent relative to* x_k *iff for every consistent assignment* $\{\langle x_i, a_i \rangle \langle x_j, a_j \rangle\}$*, there is a value* $a_k \in D_{x_k}$ *such that the assignment* $\{\langle x_i, a_i \rangle \langle x_k, a_k \rangle\}$ *is consistent and* $\{\langle x_k, a_k \rangle \langle x_j, a_j \rangle\}$ *is consistent* [2]. *A binary constraint* c_{ij} *is path consistent relative to* x_k *if and only if for every pair* $\langle a_i, a_j \rangle \in c_{ij}$*, there is a value* a_k *in* x_k*, which is a neighbor of both* x_i *and* x_j*, such that* $\langle a_i, a_k \rangle \in c_{ik}$ *and* $\langle a_k, a_j \rangle \in c_{kj}$*. A* CSP *is path consistent if and only if for every* $c_{ij} \in C$*, and for every* $x_k, k \neq i, k \neq j$*, where* x_k *is the neighbored variable to both* x_i *and* x_j*,* c_{ij} *is path consistent to* x_k*.*

Example 2.3.2 *Consider a* CSP *that has three constraints over three variables* $x, y,$ *and* z *with* $D_x = D_y = D_z = \{0, 1\}$*,* $c_{xy} = \{\langle 0, 1 \rangle \langle 1, 0 \rangle\}$*,* $c_{xz} = \{\langle 0, 1 \rangle \langle 1, 0 \rangle\}$*, and* $c_{zy} = \{\langle 0, 1 \rangle \langle 1, 0 \rangle\}$*.*

Obviously, this CSP is arc-consistent. However, consider $\langle 0, 1 \rangle \in c_{xy}$, there doesn't exist a value a for z such that $\langle 0, a \rangle \in c_{xz}$ and $\langle a, 1 \rangle \in c_{zy}$ simultaneously, so the CSP is not path consistent. ■

Enforcing arc-consistency makes a CSP consistent between two variables, while enforcing path consistency makes a CSP consistent among three variables. A general notation of consistency can be defined based on the consistency among i variables.

Definition 2.3.3 *Given a* CSP $\mathcal{A}(X, D, C)$*, if for any consistent instantiation of any* $i-1$ *distinct 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* [2]*, then* A *is i-consistent. If* $i = 2$*, it is called arc-consistent. If* $i = 3$*, it is called path consistent.*

Example 2.3.4 *Consider a* CSP *that has four variables* x_1, x_2, x_3, x_4 *with* $D_{x_1} = D_{x_2} = D_{x_3} = D_{x_4} = \{0, 1, 2\}$*, and four constraints* $x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4, x_4 \neq x_1$*. We can see that this* CSP *is both arc and path consistent, but not 4-consistent.* ■

2.4 General Search Strategies for CSPs

Constraint propagation can reduce the search space in computing a solution of a given CSP. But for most CSPs, we are still left with choices to reach a solution by only employing constraint propagation. The only way to proceed is *guessing* and *testing*. That is, we must search the space of possible choices. The naive algorithm for performing systematic search is chronological backtracking. It incrementally attempts to extend a partial solution toward a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. If all the values in the domain of a variable have been tried and fail to give a consistent instantiation, a *dead-end* occurs. At this point, *backtracking* takes place, and the preceding variable in the partial instantiation becomes the current variable. The search continues until the required number of solutions has been found or no variable can be instantiated.

Chronological backtracking algorithm is not efficient. In general, it has three disadvantages.

1. One is *thrashing* [7], i.e., repeated failure due to the same reason. Thrashing takes place because the chronological backtracking algorithm does not identify the real reason of the conflict, i.e., the conflicting variables. As a result, search in different parts of the space without identifying the real conflict keeps failing for the same reason. Thrashing can be avoided by *look back* techniques, by which the algorithm backtracks directly to the source of failure, instead of backtracking to the preceding variable in the partial instantiation.

2. Another disadvantage is that it always chooses a fixed order of instantiating the variables. A CSP may have different search spaces on different variable orders. When a good order is chosen, the conflict can be found and avoided early by the algorithm. There are several strategies in the literature to choose an order in which the variables are instantiated.

3. Finally, the chronological backtracking algorithm cannot detect the conflict before it really occurs. This drawback can be avoided by applying *look ahead* technique, which can be invoked when the algorithm is preparing to assign a value to the current variable. It applies consistency techniques to forward check the possible conflicts, and chooses a consistent value for the current variable.

In this thesis, as we compare the difference between look ahead techniques in the literature and arc-consistency with unit propagation, we only introduce look ahead strategies based on constraint propagations.

We now present a general look ahead algorithm in Figure 2.5 [2]. In this algorithm, different propagation methods can be embedded to form different algorithms such as forward checking [11], arc-consistency look ahead or maintain-arc-consistency [7]. For this reason, we write a SELECT-VALUE-XXX subprocedure to indicate the specific propagation method embedded in the general algorithm by replacing XXX.

Initially, this general algorithm uses a tentative domain D' to store the original domain, and the first variable is the current variable. The SELECT-

```
Backtracking
Input: a CSP $\mathcal{A}(X, D, C)$
Output: either a solution or notification that no solution can be found
$D' = D$;
$i = 1$;
while $1 \le i \le n$ do
1   instantiate $x_i \leftarrow$ SELECT-VALUE-XXX;
    if $x_i$ is null    /* no value was returned*/  then
2       $i = i - 1$    /*backtrack*/ ;
        reset each $D'_k$ to $D_k, k > i$, to its value before $x_i$ was last
        instantiated;
    else
        $i = i + 1$    /* step forward*/;
    end
end
if $i = 0$ then
    return "no solution";
else
    return instantiated values of $(x_1, \ldots, x_n)$;
end
```

Figure 2.5: A general lookahead algorithm from [2]

VALUE-XXX procedure propagates the partial instantiation to remove inconsistent values from D'. The idea here is to reduce the domains of the unassigned variables by maintaining certain level of consistency before committing to a choice. The reduction to an empty domain causes the algorithm to backtrack. At this stage, different propagation methods can be applied. Upon backtracking, D' is reset to its original set in order to give up the modifications that are caused by the current instantiation.

Forward Checking

In the literature, it is often felt that arc-consistency is too expensive to be beneficial in real applications. As a result, a restricted version, called *forward checking* (FC), where values from the domains of future variables are filtered out if they are inconsistent with the current instantiation, is sometimes preferred [28].

The forward checking algorithm is given in Figure 2.6 [2]. It enforces a partial consistency check in the intermediate stage of the general look ahead algorithm. It assigns a tentative value for the current variable, then performs arc-consistency check between the partial instantiation and the future variables. The values that are inconsistent with the current partial value assignment are removed from the domain of each future variable. If the domain of any future variable becomes empty, it chooses another value for the current variable, enforces an arc consistency check, and resets the domains for all the future variables. If all the values in the domain of current variable have been checked, and no value can be chosen, it resets the domains and backtracks to the previous instantiated variable.

Example 2.4.1 *Consider the following* CSP *with four variables* $x_1, x_2, x_3,$ *and* x_4, *and respective domains* $D_{x_1} = \{0,1\}, D_{x_2} = \{0,1,2\}, D_{x_3} = \{0,1\}$, $D_{x_4} = \{0,1\}$. *Suppose there are six constraints :*

$$\begin{aligned}
c_{x_1,x_2} &= \{\langle 1,2\rangle, \langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle\} \\
c_{x_1,x_3} &= \{\langle 0,0\rangle\} \\
c_{x_1,x_4} &= \{\langle 0,1\rangle\} \\
c_{x_2,x_3} &= \{\langle 1,0\rangle, \langle 0,0\rangle\} \\
c_{x_2,x_4} &= \{\langle 1,0\rangle, \langle 0,1\rangle\} \\
c_{x_4,x_3} &= \{\langle 0,0\rangle\}
\end{aligned}$$

Suppose the order of instantiating variables is x_1, x_2, x_3, x_4. Initially, let's start at $x_1 \to 0$. By applying arc-consistency on the future variables x_2, x_3, and x_4, the value 1 is removed from D_{x_3}, and the value 0 from D_{x_4}. No value is removed from D_{x_2}. Now suppose $x_2 \to 0$. At this stage, arc-consistency is again applied to the future variables x_3, x_4, but no value can be reduced from D_{x_3}, D_{x_4}. Suppose 0 is chosen as the instantiated value for variable x_3.

```
SELECT-VALUE-FORWARD-CHECKING
Input: a CSP $\mathcal{A}(X, D', C)$, current variable $x_i$
Output: a value for the current variable, or a null if no consistent value
while $D'_i$ is not empty do
    select an $a \in D'_i$, and remove $a$ from $D'_i$;
    for each $k, i \leq k \leq n$ do
        for each value $b \in D'_k$ do
            if not consistent$(\overrightarrow{a_{i-1}}, x_i = a, x_k = b)$ then
                remove $b$ from $D'_k$;
            end
        end
        if $D'_k$ is empty then
            reset each $D'_k, i < k \leq n$ to value before $a$ was selected;
        end
        return a;
    end
end
return null;
```

Figure 2.6: SELECT-VALUE subprocedure for forward-checking

At this stage, consider the propagation on future variable x_4. All the values are removed from D_{x_4} because they are inconsistent with the instantiation $\{x_1 \rightarrow 0, x_2 \rightarrow 0, x_3 \rightarrow 0\}$. Because D_{x_4} is empty, backtracking happens. The algorithm will backtrack to the previous variable x_2, and consider the next value 1 in D_{x_2} for x_2, and the domain D_{x_4} is reset to $\{1\}$. Now $x_2 \rightarrow 1$ is then propagated to the future variables x_3 and x_4 again. We can see that 1 in D_{x_4} is inconsistent with the current partial instantiation $x_1 \rightarrow 0, x_2 \rightarrow 1$. The fact that 1 is removed from D_{x_4} causes D_{x_4} to become empty. So backtracking tries value 3 for x_2. Upon propagating this instantiation to x_3, and x_4, D_{x_3} is annihilated. Hence the algorithm will backtrack to the next value 1 for x_1, and all the propagation effects caused by $x \rightarrow 0$ are undone. At this stage, all the domains are thus back to their original form. Now the instantiation $x \rightarrow 1$ is propagated to the future variables x_2, x_3 and x_4. 0 and 1 are removed from D_{x_2} and all the values in D_{x_3} are removed due to inconsistency. Therefore, backtrack happens again, and no solution can be found. ■

Arc-Consistency Look Ahead

Experiments show that for problems with relatively tight constraints and relatively sparse constraint graphs, algorithms where future variables are checked against each other could substantially outperform forward checking [6]. This approach is called *arc-consistency look ahead.* In contrast to FC, arc-consistency look ahead performs full arc-consistency check between the current variable and future variables, and among all future variables. Clearly, arc-consistency look ahead does more work than FC. In an arc-consistency look ahead algorithm, full arc-consistency on all uninstantiated variables is enforced following each tentative value assignment to the current variable. If a variable's domain becomes annihilated during this process, the current candidate value is rejected, then it chooses another value for the current variable. If it is consistent (no domain becomes empty), the algorithm chooses the current candidate value for the current variable. If all the values in the domain of current variable are checked and no value can be chosen, it backtracks to the most recent instantiated variable. An algorithm of choosing a candidate value for the current variable is given in [2] (cf. Figure 2.7). it is called SELECT-VALUE-ARC-CONSISTENCY (SVAC).

When the current variable is instantiated to a value, SVAC discovers inconsistent values due to the instantiation using arc-consistency, and removes them. If the instantiation causes an empty domain, it chooses another value for the current variable, and all the removed values are undone. Otherwise, the current value is assigned to the current variable.

A popular variant of arc consistency look ahead is *maintaining arc-consistency,* which performs arc-consistency after each domain value is rejected. For example, suppose the current variable x has five values $\{1, 2, 3, 4, 5\}$. First, tenta-

```
SELECT-VALUE-ARC-CONSISTENCY
Input: a CSP $\mathcal{A}(X, D', C)$, current variable $x_i$
Output: a value for the current variable, or null if no consistent value
while $D'_i$ is not empty do
    select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$;
    make $\mathcal{A}$ arc-consistent with the partial instantiation.
    if any future domain is empty (don't select $a$) then
        reset each $D'_j, i < j \le n$, to its value before $a$ was selected;
    else
        return $a$;
    end
end
return null;
```

Figure 2.7: SELECT-VALUE subprocedure for arc-consistency look ahead

tively assign 1 to x, and apply a full arc-consistency check. If an empty domain occurs, then $x = 1$ is reject, and 1 is removed from domains $D_x = \{1, 2, 3, 4, 5\}$. Now we have $D_x = \{2, 3, 4, 5\}$, and apply full arc-consistency again with this reduced domain.

Chapter 3

Logic Programming with Stable Model Semantics

3.1 Introduction

Logic programming with stable model semantics has been developed as a viable constraint programming paradigm [16], where a constraint problem is expressed by a logic program, and the stable models for the program are the solutions to the constraint problem.

Normal logic program

A *normal logic program* is a set of rules of the form

$$h \leftarrow a_1 , ..., a_n , \textit{not } b_1, ..., \textit{not } b_m .$$

where h, a_1 , ..., a_n, b_1, ..., b_m are function-free atoms. In the above rule, atom h is called the head of the rule and the other literals make up the body. An expression such as *not b* is called a *not-atom*. Atoms and not-atoms are referred to as *literals*. Not-atoms are also called *negative* literals and atoms are called *positive* literals.

Positive logic progam

A logic program without not-atoms is called a *positive* program or a *negation free* program. A model of a logic program is an interpretation that satisfies all the rules in the logic program. A model is minimal if no proper subset is also a model. It is well-known that a positive program always has a unique minimal model [3]. The minimal model of a positive program can be computed by a fixpoint operator T_P defined as follows [13]:

$$T_P(M) = M \cup \{h \mid h \leftarrow a_1, ..., a_n \in P,\ \{a_1, ..., a_n\} \subseteq M\}$$

where M is a set of atoms and P is a positive program. The function T_P is monotonic since given sets of atoms S_1 and S_2, if $S_1 \subseteq S_2$ then $T_p(S_1) \subseteq T_p(S_2)$. The unique minimal model can be constructed by starting with an empty set and applying the T_P operator until a fixpoint is achieved as follows [13]:

$$T_p^{\uparrow 0} = \emptyset$$
$$T_p^{\uparrow(i+1)} = T_p(\ T_p^{\uparrow i}\ )\ when\ i \geq 0$$

When S is a fixpoint, $S = T_p(\ T_p^{\uparrow i}\ ) = T_p^{\uparrow(i)}$.

3.2 Stable Model for Normal Logic Program

The definition of a stable model for a normal logic program is based on the idea that one guesses a set of atoms in the program, and then tests whether it is a stable model.

Before giving the definition, we introduce *reduct* first.

Given a program P and a set of atoms M, the reduct P^M of P can be derived as follows [25]:

1. delete each rule in P that has a not-atom *not* x in its body such that $x \in M$, and
2. delete all not-atoms in the remaining rules.

Clearly, P^M is a positive program. Therefore P^M must have a unique minimal model.

Definition 3.2.1 [25] *Give a set of atoms M in program P, M is a stable model of P iif M is the least fixpoint of P^M, where P^M is the reduct of the normal logic program P with respect to the set of atoms M.*

We have defined the stable model semantics for normal logic programs based on guessing and testing. A naive algorithm that computes the stable models can be based on the definition. First, a set of atoms is guessed, then this set is tested whether it is a stable model or not by the definition. By using this naive approach, we can find all the stable models by enumerating all subsets of the atoms in a program and testing whether each subset is a stable model. There are 2^n subsets of atoms, where n is the number of atoms in a program.

The problem of determining whether a normal logic program has a stable model is NP-complete [17].

Example 3.2.2 *Consider the following program P:*

$a \leftarrow c,\ not\ \ b.$
$b \leftarrow not\ \ c.$
$c \leftarrow a,\ not\ \ b.$

One can check that $M = \{b\}$ is the only stable model for program P. The reduct of P^M, for example, is obtained as follows:

1. deleting the first and third rules from P to get $\{b\ \leftarrow\ \ not\ c.\ \}$, and

2. deleting the *not* c from the second rule.

We then get $P^M = \{b \leftarrow\}$. As there is only one rule with an empty body in the program, the minimal model is $\{b\}$, which is the same as M. So $M = \{b\}$ is a stable model of program P.

However, $M_1 = \{a\}$ is not a stable model, since $P^{M_1} = \{a \leftarrow c.\quad b.\quad c \leftarrow a.\}$, the minimal model of P^{M_1} is $\{b\}$, which does not coincide with M_1. Similarly, one can verify that all the other sets of atoms are not stable models. ■

All stable models are *justified* in the sense that every atom in a stable model has to have some reason to be there. That is, if an atom is in a stable model, there must be a rule with the atom as the head where all the body literals are satisfied.

A program may have no stable model. For example, let program P be

$a\ \leftarrow\ not\ a.$

There is no stable model for P. There are only two possible models: $\{a\}$ and $\emptyset$. If $\{a\}$ is a model, then the body of the only rule in program P, *not a*, is $false$, which does not support a. $\emptyset$ is not a model either, since the minimal model for the reduct $P^{\emptyset}$ is a.

A constraint in logic programs is of the following form:

$\leftarrow\ a_1\,, ..., a_n\,, not\ \ b_1, ...,\ \ not\ \ b_m.$

A model M satisfies the constraint if the body of the constraint is $false$ in M. If M is a stable model of a program, then M must satisfy all the constraints in the program.

Under the stable model semantics, the above constraint can be translated to the following rule:

$f\ \leftarrow\ not\ f,\ a_1,\ ...,\ a_n,\ not\ \ b_1, ...,\ not\ \ b_m.$

where f is a new unique atom in the program.

3.3 An Example Program

In this section, we will use a well-known problem, namely the colorability problem, to illustrate logic programming based on the stable model semantics.

The problem is, given facts about vertices, arcs and available colors, to find an assignment of colors to vertices such that each vertex has a color and any two vertices connected with an arc do not share the same color. The following is a logic program that solves the colorability problem based on Smodels [12].

Note, in general, we write function-free programs with variables. The stable models for a program with variables are the stable models for its instantiated, ground program.

Colorability

```
% rules to generate answer sets

color(V,C) ← vertex(V), col(C), not otherColor(V,C).
otherColor(V,C)← vertex(V), col(C), not color(V,C).

%utility rule

hasColor(V)← vertex(V), col(C), color(V,C).


%Constraints

← edge(V,V1), color(V,C), color(V1,C), V ≠ V1,
   vertex(V), vertex(V1), col(C).

← not hasColor(V), vertex(V).

← color(V,C), color(V,C1), col(C), col(C1), vertex(V).
```

In the above program, *col*, *vertex* and *edge* are domain predicates. A domain predicate restricts the range over which a variable can obtain values. The values a variable can obtain are the values that make domain predicates *true*. A domain predicate also restricts the instantiation of a rule with the predicate. In this program, rule

$$color(V,C) \leftarrow vertex(V),\ col(C),\ not\ otherColor(V,C).$$

can be instantiated to the following equivalent ground rules:

$$color(1,red) \leftarrow vertex(1),\ col(red),\ not\ otherColor(1,red).$$
$$color(1,green) \leftarrow vertex(1),\ col(green),\ not\ otherColor(1,green).$$
$$color(1,blue) \leftarrow vertex(1),\ col(blue),\ not\ otherColor(1,blue).$$

...
$color(5, red) \leftarrow\ vertex(5),\ col(red),\ not\ otherColor(5, red).$
$color(5, green) \leftarrow\ vertex(5),\ col(green),\ not\ otherColor(5, green).$
$color(5, blue) \leftarrow\ vertex(5),\ col(blue),\ not\ otherColor(5, blue).$

$color(V, C)$ means that vertex V gets color C. The first two rules in this program are used to generate all possible assignments of colors to vertices. If $color(V, C)$ is in a stable model, the second rule specifies that $otherColor(V, C)$ can not be in the same stable model. On the other hand, if $otherColor(V, C)$ is in a stable model, the first rule determines that $color(V, C)$ can not be in the same model. The auxiliary predicate $otherColor(V, C)$ is used to provide all assignments where $color(V, C)$ gets value *false*.

The fourth rule says that two vertices connected with an arc do not share the same color. The fifth rule specifies that every vertex must get at least one color. The last rule states that a vertex can only get one color.

3.4 Constraint Propagation in Smodels

In this thesis, we focus on the constraint propagation technique used in Smodels, as we want to compare it with arc-consistency for CSPs.

First, we give the following notations.

A set of literals is *consistent* if there is no atom a such that a and *not* a are both in the set, otherwise there is a *conflict* in the set. Given a set of literals A, $A^+ = \{a \mid a \in A, \text{a is an atom}\}$ and $A^- = \{a \mid not\ a \in A, \text{ a is an atom}\}$. Let B be a set of atoms. $not(B) = \{not\ b \mid b \in B, \text{b is atom}\}$. For a program P, atoms(P) denotes the set of all atoms x such that either x or *not* x appears in P. An atom set A *agrees with* B if $B^+ \subseteq A$ and $B^- \cap not(A) = \emptyset$. The *negation* of literal x is defined as $not(x)$ where if x is an atom, $not(x) = not\ x$, $not(not\ x) = x$. B *covers* A if $A \subseteq \{x \mid x \in B,\ or\ (not\ x) \in B\}$.

The algorithm for Smodels is given in Figure 3.1, where P is a ground program, and A is a set of literals. The function $smodels(P, A)$ returns *true* if there is a stable model of P that agrees with A. Given a set of literals, the algorithm first performs constraint propagation, which consists of two functions: $expand(P, A)$ and $lookahead(P, A)$. A superset of A, called A', is returned. It is shown that any stable model that agrees with A also agrees with A'. Then it checks whether a conflict happens in A'. If so, it returns false, causing backtracking. If A' is consistent and covers all the atoms in program P, then the atom set A'^+ is a stable model, and therefore the algorithm stops. If A' does not cover all the atoms in P, then the algorithm chooses a literal x outside of A' computed by the procedure $heuristic(P, A')$, and adds it to A'. If the chosen literal along with A' can be extended to a stable model, *true* is

```
Function smodels($P$, $A$)

$A$ := expand($P$, $A$)
$A$ := lookahead($P$, $A$)

If conflict($P$, $A$) then
   Return $false$
Else if $A$ covers Atoms($P$) then
  Return $true$ $\{A^+$ is stable model$\}$
Else
  $x$ := heuristic($P$, $A$)
  If smodels($P$, $A \cup \{x\}$) then
    Return $true$
  Else
    Return smodels($P$, $A \cup \{not\ x\}$)
  End if
End if
```

Figure 3.1: Algorithm of *smodels*

returned. Otherwise, the function tests whether $not\ x \cup A'$ can be extended to a stable model.

Function Expand(P, A)

Figure 3.2 shows the details of the function *expand*(P, A). The function *expand*(P, A) expands atom set A by using two functions: *atleast*(P, A) and *atmost*(P, A). Function *atleast*(P, A) returns a super set A' of A by applying four propagation rules. Any stable model that agrees with A must agree with

```
Function expand(P, A)
repeat
  $A' := A$
  $A$ := atleast$(P, A)$
  $A$ := $A \cup \{ not\ \chi \mid \chi \in Atoms(P)$
               $and\ \chi \notin atmost(P, A) \}$
until $A = A'$
return $A$.
```

Figure 3.2: Function expand(P, A)

atom set A'. The function *atmost*(P, A) computes another atom set A'', which contains all the possible consequences of A. Any literal x that is not in A'' must not be in any stable model that agrees with A, so *not* x must be in the stable model that agrees with A.

atleast(P, A) can be computed according to the properties of stable model semantics. Given a rule r in program P:

$$h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_m.$$

Define $min_r(A)$ to be the inevitable consequences of A with respect to rule r as

$$min_r(A) = \{h \mid \{a_1, ..., a_n\} \subseteq A^+,\ \{b_1, ..., b_m\} \subseteq A^-\}$$

Let S be the stable model that agrees with A. It means that if S agrees with the body of a rule, it also agrees with the head of the rule.

Define $max_r(A)$ to be the possible consequences of atom set A w.r.t rule r as

$$max_r(A) = \{h \mid \{a_1, ..., a_n\} \cap A^- = \emptyset,\ \{b_1, ..., b_m\} \cap A^+ = \emptyset\}$$

If there exists an atom a such that for all $r \in P, a \notin max_r(A)$, then S agrees with *not a*.

There are four propagation rules in *atleast*(P, A) [25]:

1. If $r \in P$, then $A := A \cup min_r(A)$.

2. If there is an atom a such that for all $r \in P$, $a \notin max_r(A)$, then $A := A \cup \{not\ a\}$.

3. If for an atom $a \in A$, there is only one $r \in P$ for which $a \in max_r(A)$, and there exists a literal x such that $a \notin max_r(A \cup \{x\})$, then $A := A \cup \{not\ x\}$.

4. If $not\ a \in A$, and there exists a literal x such that for some $r \in P$, $a \in min_r(A \cup \{x\})$, then $A := A \cup \{not\ x\}$.

Rule 1 adds the head of a rule to A if the body is true in A. Rule 2 says that if there is no rule with a as the head whose body is not false w.r.t. A, then a cannot be derived by any consistent extension of A, and thus cannot be in any stable model agreeing with A. Therefore, *not a* is added to A. Rule 3 says that if $a \in A$, the only rule r with a as the head must have its body true in A, so the body of r is added to A. Rule 4 forces the body of a rule to be false if the head is false in A.

The four rules can help derive new literals that the stable model agrees with. A fixpoint of operator $f_A(B)$ [25] can be defined to compute the *atleast*(P, A) which returns the smallest set of literals that can not be enlarged by using rules 1-4 above further.

$$\begin{aligned}
f_A(B) = A \cup B \\
&\cup \{a \mid a \in Atoms(P) \text{ and } a \in \min_r(B) \text{ and } r \in P\} \\
&\cup \{not\ \ a \mid a \in atom(P) \text{ and for all } r \in P, a \notin max_r(B)\} \\
&\cup \{not(x) \mid \text{there exists } a \in B \text{ such that } a \in max_r(B) \\
&\quad \text{for only one } r \in P \text{ and } a \notin max_r(B \cup x)\} \\
&\cup \{not(x) \mid \text{there exists } not\ \ a \in B \text{ and } r \in P \text{ such that} \\
&\quad a \in min_r(B \cup x)\}
\end{aligned}$$

Example 3.4.1 *Let P be the following program:*

$$\begin{array}{llll}
a & \leftarrow & c, not\ \ e. & \texttt{(r1)} \\
b & \leftarrow & not\ \ a. & \texttt{(r2)} \\
c & \leftarrow & a, not\ \ b. & \texttt{(r3)} \\
f & \leftarrow & b, not\ \ c. & \texttt{(r4)}
\end{array}$$

Given $A = \{b\}$, we compute *atleast*(P, A) as follows:

1. Because e does not appear in the head of any rule in P, by applying rule 2, we get $A = \{b,\ not\ e\}$;

2. Applying rule 3, since $b \in A$, and r_2 is the only rule with b as head, *not a* will be added to A. Now we get $A = \{b,\ not\ a,\ not\ e\}$;

3. Since $not\ a \in A$, and a is the head of r_1, $a \in min_{r_1}(A \cup c)$, so *not c* is added to A by applying rule 4. Now we get $A = \{b,\ not\ a,\ not\ e,\ not\ c\}$;

4. Applying rule 1, $f \in min_{r_4}(A)$, so f is added to A, then we get $A = \{b,\ not\ a,\ not\ e,\ not\ c,\ f\}$;

5. Since no new literals could be deduced, stop. ■

Function *atmost*(P, A) can be computed by the another fixpoint operator based on the following definition [25].

$$\begin{aligned}
f_r(S, C) = \{h \mid\ & h \leftarrow a_1, ..., a_n, not\ \ b_1, ..., not\ \ b_m \in P, \\
& a_i \in C \text{ for } 1 \leq i \leq n \text{ and } b_j \notin S \text{ for } 1 \leq j \leq m\}
\end{aligned}$$

where S and C are sets of atoms.

Define *atmost*(P, A) [25] as the least fixpoint of

$$f'(B) = \bigcup_{r \in P} f_r(A^+, B - A^-) - A^-$$

where B is a set of atoms.

$f'(B)$ contains h if a_i's are already in B but not in A^-, and none of the b_j's is in A^+. By including *not a* if $a \notin atmost(P, A)$, atmost generates *unfounded atoms* in the same way as in computing the well-founded model [8].

Example 3.4.2 *Let P be the program:*

$$\begin{array}{lll} a \leftarrow & not\ b. & (r_1) \\ c \leftarrow & a, not\ c. & (r_2) \\ b \leftarrow & c,\ not\ a. & (r_3) \end{array}$$

Given $A = \emptyset$, $atmost(P, \emptyset)$ $=\{a, c\}$ is computed as follows:
Starting with $B = \emptyset$:

$$\begin{array}{l}
f_{r_1}(\emptyset, B) = \{a\}, \\
f_{r_2}(\emptyset, B) = \emptyset, \\
f_{r_3}(\emptyset, B) = \emptyset, \\
B = f'(B) = f_{r_1}(\emptyset, B) \cup f_{r_2}(\emptyset, B) \cup f_{r_3}(\emptyset, B) - A^- = \{a\}, \\
f_{r_1}(\emptyset, B) = \{a\}, \\
f_{r_2}(\emptyset, B) = \{c\}, \\
f_{r_3}(\emptyset, B) = \emptyset, \\
B = f'(B) = f_{r_1}(\emptyset, B) \cup f_{r_2}(\emptyset, B) \cup f_{r_3}(\emptyset, B) - A^- = \{a, c\}, \\
f_{r_1}(\emptyset, B) = \{a\}, \\
f_{r_2}(\emptyset, B) = \{c\}, \\
f_{r_3}(\emptyset, B) = \{b\}, \\
B = f'(B) = f_{r_1}(\emptyset, B) \cup f_{r_2}(\emptyset, B) \cup f_{r_3}(\emptyset, B) - A^- = \{a, c, b\}, \\
f_{r_1}(\emptyset, B) = \{a\}, \\
f_{r_2}(\emptyset, B) = \{c\}, \\
f_{r_3}(\emptyset, B) = \{b\}, \\
B = f'(B) = f_{r_1}(\emptyset, B) \cup f_{r_2}(\emptyset, B) \cup f_{r_3}(\emptyset, B) - A^- = \{a, c, b\}.
\end{array}$$

Because the result of $f'(B)$ cannot be enlarged, so the iteration stops. Consequently *expand*(P, A) derives *not b* due to $b \notin atmost(P, A)$. ■

Lemma 3.4.3 [25] *The function expand(P, A) is monotonic in the parameter A.*

Because both $\bigcup_{r \in P} max_r(P, A)$ and $atmost(P, A)$ compute the inevitable consequences of A, a question arises. Are they the same? If not the same, what is the difference between them? In the following, we will answer this question.

Given a logic program, it is convenient to construct a *dependency graph*: for each rule $a \leftarrow b_1, ..., b_n, not\ c_1, ..., not\ c_m$ in the program, there is a positive edge from a to each b_i, $1 \leq i \leq n$, and a negative edge from a to each c_j, $1 \leq j \leq m$. A program has a *positive loop* if there is a path from an atom to itself which only contains positive edges.

Lemma 3.4.4 [28] *If a program P has no positive loops, then for any $a \notin atmost(P, A)$, $not\ a \in atleast(P, A)$.*

If there is no positive loop in the program, $atleast(P, A)$ can expand the literals without using $atmost(P, A)$.

Example 3.4.5 *Let P be the program:*

$$\begin{array}{lll} a \leftarrow c,\ not\ b. & & (r_1) \\ b \leftarrow not\ a. & & (r_2) \\ c \leftarrow c. & & (r_3) \end{array}$$

We can see that this program has only one stable model $\{b\}$. $atleast(P, \emptyset) = \emptyset$, and $atmost(P, \emptyset) = \{b\}$. As a result, *not a* and *not c* are derived by $expand(P, \emptyset)$. Due to the loop $c \leftarrow c$ in the program P, $atleast(P, \emptyset)$ cannot derive *not a*, neither *not c*. But they can be computed by the result of $atmost(P, \emptyset)$. ■

Function Lookahead(P, A)

The *expand* function uses some properties of stable semantics to refine a partially computed model, thus reducing the search space. After applying the *expand* function, we reach a point that no further literals can be deduced by *expand*. We then can further expand the partial model by the function *lookahead*, which tests every possible choice before committing to one. Consider a program P and a set of literals A. We choose an additional literal x outside of A, then set $A' = expand(P, A \cup \{x\})$. If there is a conflict in A', then *not* x is added to the partial model.

This is based on the following propagation rule:

If the stable model M agrees with the set of literals B but does not agree with $B \cup \{x\}$, it holds that M must agree with $B \cup \{not\ x\}$.

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

Figure 3.3: Function lookahead(P, A)

lookahead has been shown to be the most effective cost-saving method used in Smodels [23]. The function *lookahead*(P, A) is given in Figure 3.3.

Lookahead(P, A) calls function *lookahead_once*(P, A) repeatedly until no further literal can be added. *Lookahead_once*(P, A) picks up a literal from outside of A, say x, and expands A with x. Let $A' = expand(P, A \cup \{x\})$. If there is a conflict in A', then it returns $expand(P, A \cup \{not\ x\})$, otherwise, the function *lookahead_once*(P, A) tries all the literals outside of A, until a conflict happens or all the literals outside of A have been tried. In this case, it just returns A.

Lemma 3.4.6 [28] *The function lookahead*(P, A) *returns the same set, independent of any order in which literals are chosen from* B *in the while loop of the function lookahead_once*(P, A).

Example 3.4.7 *Consider the following program.*

$a \leftarrow not\ b.$
$b \leftarrow not\ a.$
$a \leftarrow b.$

The stable model for this program is $\{a\}$.

If lookahead checks *not a* first, a conflict occurs. Consequently, a is added to the resulting set. Suppose it then checks b, another conflicts happens. As a result, $\{a, not\ b\}$ is added to the result.

It can be shown that lookahead returns the same result regardless of the orders in which literals are checked. ■

Chapter 4

Arc-Consistency Vs. Lookahead

In this chapter, we first introduce a standard encoding from CSPs to answer set programs, then show that *arc-consistency* is weaker than *lookahead* and stronger than *expand* in Smodels under this translation.

4.1 Translation from CSPs to Logic Programs

There are different encodings of CSPs into SAT instances. To perform the comparison between *arc-consistency* for CSPs and *lookahead* in Smodels, we use a "standard translation" first given by Niemelä [21] from CSPs to logic programs. It is called "standard" because arguably it is the most direct and straightforward way to represent a CSP as a ground logic program. Other translations to function-free logic programs can be found in [16, 21].

Let $\mathcal{A}(X, D, C)$ be a CSP. We denote by $P_{\mathcal{A}}$ the logic program translated from it. The translation of Niemelä [21] consists of three parts. The first part specifies the uniqueness property - a variable in CSP can only be assigned with one value. For each variable $x_i \in X$ and its domain $D_{x_i} = \{a_1, ..., a_l\}$, we use an atom, $x_i(a_j)$, to represent whether or not x_i gets the value a_j. Thus, for each $1 \leq j \leq l$, we add a rule

$$x_i(a_j) \leftarrow not\ x_i(a_1), ..., not\ x_i(a_j), not\ x_i(a_{j+1}), ..., not\ x_i(a_l). \tag{4.1}$$

In the second part, constraints are expressed. For each constraint $c_i(x_1, ..., x_n)$ where $c_i \in C$, and each tuple $(a_1, ..., a_n) \in c_i$, we add

$$sat(c_i) \leftarrow x_1(a_1), ..., x_n(a_n). \tag{4.2}$$

Finally, we express that every constraint in C must be satisfied by

$$sat \leftarrow sat(c_1), ..., sat(c_m). \tag{4.3}$$
$$f \leftarrow not\ f, not\ sat. \tag{4.4}$$

where f is a new symbol.

The rules in 4.3 and 4.4 can be omitted if we ask Smodels to compute the stable models containing $sat(c_1), ..., sat(c_m)$. This is what we assume in this thesis. We denote this set by $sat(C) = \{sat(c_1), ..., sat(c_m)\}$.

Under this assumption, it can be shown that given a CSP $\mathcal{A}(X, D, C)$ and a set of literals M such that sat(c_i) $\in M$ for any $c_i \in C$, M is a stable model of $P_\mathcal{A}$ iff $\{x \to a \mid x \in X, a \in D_x, x(a) \in M\}$ is a solution to the given CSP.

Proposition 4.1.1 *A* CSP $\mathcal{A}$ *has a solution iff* $P_\mathcal{A}$ *has a stable model.*

Proof Suppose $\mathcal{A}(X, D, C)$ has a solution with $\{x_1 = v_1, \ldots, x_n = v_n\}$ where n is the number of variables in $\mathcal{A}(X, D, C)$. Then for every assignment $x_i = v_i$ in the solution, we define a set $\mathcal{B}$ that contains $x_i(v_i)$. Obviously, $\mathcal{B}$ is a stable model of $P_\mathcal{A}$, because it satisfies all translated rules in $P_\mathcal{A}$, and cannot be reduced to a smaller set.

According to the translation, every constraint must be satisfied, and every variable has only one assignment. If $P_\mathcal{A}$ has a stable model, it must include $\{sat(c_1), \ldots, sat(c_m), x_1(v_1), \ldots, x_n(v_n)\}$, Hence the assignment $x_1 \to , \ldots, x_n \to v_i$ for the variables satisfies all the constraints, and is just a solution for $\mathcal{A}(X, D, C)$. ■

The size of $P_\mathcal{A}$ is calculated as follows. For each variable the uniqueness property is expressed by at most k rules, and each rule with at most k literals, where k bounds the domain size. The number of tuples in a constraint is at most k^2. We have e constraints, therefore

Proposition 4.1.2 [28] *For any binary* CSP $\mathcal{A}(X, D, C)$, $P_\mathcal{A}$ *can be constructed in* $O(ek^2)$ *time. The size of* $P_\mathcal{A}$ *is also bounded by* $O(ek^2)$. *The number of literals is bounded by* $O(ek)$. ■

Example 4.1.3 *Consider the* CSP $\mathcal{A}(X, D, C)$ *in Example 2.2.2 again:* $C = x < y \wedge y < z \wedge z \leq 3$, *with* $D_x = D_y = \{1, 2, 3\}$ *and* $D_z = \{2, 3, 4\}$. *Its translation* $P_\mathcal{A}$ *consists of the following rules:*

$x(1) \leftarrow not\ x(2), not\ x(3).$
$x(2) \leftarrow not\ x(1), not\ x(3).$
$x(3) \leftarrow not\ x(1), not\ x(2).$
$y(1) \leftarrow not\ y(2), not\ y(3).$
$y(2) \leftarrow not\ y(1), not\ y(3).$
$y(3) \leftarrow not\ y(1), not\ y(2).$
$z(2) \leftarrow not\ z(3), not\ z(4).$
$z(3) \leftarrow not\ z(2), not\ z(4).$
$z(4) \leftarrow not\ z(2), not\ z(3).$
$sat(c_1) \leftarrow x(1), y(2).$

$sat(c_1) \leftarrow x(1), y(3).$
$sat(c_1) \leftarrow x(2), y(3).$
$sat(c_2) \leftarrow y(1), z(2).$
$sat(c_2) \leftarrow y(1), z(3).$
$sat(c_2) \leftarrow y(1), z(4).$
$sat(c_2) \leftarrow y(2), z(3).$
$sat(c_2) \leftarrow y(2), z(4).$
$sat(c_2) \leftarrow y(3), z(4).$
$sat(c_3) \leftarrow z(2).$
$sat(c_3) \leftarrow z(3).$
$sat \leftarrow sat(c_1), sat(c_2), sat(c_3).$
$f \leftarrow not\ f, sat.$

One can verify that $lookahead(P_{\mathcal{A}}, sat(C))$ returns the following set of literals
$\{not\ x(2), not\ x(3), not\ y(1), not\ y(3), not\ z(2), not\ z(4), x(1), y(2), z(3), sat(c)\}$.
The reader may want to examine this construction against domain reduction by arc-consistency as given in Example 2.2.2. We can see that 2 and 3 are removed from D_x, 1 and 3 from D_y, 2 and 4 from d_z. ■

4.2 AC^+ Vs. Expand and Lookahead

Under the above translation, one question arises in the equivalence between *arc-consistency* and *lookahead*: Is the propagation by the *expand* function powerful enough to enforce arc-consistency? The answer is NO, as shown by the following example.

Example 4.2.1 *Consider a* CSP $\mathcal{A}(X, D, C)$, $X = x, y$, *with one constraint* $c_{x,y} = \{(0,0),\ (1,1)\}$, *where* $D_x = \{0,1,2\}$ *and* $D_y = \{0,1\}$. *Under Niemelä's translation, we have the following rules that ensure every variable can only be assigned one value:*
$x(0) \leftarrow not\ x(1), not\ x(2).$
$x(1) \leftarrow not\ x(0), not\ x(2).$
$x(2) \leftarrow not\ x(0), not\ x(1).$
$y(0) \leftarrow not\ y(1).$
$y(1) \leftarrow not\ y(0).$
and two rules with $sat(c)$ *as the head:*

$sat(c) \leftarrow x(0), y(0).$
$sat(c) \leftarrow x(1), y(1).$

Enforcing arc-consistency removes 2 from D_x. *not* $x(2)$ cannot be derived by *expand*$(P_{\mathcal{A}}, sat(c))$ because no propagation rule can be applied in *expand*$(P_{\mathcal{A}}, sat(c))$. However, if we perform lookahead, by picking up $x(2)$ in *lookahead_once*$(P_{\mathcal{A}}, sat(C))$, *expand*$(P_{\mathcal{A}}, sat(C) \cup x(2))$ derives *not* $sat(c)$, resulting in a conflict with $sat(c)$. ■

The following proposition defines a condition under which *expand* coincides with *arc-consistency.*

Proposition 4.2.2 [23] *If every constraint c_i in a* CSP $\mathcal{A}(X, D, C)$ *only has one tuple, then expand*$(P_{\mathcal{A}}, \{sat(c_1), \ldots, sat(c_n)\}$ *enforces arc-consistency on the* CSP $\mathcal{A}(X, D, C)$.

Proof It can be proved by trivially applying the propagation rule 3 (cf. Section 3.4). ■

In the rest of this thesis, as we are going to enforce the propagation rule in *atleast*, we revise the propagation rule 3 as follows:

If for all $r \in P$ with $a \in max_r(A)$, there exists a literal x such that $a \notin \bigcup_r max_r(A \cup \{x\})$, then $A := A \cup \{not\ x\}$.

Example 4.2.3 *Let P be the following program:*

$c_1 \leftarrow x(0), y(0), z(0).$
$c_1 \leftarrow x(0), y(0), z(1).$

$x(0)$ can be derived by *expand*$(P, \{c_1\})$ because $c_1 \notin max_r(\{c_1 \cup \{not\ x(0)\}\})$ for all $r \in P$ with $c_1 \in max_r(\{c_1\})$. Similarly, $y(0)$ can also be derived because $c_1 \notin max_r(\{c_1 \cup \{not\ y(0)\}\})$ for all $r \in P$ with $c_1 \in max_r(\{c_1\})$. ■

Proposition 4.2.4 [28] *Let* $\mathcal{A}(X, D, C)$ *be a* CSP. *Suppose after an application of arc-consistency, D is reduced to D'. If for any variable $x_j \in X$, a value d is removed from D_{x_j}, then not $x_j(d)$ is in the set returned by lookahead*$(P_{\mathcal{A}}, sat(C))$. ■

The following example shows that lookahead is strictly stronger than arc-consistency.

Example 4.2.5 *Consider a* CSP *that consists of three variables x, y and y, all with the domain $\{0, 1\}$, and the following constraints:*

$$c_1(x, y) = \{(0, 0), (1, 1)\}$$
$$c_2(y, z) = \{(0, 1), (1, 0)\}$$
$$c_3(z, x) = \{(0, 0), (1, 1)\}$$

It is clear that arc-consistency cannot reduce any domain. A backtracking algorithm that begins with the assignment $x \rightarrow 0$ would need to backtrack. However, when $x(0)$ is selected by lookahead, a contradiction is derived so that *not* $x(0)$ is added, and subsequently *not* $x(1)$, *not* $y(0)$, *not* $y(1)$, *not* $z(0)$ and *not* $z(1)$. ■

Chapter 5

Arc-Consistency with Unit Propagation

In the previous chapter, we knew that arc-consistency is strictly weaker than lookahead. In this chapter, we identify what is missing in *arc-consistency*, and devise a new method that prunes the same search space as lookahead. Then we present an algorithm for this method, and prove its correctness.

5.1 Definition

The insight revealed in the last chapter suggests that some space pruning power is missing in arc-consistency. In this section we identify what is missed.

Given a CSP $\mathcal{A}(X, D, C)$ and a partial instantiation Π, we define a function that extends Π as follows [28]:

$$\begin{aligned}&\mathit{unit\_propagate}(\mathcal{A}, \Pi)\\ &= \Pi \cup \{x \to a \mid c_{y,x} \in C \text{ or } c_{x,y} \in C \text{ and } y \to b \in \Pi \text{ such that}\\ &\qquad a \text{ is the only value in } D_x \text{ that is consistent with } b\}\end{aligned}$$

That is, in a binary constraint, if an unassigned variable has exactly one value in its domain that is consistent with the value of the assigned variable, the unassigned variable must be assigned this value, given Π.

A collection of pairs Π is said to be in *conflict* if and only if there are distinct values d and d' such that $x \to d, x \to d' \in \Pi$. The notion of conflict is needed because, as we will see shortly, the function unit_propagate may lead to such a conflict.

The function $\mathit{unit\_propagate}^*(\mathcal{A}, \Pi)$ below calls $\mathit{unit\_propagate}(\mathcal{A}, \Pi)$ repeatedly until nothing can be further propagated or a conflict is reached [28].

That is,

```
Function unit_propagate*($\mathcal{A}, \Pi$)
    repeat
    $\Pi' := \Pi$
    $\Pi :=$ unit_propagate($\mathcal{A}, \Pi'$)
    if $\Pi$ is in conflict then
            return "conflict"
    until $\Pi = \Pi'$
return $\Pi$.
```

Example 5.1.1 *Let $\mathcal{A}(X, D, C)$ be a* CSP *with $X = \{x, y\}$, $D_x = D_y = \{0, 1\}$, and C consisting of*

$$c_1(x, y) = \{(0, 1)\} \quad c_2(y, z) = \{(1, 1)\} \quad c_3(z, x) = \{(1, 1)\}$$

Given $\Pi = \{x \to 0\}$, unit_propagate$(\mathcal{A}, \Pi)$ returns "conflict" because the set returned by unit_propagate$(\mathcal{A}, \Pi)$ is $\{x \to 0, y \to 1, z \to 1, x \to 1\}$.* ■

Now we strengthen the process of enforcing arc-consistency by adding unique value propagation, called unit propagation. We name the function AC^+, and describe it as an abstract, non-deterministic procedure.

Procedure AC^+ [28]

AC^+ takes as input a CSP $\mathcal{A}(X, D, C)$, and performs the following domain reduction operations repeatedly until no domain can be further reduced.

1. For any $c \in C$, if there is exactly one uninstantiated variable x in its scope, remove any d from D_x if d is inconsistent with the value of the instantiated variable in c.

2. For any $c \in C$ where both variables x and y in the scope of c are uninstantiated, remove any d from D_x if

   (a) there is no value in D_y that, together with d, is consistent with c; or

   (b) *unit_propagate**$(\mathcal{A}, \{x \to d\})$ returns "conflict".

AC^+ is obviously correct. The only addition is part (b), in which case the conflict is a sufficient ground for removing d. Furthermore, AC^+ doesn't seem to take a lot more time than AC. Note that when the domains of all affected variables have more than one valid value, the process of unit propagation stops. So the overhead is proportional to the occurrences of unique values for variables during unit propagation.

Theorem 5.1.2 [28] *Let $\mathcal{A}(X, D, C)$ be a* CSP. *Suppose after an application of AC^+, D is reduced to D'. Then, for any variable $x \in X$, a value d is removed from D_x iff not $x(d)$ is in the set returned by lookahead$(P_{\mathcal{A}}, sat(C))$.* ■

By Theorem 5.1.2 and Lemma 3.4.6, we therefore have:

Corollary 5.1.3 *Let $\mathcal{A}(X, D, C)$ be a* CSP. *AC^+ returns a same set of removal values from D, independent of any order in which variables are chosen from X, and values are chosen from D.* ■

Note that both node consistency and arc-consistency are implemented in AC^+. In the following section, as we are going to give an algorithm for arc-consistency with unit propagation, we delete the first part of AC^+ which enforces node consistency. We call it ACUP.

<u>**Procedure ACUP**</u> [28]
ACUP takes as input a CSP $\mathcal{A}(X, D, C)$, and performs the following domain reduction operations repeatedly until no domain can be further reduced.

1. For any $c \in C$ where both variables x and y in the scope of c are uninstantiated, remove any d from D_x if
   (a) there is no value in D_y that, together with d, is consistent with c; or
   (b) *unit_propagate*$^*(\mathcal{A}, \{x \rightarrow d\})$ returns "conflict".

Corollary 5.1.4 *Let $\mathcal{A}(X, D, C)$ be a* CSP. ACUP *returns a same set of removed values from D, independent of any order in which variables are chosen from X, and values are chosen from D.* ■

5.2 An Algorithm for ACUP

We design an algorithm for ACUP based on AC-4. We call this algorithm AC4-UP. AC4-UP uses the same data structure as AC-4. An array *counter*(x_i, a_i, x_j) is used to store the number of support values from neighboring variables for the assignment of $\langle x_i, a_i\rangle$, and an array $S\langle x_j, a_j\rangle$ to store values in the other variables supported by $\langle x_j, a_j\rangle$. Additionally, $S_1(x_j, a_j, x_i)$ is used to maintain consistent values for x_i that are supported by $\langle x_j, a_j\rangle$.

AC4-UP is given in Figure 5.1. It works as follows. First, it initializes the values for S, S_1 and *counter* for all the constraints. In the remaining parts of the algorithm, it adds unsupported values to $\mathcal{L}$ ($\mathcal{L}$ stores the conflict pairs $\langle x_i, a_i\rangle$ due to *counter*$(x_i, a_i, x_j) = 0$, or *unit_propagate*$^*(\mathcal{A}, \{x_i \rightarrow a_i\})$ returning true). In each step of the while loop in part 5, the algorithm picks up an unsupported value from $\mathcal{L}$, updates all the affected counters and affected sets of consistent values, puts any unsupported value to $\mathcal{L}$ by checking the counter and checking the conflict using unit propagation. This procedure continues until all the unsupported values are removed.

Consider applying this algorithm to Example 4.2.5.
Initially, we have

$$
\begin{aligned}
&S(x,0) = \{\langle y,0\rangle, \langle z,0\rangle\}\\
&S(x,1) = \{\langle y,1\rangle, \langle z,1\rangle\}\\
&S(y,0) = \{\langle x,0\rangle, \langle z,1\rangle\}\\
&S(y,1) = \{\langle x,1\rangle, \langle z,0\rangle\}\\
&S(z,0) = \{\langle y,1\rangle, \langle x,0\rangle\}\\
&S(z,1) = \{\langle y,0\rangle, \langle x,1\rangle\}\\
&S_1(x,0,y) = \{\langle y,0\rangle\}\\
&S_1(x,0,z) = \{\langle z,0\rangle\}\\
&S_1(x,1,y) = \{\langle y,1\rangle\}\\
&S_1(x,1,z) = \{\langle z,1\rangle\}\\
&S_1(y,0,x) = \{\langle x,0\rangle\}\\
&S_1(y,0,z) = \{\langle z,1\rangle\}\\
&S_1(y,1,x) = \{\langle x,1\rangle\}\\
&S_1(y,1,z) = \{\langle z,0\rangle\}\\
&S_1(z,0,x) = \{\langle x,0\rangle\}\\
&S_1(z,0,y) = \{\langle y,1\rangle\}\\
&S_1(z,1,x) = \{\langle x,1\rangle\}\\
&S_1(z,1,y) = \{\langle y,0\rangle\}\\
&\mathit{counter}(x,0,y) = 1\\
&\mathit{counter}(x,0,z) = 1\\
&\mathit{counter}(x,1,y) = 1\\
&\mathit{counter}(x,1,z) = 1\\
&\mathit{counter}(y,0,x) = 1\\
&\mathit{counter}(y,0,z) = 1\\
&\mathit{counter}(y,1,x) = 1\\
&\mathit{counter}(y,1,z) = 1\\
&\mathit{counter}(z,0,x) = 1\\
&\mathit{counter}(z,0,y) = 1\\
&\mathit{counter}(z,1,x) = 1\\
&\mathit{counter}(z,1,y) = 1
\end{aligned}
$$

After initiation, $\mathcal{L}$ is empty, and
unit_propagate*(A, $\{x \rightarrow 0\}$) is called and the following conflict chain is generated.

$$\{\langle x,0\rangle, \langle y,0\rangle, \langle z,1\rangle, \langle x,1\rangle\}.$$

```
AC4-UP
Input: a CSP $\mathcal{A}(X, D, C)$
Output: A CSP equivalent to $\mathcal{A}(X, D, C)$
1 Initialize $S(x_i, a_i), counter(x_i, a_i, x_j), S_1(x_i, a_i, x_j)$ from all $C_{ij}$;
2 for all counters do
      if $counter(x_i, a_i, x_j)$=0 (if $\langle x_i, a_i\rangle$ is unsupported by $x_j$ ) then
          add $\langle x_i, a_i\rangle$ to $\mathcal{L}$;
      end
  end
3 for all counters do
      if $counter(x_i, a_i, x_j)$=1 (if there is unit propagation ) then
          add $\langle x_i, a_i\rangle$ to $\mathcal{L}_1$;
      end
  end
4 for each $\langle x_j, a_j\rangle \in \mathcal{L}_1$ do
      if $unit\_propagate^*(\mathcal{A}, \{x_j \to a_j\})$ then
          add $\langle x_j, a_j\rangle$ to $\mathcal{L}$;
      end
  end
5 while $\mathcal{L}$ is not empty do
      Choose $\langle x_i, a_i\rangle$ from $\mathcal{L}$, remove it from $\mathcal{L}$, and remove $a_i$ from $D_{x_i}$;
6     for each $\langle x_j, a_j\rangle \in S(x_i, a_i)$ do
          decrement $counter(x_j, a_j, x_i)$;
          if $counter(x_j, a_j, x_i) = 0$ then
              add $\langle x_j, a_j\rangle$ to $\mathcal{L}$;
          end
          delete $\langle x_i, a_i\rangle$ from $S_1(x_j, a_j, x_i)$;
7         for all assignments do
              if $counter(x_k, a_k, x_i) = 1$ then
                  if $unit\_propagate^*(\mathcal{A}, \{x_k \to a_k\})$ then
                      add $\langle x_k, a_k\rangle$ to $\mathcal{L}$ ;
                  end
              end
          end
      end
  end
```

Figure 5.1: An algorithm for ACUP based on AC-4

```
unit_propagate*
Input: a CSP A, an assignment $x_i \to a_i$
Output: true: if there is a conflict; false: if there is no conflict.
initialize queue, $R$ to empty;
add $\langle x_i, a_i \rangle$ to queue;
1 while queue is not empty do
      remove first element $\langle x_j, a_j \rangle$ from queue;
      add $\langle x_j, a_j \rangle$ to $R$      /*$R$ stores the set of assignments propagated
                                      from $x_i \to a_i$ */;
      if there is a conflict in $R$ then
          return true       /*conflict */;
      else
2             for every $x_i$ such that $counter(x_j, a_j, x_i) = 1$ do
                  fetch $\langle x_i, a_i \rangle \in S_1(x_j, a_j, x_i)$;
                  add $\langle x_i, a_i \rangle$ to queue;
              end
      end
  end
  return false;
```

Figure 5.2: *unit_propagate** algorithm

A conflict occurs since x is assigned 0 and 1 simultaneously. So $\langle x, 0\rangle$ is added to $\mathcal{L}$. Similarly, all tuples in

$$\{\langle y, 0\rangle, \langle z, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle, \langle z, 0\rangle\}$$

are also added to $\mathcal{L}$.
Now arc-consistency is applied, and the domain is annihilated.

5.3 The Correctness of AC4-UP

Before giving a proof of the correctness of AC4-UP, we need to introduce the following definition and lemmas.

Definition 5.3.1 *A unit propagation chain in* CSPs *is a sequence of assignments* $\{x_1 \rightarrow a_1, \ldots, x_n \rightarrow a_n\}$. *If there is a conflict in it, we call it a conflict chain.*

Example 5.3.2 *Consider the Example 4.2.5. There is a conflict propagation chain* $\{x \rightarrow 0, y \rightarrow 0, z \rightarrow 1, x \rightarrow 1\}$ *in it.* ■

Lemma 5.3.3 *Given a* CSP $\mathcal{A}(X, D, C)$, *a conflict chain* $\{x_1 \rightarrow a_1, \ldots, x_n \rightarrow a_n\}$ *in* $\mathcal{A}$, *if the domain value* a_i, *where* $1 \leq i \leq n$, *is removed from its domain, then* a_i *can be removed by arc-consistency.*

Proof Suppose a_i is removed for the domain D_{x_i} where $1 \leq i \leq n$. Because a_i is the only domain value for x_i that supports $x_{i-1} \rightarrow a_{i-1}$ in a certain constraint, whose scope contains x_{i-1} and x_i. If a_i is removed, there is no domain value that supports $x_{i-1} \rightarrow a_{i-1}$. So a_{i-1} is removed from $D_{x_{i-1}}$. Continuously, $a_{i-2}, \ldots, a_2$, and a_1 are removed from $D_{x_{i-2}}, \ldots, D_{x_2}$, and D_{x_1} respectively. ■

Lemma 5.3.3 specifies that when the conflict chain from a certain assignment is broken, the inconsistency due to this assignment can still be found by arc-consistency.

Lemma 5.3.4 *Let* $\mathcal{A}(X, D, C)$ *be a* CSP. AC4-UP *returns the same set of the removed values from* D, *independent of any order in which a pair is chosen from* $\mathcal{L}$ *in the while loop of part 5 in* AC4-UP.

Lemma 5.3.4 is used in the proof of the correctness for AC4-UP.

Proof Let E and E' be the sets of removed values of any two invocations A_1 and A_2 of AC4-UP. Let the corresponding sequences of such removals be $E = \{d_1, ..., d_n\}$ and $E' = \{d'_1, ..., d'_{n'}\}$. We will prove that for any $d_i \in E$, we have $d_i \in E'$. We show, by a simple induction on n, that each $d_i \in E'$, where $1 \leq i \leq n$.

Obviously, if d_1 is removed by A_1, $\langle x_1, d_1 \rangle$ is added to $\mathcal{L}$ before the while loop in part 5, then there must be *counter*(x_1, d_1, x_k)= 0 in part 3, or *unit_propagate*$^*(\mathcal{A}, \{x_1 \rightarrow d_1\})$ returning true in part 4 initially. So d_1 is removed by A_2.

Now suppose any d_k, where $1 \leq k \leq i-1$, is removed by A_1, and d_k is also removed by A_2. We want to show that if d_i is removed by A_1, d_i is also removed by A_2.

If $\langle x_i, d_i \rangle$ is added to $\mathcal{L}$ before the while loop in part 5 in A_1, d_i can be removed by A_2 in a similar way as above.

If $\langle x_i, d_i \rangle$ is added to $\mathcal{L}$ after the while loop in part 5 in A_1, then there are two cases:

$\langle x_i, d_i \rangle$ is added to $\mathcal{L}$ by *counter*$(x_i, d_i, x_k) = 0$ or *unit_propagate*$^*(\mathcal{A}, \{x_i \rightarrow d_i\}) =$ true. We show that in either case A_2 also achieves the same result, and removes d_i from D_{x_i}.

1. If *counter*$(x_i, d_i, x_j) = 0$ in A_1, it is caused by the removal of $d_1, d_2, \ldots, d_{i-1}$. Therefore, for some x_j, where $x_j \in \{x_1, x_2 \ldots, x_{i-1}\}$, $S(x_j, d_j) = \{\langle x_i, d_i \rangle, \ldots\}$, and *counter*$(x_i, d_i, x_j)$ is decreased by one for x_j. Due to the fact that for any d_k, where $1 \leq k \leq i-1$, $d_k \in \{d'_1, d'_2 \ldots, d'_{i'-1}\}$ is also removed by A_2, A_2 always has a chance to decrease *counter*(x_i, d_i, x_j) for every x_j. So *counter*$(x_i, d_i, x_k) = 0$ can always be reached for A_2 regardless of the order in which d'_k are removed. As a result, d_i is removed by A_2.

2. If *unit_propagate*$^*(\mathcal{A}, \{x_i \rightarrow d_i)\} =$ true in A_1, there is a conflict propagation chain. In A_2, because there is a loop in part 7, *uni_propagate*$^*(\mathcal{A}, \{x_i \rightarrow d_i)\}$ can always be called. Also, for any d_k, where $1 \leq k \leq i-1$, $d_k \in \{d'_1, d'_2 \ldots, d'_{i'-1}\}$ is also removed by A_2. Thus, A_2 can decrease *counter*(x_i, d_i, x_j) for every x_j. As a result, the counters in the propagation chain can be set to 1. If any counter in the propagation chain becomes zero, by Proposition 5.3.3, $\langle x_i, d_i \rangle$ is added to $\mathcal{L}$, hence d_i is removed by A_2. Otherwise, *unit_propagate*$^*(\mathcal{A}, \{x_i \rightarrow d_i\})$ returns true in A_2. Consequently, d_i is removed by A_2.

We therefore conclude that $E \subseteq E'$. By symmetry, $E' \subseteq E$. Therefore, $E = E'$. ■

Now we give the theorem stating the correctness of AC4-UP.

Theorem 5.3.5 *Let $\mathcal{A}(X, D, C)$ be a* CSP. *Suppose after an application of* ACUP, *D is reduced to D'. Then for any variable $x \in X$, a value d is removed from D_x by* ACUP *iff d is also removed from D_x by* AC4-UP.

Note that by Lemma 5.3.4 and Lemma 5.1.4, we know that both ACUP and AC4-UP return a same set of removed values from their respective domains, independent of any order of removed values. Our proof is based on the idea that for any sequence of removed values in one side, there is a corresponding sequence in the other.

Proof ($\Rightarrow$) Let $d_1, ..., d_n$ be the sequence of domain values removed from the respective domains $D_{x_1}, ..., D_{x_n}$ by ACUP, in that order. We show that there is a corresponding sequence $d_1, ..., d_n$ of domain values removed by AC4-UP.

We show, by a simple induction on n, that each d_i, where $1 \leq i \leq n$, is removed by AC4-UP.

First we prove that if d_1 is removed by ACUP, d_1 is also removed by AC4-UP. According to the function ACUP, d_1 is removed by *arc-consistency* or *unit propagation*. The proof is based on above two cases.

1. Suppose d_1 is removed from D_{x_1} by ACUP due to the fact that there is no value b in y such that $\langle d_1, b\rangle \in c_{x_1,y}$.

   In AC4-UP, according to the definition of *counter*, after the initialization in part 1, *counter*$(x_1, d_1, y) = 0$. So $\langle x_1, d_1\rangle$ will be added to $\mathcal{L}$. In part 5, the while loop will pick up $\langle x_1, d_1\rangle$ in $\mathcal{L}$ (note that this assumption is valid because the order of picking up pair in $\mathcal{L}$ is unimportant (cf. Lemma 5.3.4)) and d_1 is removed from D_{x_1}.

2. Suppose d_1 is removed from D_{x_1} in ACUP, because *unit_propagate** $(\mathcal{A}, \{x_1 \rightarrow d_1\})$ returns "conflict". We need to show that d_1 can also be removed by AC4-UP.

   Suppose "conflict" by *unit_propagate**$(\mathcal{A}, \{x_1 \rightarrow d_1\})$ is due to a conflict chain as follows:

   $$\{x_1 \rightarrow d_1, x_{i'_1} \rightarrow d_{i'_1}, ..., x_{i'_n} \rightarrow d_{i'_n}\} \tag{5.1}$$

   where $x_1 = x_{i'_n}$ and $d_1 \neq d_{i'_n}$.

   In AC4-UP, after the initialization in part 1, *counter*$(x_1, d_1, x_{i'_2}) = 1$, *counter*$(x_{i'_2}, d_{i'_2}, x_{i'_3}) = 1$, ..., *counter*$(x_{i'_{n-1}}, d_{i'_{n-1}}, x_{i'_n}) = 1$. In part 3, $\langle x_1, d_1\rangle, \ldots, \langle x_{i'_n}, d_{i'_n}\rangle$ are added to $\mathcal{L}_1$. In part 4, *unit_propagate**$(\mathcal{A}, \{x_1 \rightarrow a_1\})$ causes a conflict, so $\langle x_1, d_1\rangle$ is added to $\mathcal{L}$ in part 4 of AC4-UP. In part 5, the while loop picks up $\langle x_1, d_1\rangle$ in $\mathcal{L}$ and d_1 is removed from D_{x_1}.

Now we will show that if any d_i, $i > 0$, is removed by AC4-UP, d_{i+1} is also removed by AC4-UP.

There are two cases in which d_{i+1} is removed by ACUP.

1. d_{i+1} is removed by ACUP due to the fact that there is no value b in y for some k, where $1 \leq k \leq i$, such that $\langle d_{i+1}, b\rangle \in c_{x_i,y}$.

   So there is $counter(x_{i+1}, d_{i+1}, y) = 0$. If this is caused by removal of $d_1, d_2, \ldots, d_i$ in ACUP, there is also a corresponding sequence in AC4-UP, so $\langle x_{i+1}, d_{i+1}\rangle$ is removed. As a result d_{i+1} is removed in AC4-UP by part 5.

2. Suppose d_{i+1} is removed from $D_{x_{i+1}}$ in ACUP, because *unit_propagate** $(\mathcal{A}, \{x_{i+1} \rightarrow d_{i+1}\})$ returns "conflict". We need to show that d_{i+1} can also be removed by AC4-UP.

   If this is caused by a sequence of removal $d_1, d_2, \ldots, d_i$ in ACUP, there exists the same sequence in AC4-UP to cause this result (the assumption holds according to the Lemma 5.1.4). In AC4-UP, *unit_propagate**$(\mathcal{A}, \{x_{i+1} \rightarrow d_{i+1}\})$ can always be called, and return true. So $\langle x_{i+1}, d_{i+1}\rangle$ is added to $\mathcal{L}$, and d_{i+1} is removed by AC4-UP.

($\Leftarrow$) Let $d_1, ..., d_n$ be the sequence of domain values removed from the respective domains $D_{x_1}, ..., D_{x_n}$ by AC4-UP, in that order. We show that there is a corresponding sequence $d_1, ..., d_n$ of domain values removed by ACUP. We show, by a simple induction on n, that each d_i where $1 \leq i \leq n$ is also removed by ACUP.

Obviously d_1 is removed due to the fact that $counter(x_1, d_1, y) = 0$, or *unit_propagate**$(\mathcal{A}, \{x_1 \rightarrow d_1\}) =$ true outside the loop in part 5 of AC4-UP.

If $counter(x_1, d_1, y) = 0$, it means that for a certain constraint $c_{x_1,y}$, there is no value in D_y that, together with d_1 in D_{x_1}, is consistent with $c_{x_1,y}$, so d_1 is removed by ACUP (Note that we can choose this x_1 first, and do a comparison based on the Lemma 5.1.4).

If *unit_propagate**$(\mathcal{A}, \{x_1 \rightarrow d_1\}) =$ true, *unit_propagate**$(\mathcal{A}, \{x_1 \rightarrow d_1\})$ returns "conflict". As a result, d_1 is removed also by ACUP.

Now suppose any d_k, where $1 \leq k \leq i$, is removed by ACUP, we show that d_{i+1} is also removed by ACUP.

If $\langle x_{i+1}, d_{i+1}\rangle$ is added to $\mathcal{L}$ by the outside loop in part 5 of AC4-UP. d_{i+1} can be removed by the same way as d_1.

If $\langle x_{i+1}, d_{i+1}\rangle$ is added to $\mathcal{L}$ by the inside loop in part 5 of AC4-UP, then $counter(x_{i+1}, d_{i+1}, x_k) = 0$ or *unit_propagate**$(\mathcal{A}, \{x_{i+1} \rightarrow d_{i+1}\}) =$ true.

1. $counter(x_{i+1}, d_{i+1}, y) = 0$. It must be caused by AC4-UP in a sequence $\{d_1', d_2', \ldots, d_i'\} \in \{d_1, d_2, \ldots, d_i\}$. In ACUP, we can apply the same sequence $\{d_1', d_2', \ldots, d_i'\}$ (note that this can be achieved by Lemma 5.1.4).

Finally, there exists a certain constraint $c_{x_{1+1},y}$, such that there is no value in D_y that, together with d_{i+1} in $D_{x_{x+1}}$, is consistent with $c_{x_{i+1},y}$. Consequently, d_{i+1} is removed.

2. *unit_propagate*$^*(\mathcal{A}, \{x_{i+1} \rightarrow d_{i+1}\}) =$ true. Obviously, this is caused by a sequence $d'_1, d'_2, \ldots, d'_i$ of removed values from domains in AC4-UP generating a conflict propagation chain, where for any d'_k, $d'_k \in \{d_1, d_2, \ldots, d_i\}$. In ACUP, we can apply the same sequence $\{d'_1, d'_2, \ldots, d'_i\}$ by Lemma 5.1.4.

   unit_propagate$^*(\mathcal{A}, \{x_{i+1} \rightarrow d_{i+1}\})$ returns "conflict" for this propagation chain. So d_{i+1} is removed. ■

5.4 Complexity

Suppose e is the number of constraints, and k is the maximum domain size. We have the following theorem:

Theorem 5.4.1 *The time complexity of* AC4-UP *is* $O(e^3k^3)$*, and space complexity is* $O(ek^2)$.

Proof In *unit_propagate**, there are at most $O(e)$ assignments added to *queue* before a conflict happens, so the total running time for *unit_propagate** is bounded by $O(e)$.

There are a total of $O(ek^2)$ elements in S, so *decrement counter* is performed at most $O(ek^2)$ times. In each iteration of decreasing the counters, *unit_propagate** is performed $O(ek)$ times, and each call of *unit_propagate** takes $O(e)$ time. Consequently the total running time is $O(e^3k^3)$.

Since AC4-UP doesn't use any array that exceeds the size of S in AC-4, the space complexity of AC4-UP is the same as AC-4, which is $O(ek^2)$. ■

We have presented an algorithm for AC^+ without node consistency. From the complexity point view, node consistency doesn't play any role in the complexity of our algorithm. In [28], we showed that AC^+ has the same pruning power as lookahead. The time complexity of lookahead for a translated program from a CSP is $O(e^3k^4)$. Indirectly, we know a bound for AC^+, which is higher than that of our algorithm. Consequently, our algorithm is more efficient than lookahead.

In AC4-UP, when every tuple in a CSP $\mathcal{A}$ is removed, each pair $\langle x_i, a_i \rangle$ in the CSP needs to be checked to see if there is a conflict caused by *unit_propagate** $(\mathcal{A}, \{x_i \rightarrow a_i\})$. Practically, we can find all the conflicts by applying arc-consistency and unit propagation alternately in a constant number of iterations. Unit propagation can be achieved in $O(e^2k)$ time separately, so the total running time is $O(e^2k + ek^2)$.

Chapter 6

Further Observations

ACUP has more pruning power than arc-consistency. What is the relation between ACUP and other space pruning techniques in the literature such as arc-consistency look ahead (ACLA) [2], path consistency. Do they coincide or not? In this chapter, we will answer these questions.

6.1 ACUP Vs. Arc-Consistency Look Ahead

The following example shows that there exists a CSP for which ACUP is more powerful in pruning the search space than ACLA.

Example 6.1.1 *Consider a* CSP *with* $X = \{x_1, x_2, x_3, x_4\}$, *all with domain* $\{0, 1\}$, *and the following constraints*

$$\begin{aligned}
c_1(x_1, x_2) &= \{(0,0), (0,1), (1,1)\}\\
c_2(x_2, x_3) &= \{(0,0), (0,1), (1,1), (1,0)\}\\
c_3(x_3, x_4) &= \{(0,0), (1,1)\}\\
c_4(x_4, x_5) &= \{(0,1), (1,0)\}\\
c_5(x_5, x_3) &= \{(1,1), (0,0)\}
\end{aligned}$$

c_3 and c_4 together show that there is no solution, but arc-consistency cannot detect the conflict between c_3 and c_4, since for each value of any one variable there is a value for the other variable. In fact, arc-consistency enforced on this CSP does not remove any domain value.

However, with ACUP, 0 is removed from D_{c_3}, so is 1, resulting in an empty domain, hence the result that this CSP has no solution can be generated without search.

Now consider ACLA. Suppose the current variable is x_1. Let us tentatively assign x_1 to 0. Now, since enforcing arc-consistency involves two variables, with a tentative assignment to the current variable, the consistency check is among triplets, much stronger than arc-consistency alone. However, one can see that since the spot of conflict is unrelated to variable x_1, inconsistencies cannot be detected. ■

In the above example we notice that if the current variable were chosen to be one in $\{x_2, x_3, x_4\}$, then the inconsistencies could be detected by arc-consistency. Is it enough to achieve the same pruning power as ACUP if arc-consistency look ahead is applied to all the future variables (let's call it ACLA$^+$)?

The following example shows that there exists a CSP in which ACUP has more pruning power than ACLA$^+$.

Example 6.1.2 *Consider a* CSP *with* $X = \{x_1, x_2, x_3, x_4\}$ *with* $D_{x_1} = D_{x_3} = D_{x_4} = \{0,1,2\}, D_{x_2} = \{0,1,2,3\}$, *and the following constraints*

$$\begin{aligned}
c_1(x_1,x_2) &= \{(0,0),(2,2),(1,0),(1,1),(2,1),(1,3)\}\\
c_2(x_2,x_3) &= \{(0,0),(1,2),(2,1),(2,2),(2,0),(3,1)\}\\
c_3(x_3,x_4) &= \{(0,0),(2,2),(1,0),(1,1),(1,2)\}\\
c_4(x_4,x_1) &= \{(0,2),(1,0),(2,0),(1,1),(2,2)\}
\end{aligned}$$

Obviously this CSP is arc-consistent. We can see that only the propagation from $x_1 = 0$ can cause a conflict propagation chain $\{x_1 = 0,\ x_2 = 0,\ x_3 = 0,\ x_4 = 0,\ x_1 = 2\}$. So 0 is removed from the domain of x_1, and $D_{x_1} = \{1,2\}$. Enforcing arc-consistency now, no domain can be reduced further. By removing the inconsistent tuples, the constraints become

$$\begin{aligned}
c_1(x_1,x_2) &= \{(2,2),(1,0),(1,1),(2,1),(1,3)\}\\
c_2(x_2,x_3) &= \{(1,2),(2,1),(2,2),(2,0),(3,1)\}\\
c_3(x_3,x_4) &= \{(0,0),(2,2),(1,0),(1,1),(1,2)\}\\
c_4(x_4,x_1) &= \{(0,2),(1,1),(2,2)\}
\end{aligned}$$

The propagation from $x_2 = 1$ can cause a another conflict propagation chain $\{x_2 = 1,\ x_3 = 2,\ x_4 = 2,\ x_1 = 2,\ x_2 = 2\}$. Therefore 1 is removed from D_{x_2}.

If we apply ACLA$^+$ to all future variables to this CSP, and x_2 is checked first, x_1 later, then 1 can not be removed from D_{x_2}.

Suppose 1 is tentatively assigned to the current variable x_2. After applying full arc-consistency check, the domains for $D_{x_3}, D_{x_4}, D_{x_1}$ become $\{2\}$, $\{2\}$, $\{2\}$ respectively. No domain becomes empty. Consequently, ACLA$^+$ cannot find the conflict caused by the assignment $x_2 \to 1$.

But if we apply ACUP, 2 can be removed from D_{x_2}. ■

The reason that ACLA$^+$ cannot find the conflict is that ACLA$^+$ only checks the conflicts in a fix order. That is, it does not check back to see if the removal of a conflict can cause another conflict. A conflict can depend on another conflict. If the conflicts do not occur in a certain order, ACLA$^+$ may not find all the conflicts.

We have seen the examples that show ACUP is more powerful in pruning the search space than ACLA$^+$ for some CSPs. Is ACUP strictly stronger than ACLA$^+$, or ACLA? The following example shows that ACLA$^+$, or ACLA is more powerful than ACUP for some CSPs.

Example 6.1.3 *Consider the* CSP *that has three variable* x, y, z *with* $D_x = \{0,1\}, D_y = \{1,2,3,4\}, D_z = \{1,2,3,4\}$*, and three constraints:*

$$\begin{aligned} c_{xy} &= \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle\} \\ c_{yz} &= \{\langle 1,3\rangle, \langle 2,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle, \langle 3,1\rangle, \langle 4,1\rangle, \langle 3,2\rangle\ \langle 4,2\rangle\} \\ c_{zx} &= \{\langle 1,0\rangle, \langle 2,0\rangle, \langle 3,1\rangle, \langle 4,1\rangle\} \end{aligned}$$

If we have the partial instantiation $x \to 0$, in ACLA, 3, and 4 are removed from D_y. Similarly, 3 and 4 are removed from D_z. In the next iteration, w.r.t. c_{yz}, 1 and 2 are removed from both D_y and D_z. As a result, ACLA can find this conflict.

By applying ACUP, no unit propagation can cause a conflict, so ACUP cannot reduce the domain. ■

We notice that there is a gap between ACUP and ACLA. Now we add ACUP to ACLA to make ACLA achieve at least the same pruning power as ACUP.

We only need to revise SELECT-VALUE-ARC-CONSISTENCY (cf. Section 2.4) in ACLA. The revised version of SELECT-VALUE-ARC-CONSISTENCY is given in Figure 6.1. We call it SVAC$^+$.

Example 6.1.4 *Consider above Example 6.1.2 again. A* CSP $\mathcal{A}(X, D, C)$ *with* $X = \{x, y, z\}$*, all with domain* $\{0,1,2\}$*, and the following constraints*

$$\begin{aligned} c_1(x,y) &= \{(1,1),(0,0),(0,1),(0,2),(2,1)\} \\ c_2(y,z) &= \{(1,1),(2,1),(0,0),(1,2)\} \\ c_3(z,x) &= \{(1,0),(0,1),(0,0),(2,0),(0,2)\} \end{aligned}$$

Suppose we use AC-4 as arc-consistency algorithm in SVRC$^+$. Initially $\langle x, 1\rangle$ is added into $\mathcal{L}$, which stores the conflict pairs in AC-4. Now applying arc-consistency, 1 is removed from domain D_x, and *counter*$(y, 1, x)$ becomes 1. When *unit_propagate*$^*(\mathcal{A}, \{y \to 1\})$ is applied, a conflict occurs. $y \to 1$ is added to $\mathcal{L}$. Applying arc-consistency again, 2, 1, and 0 are removed from D_y, D_z, and D_x respectively. ■

Complexity of SVAC$^+$

The general optimal time complexity for arc-consistency is $O(ek^2)$, where e is the number of constraints, and k is the cardinality of the largest domain. Since the original SVAC needs to check each value in the domain of

```
SVRC$^+$
Input: a CSP $\mathcal{A}(X, D', C)$, a current variable $x_i$
Output: a value for $x_i$, or a null if no consistent value
while $D'_i$ is not empty do
    select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$;
1   make $\mathcal{A}$ arc-consistent with the partial instantiation;
    if any future domain is empty (don't select $a$) then
        reset each changed domain to values before $a$ was selected;
    else
        conflict = false;
2       for each $x_j \in X$ do
            for each $a_j \in D'_j$ do
                if unit_propagate*($\mathcal{A}$, $\{x_j \rightarrow a_j\}$) returns "conflict"
                then
                    conflict $\rightarrow$ true;
                    break;
                end
            end
        end
        if conflict = true then
            reset each changed domain to values before $a$ was selected;
        else
            return $a$;
        end
    end
end
return null;
```

Figure 6.1: SVAC$^+$ subprocedure

the current variable, if we implement this procedure using the optimal arc-consistency algorithm AC-4, SVAC has a worst case time complexity $O(ek^3)$. Note that, in SVAC$^+$, each pair $\langle x_i, a_i\rangle$ in a CSP $\mathcal{A}(X, D, C)$ is checked using *unit_propagate*$(\mathcal{A}, \{x_i \rightarrow a_i\})$ which takes $O(e)$ time if it is combined with AC-4. Therefore, the loop in part 2 of ACLA$^+$ takes $O(e^2k)$, and the time complexity of SVAC$^+$ is $O((ek^2 + e^2k)k) = O(ek^2(k+e))$. ■

6.2 ACUP Vs. Path Consistency and i Consistency

Path consistency makes every path of length 3 (the path from x to y to z) consistent, while ACUP enforces consistency of every unit propagation chain. The following example shows the difference between path consistency and ACUP.

Example 6.2.1 *consider a* CSP$\mathcal{A}(X, D, C)$ *that has three variables* x, y, z *with domain* $D_x = D_y = D_z = \{0, 1, 2, 3\}$*, and three constraints:*

$$\begin{aligned}
c_{xy} &= \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle\}\\
c_{xz} &= \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle\}\\
c_{zy} &= \{\langle 0,2\rangle, \langle 1,3\rangle, \langle 2,0\rangle, \langle 3,1\rangle, \langle 1,2\rangle, \langle 0,3\rangle, \langle 3,0\rangle, \langle 2,1\rangle\}
\end{aligned}$$

Obviously, this CSP is arc-consistent. But we can see that for any pair $\langle a_i, a_j\rangle \in c_{xy}$, there is no value a_k for z such that $\langle a_i, a_k\rangle \in c_{xz}$, and $\langle a_k, a_j\rangle \in c_{zy}$. Applying path consistency, the domains are reduced to empty.

But applying ACUP, since there is no unit propagation in this CSP, no domain value can be reduced. ■

It can be seen that there is no unit propagation in this CSP. So ACUP cannot be applied here. Therefore path consistency has more pruning power than ACUP for this CSP.

Definition 6.2.2 *The distinct length of a unit propagation chain C is the number of different variables in C.*

Example 6.2.3 *The unit propagation chain* $\{x_1 \rightarrow 0, x_2 \rightarrow 1, x_3 \rightarrow 0, x_4 \rightarrow 1, x_1 \rightarrow 1\}$ *has length 4 because the different variables in the propagation chain are* $\{x_1, x_2, x_3, x_4\}$. ■

Obviously, we have the following proposition:

Proposition 6.2.4 *Given a* CSP, *if there is a conflict chain whose length is i, and the* CSP *is $i-1$ consistent, then ACUP can find the inconsistency while $i-1$ consistency cannot find it in this* CSP.

Example 6.2.5 *consider a* CSP $\mathcal{A}(X, D, C)$ *that has 4 variables* x_1, x_2, x_3, x_4 *with domain* $D_{x_1} = D_{x_2} = D_{x_3} = D_{x_4} = \{0, 1\}$, *and four constraints:*

$$\begin{aligned}
c_{x_1,x_2} &= \{\langle 0,1\rangle, \langle 1,0\rangle\}\\
c_{x_2,x_3} &= \{\langle 1,0\rangle, \langle 0,1\rangle\}\\
c_{x_3,x_4} &= \{\langle 0,1\rangle, \langle 1,0\rangle\}\\
c_{x_4,x_1} &= \{\langle 1,1\rangle, \langle 0,0\rangle\}
\end{aligned}$$

$c_{x_3,x_4} =$

Ob$c_{x_4,x_1} =$this CSP is arc-consistent. We can also check that it is path consistent. But there is a unit propagation chain $\{x_1 \to 0, x_2 \to 1, x_3 \to 0, x_4 \to 1, x_1 \to 1\}$, so *unit_propagate*$^*(\mathcal{A}, \{x \to 0\})$ returns a conflict. Therefore ACUP can find the inconsistency while path consistency can not for this CSP. ■

From the above example, we see that ACUP and i consistency are different techniques that can be used to prune the search space. When CSPs have many unit propagations, ACUP has extra power as compared with i consistency. Furthermore, the time complexity for ACUP is much lower than i consistency. The following proposition defines the condition under which i consistency can achieve at least the same pruning power as ACUP.

Proposition 6.2.6 *Given a* CSP, *if the length of every unit propagation chain is at most i, then any value removed by* ACUP *can be removed by i consistency for any $i > 2$.*

Proof We only consider the case when there is a conflict in the unit propagation chain $\{x_1 \to a_1, \ldots, x_i \to a_i, x_1 = b\}$ with distinct length i, where $\{a_1 \neq b\}$. For the pair $\langle a_1, a_i\rangle$, we can't find consistent any value for $\{x_3, \ldots, x_{i-1}\}$ to achieve $a_2 \in D_{x_2}, \ldots, a_{i-1} \in D_{x_{i-1}}$ simultaneously. Therefore, i consistency can also find the conflict, and remove a_1 from the domain D_{x_i} ■

From above discussions, we see that ACUP is a new consistency technique. It is strictly stronger than arc-consistency and overlaps with arc-consistency look ahead and path consistency.

Chapter 7

Summary and Future Work

7.1 Summary

In this thesis, we present an algorithm for ACUP, and give the proof of the algorithm. The time complexity of our algorithm for ACUP is $O(e^3k^3)$, which is lower than that given in [28].

We also compare ACUP with the other consistency techniques in the literature. We show that ACUP neither dominates nor is dominated by arc-consistency look ahead and i consistency.

SVAC in arc-consistency look ahead chooses a value for the current variable and checks if this assignment causes a conflict. It tentatively assigns a value to the current variable, then performs arc-consistency checks. If there is an empty domain, it means that this tentative assignment cannot be consistently extended to the whole CSP, so another tentative value is chosen for the current variable, and the consistency check is enforced again. In this manner, SVAC is much stronger than arc-consistency alone since it performs arc-consistency checks under a tentative assignment. But if the conflict lies in the other variables instead of the current variable, SVAC cannot find it. ACUP can find this conflict if there exists a conflict unit propagation chain in the CSPs. Choosing every future variable as the current variable, assigning a tentative value for the current variable, and performing consistency check cannot make up this gap if the conflicts happen in a different order from that of choosing the current variable. A loop is needed to run this check until no further conflict can be found. Note that SVAC takes $O(ek^3)$ if the optimal arc-consistency algorithm is adopted. If SVAC is applied to all future variables, it will take $O(e^2k^3)$ time. If we want to achieve at least the same pruning power as ACUP, in a straightforward manner, we need a loop to run the conflict checking until no further conflict can be found. This approach takes $O(e^3k^4)$ time. The difference between ACUP and SVAC enables us to design a new algorithm called SVAC$^+$ to incorporate ACUP into SVAC. SVAC$^+$ can find all the conflicts in unit propagation chains, and takes $O(ek^2(e+k))$ time, which is much lower than $O(e^3k^4)$.

i consistency makes i variables in a CSP consistent. i consistency ($i > 2$) is more powerful than arc-consistency. But if the length of a conflict chain in a CSP is greater than $i-1$, i consistency still can not achieve the same pruning power as ACUP. In order to have at least the same pruning power as ACUP, i consistency requires $O(2^i(ek)^{2i})$ time for a brute force algorithm. It is much higher that $O(e^3k^3)$ for ACUP.

7.2 Future Work

In this thesis, we compare one step lookahead in Smodels with arc-consistency. It has been shown that one step lookahead is strictly stronger than arc-consistency (2 consistency) in pruning the search space. The question arises if i step lookahead is strictly stronger than $i+1$ consistency. Does i-consistency with unit propagation achieve the same pruning power as i step lookahead ?

Another interesting topic is the difference between *expand* and *arc-consistency*. *Arc-consistency* is strictly stronger than *expand*, but the optimal time complexity of *arc-consistency* is $O(ek^2)$, which is the same as the time complexity of *expand* for the translated programs [28]. Can we improve the pruning ability of *expand* and achieve the same pruning power as arc-consistency in the same time complexity?

An understanding of the relationships between different approaches often reveals insights in these approaches and the corresponding techniques. Despite their proximity, in the past constraint propagation in answer set programming has rarely been compared to consistency techniques in solving CSPs. By establishing the relations between lookahead and arc-consistency, this thesis is the first step in this direction. Many more questions remain, among which *bounds consistency* as used for CSPs appears particularly interesting, not only because it can be extremely effective in reducing domains for some CSPs, but also because it appears to have no counterpart in answer set solvers. We would like to see the arsenal of constraint propagation techniques applied to answer set programming also include a form of bounds consistency, so that the search space in large numerical domains can be reduced quickly.

Finally, our result is based on a standard encoding from CSPs to answer set programs. It is interesting to investigate whether the same conclusion can be drawn under different encodings.

Bibliography

[1] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[2] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[3] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *ACM*, 23(4):733–742, October 1976.

[4] W. Faber and G. Pfeifer. http://www.dlvsystem.com/. Technical report, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184), Database and Artificial Intelligence Group, 1996.

[5] J.W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[6] E.C. Freuder. Synthesizing constraint expressions. *CACM*, 21(11):958–966, 1978.

[7] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, Pa., 1979.

[8] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[9] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, 1988.

[10] Ian Gent. Arc consistency in SAT. In *Proc. ECAI 2003*, pages 121–125, 2002.

[11] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, pages 263–313, 1980.

[12] G. Huang, X. Jia, C. Liau, and J. You. Two-Literal logic programs and satisfiability representation of stable models: A comparison. In *Proc. 15th Canadian Conference on AI, LNCS, Springer*, pages 119–131, 2001.

[13] Xiumei Jia. Using domain dependent knowledge for planning in answer set programming. MSc thesis, 2003.

[14] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, pages 275–286, 1990.

[15] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[16] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[17] W. Marek and M. Truszczynski. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[18] K. Marriott and P. Stucky. *Programming with Constraints.* MIT Press, 1998.

[19] I. Niemel and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming,Bonn, Germany*, pages 289–303, September 1996.

[20] I. Niemel and P. Simons. Smodels - An implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, volume 1265 of Lecture Notes in Artificial Intelligence*, pages 420–429, July 1997.

[21] 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.

[22] S. Padmanabhuni. *Logic programming with stable models for constraint satisfaction.* PhD thesis, University of Alberta, 2000.

[23] Srinivas Padmanabhuni. *Logic Programming with Stable Models for Constraint Satisfaction.* PhD thesis, University of Alberta, 2000.

[24] P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Technical Report Research Report 35, Helsinki University of Technology,Helsinki, Finland, September 1995.

[25] P. Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.

[26] P. Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.

[27] T. Walsh. CSP vs. SAT. In *Proc. Principles and Practice of Constraint Programming*, pages 441–456, 2000.

[28] Jia-Huai You and Guiwen Hou. Arc-consistency+unit propagation = lookahead. Technical report, 2004.