HISTOGRAM ARC CONSISTENCY AS A VALUE ORDERING HEURISTIC

by

Wei Liu
M.A.Sc, Dalhousie University, 2000
B.Eng, Xian Institute of Architecture and Technology, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
March 2004

APPROVAL

Name:	Wei Liu
Degree:	Master of Science
Title of thesis:	HISTOGRAM ARC CONSISTENCY AS A VALUE ORDERING HEURISTIC
Examining Committee:	Dr. Ramesh Krishnamurti Chair
	Dr. William S. Havens, Senior Supervisor
	Dr. Lou Hafer, Supervisor
	Dr. Arthur Kirkpatrick, SFU Examiner

Date Approved: March 12, 2004

Abstract

The Constraint Satisfaction Problem (CSP) is NP-hard. Finding solutions requires searching in an exponential space of possible variable assignments. Good value ordering heuristics are essential for finding solutions to CSPs. Such heuristics estimate the probability that any particular variable assignment will appear in a globally consistent solution. Unfortunately, computing solution probabilities exactly is also NP-hard. Thus estimation algorithms are required. Previous results have been very encouraging but computationally expensive.

In this thesis, we present two new algorithms, called Histogram Arc Consistency (HAC) and μ Arc Consistency (μAC), which generate fast estimates of marginal solution counts during constraint propagation. This information is used as value ordering heuristics to guide backtrack search.

We conducted experiments using randomly generated CSPs. Our experimental results on random CSPs show that these methods indeed provide significant heuristic guidance compared to previous methods while remaining efficient to compute.

To my family

Acknowledgments

I would like to thank my senior supervisor Dr. Bill Havens and my supervisor Dr. Lou Hafer for their guidance through out this research. I would also like to thank Dr. Ted Kirkpatrick for being in my examining committee and reviewing my thesis.

I am grateful to Dr. Michael Horsch for his advice and providing the original source code of the pAC algorithm.

Finally, I would like to thank my dear parents, Shuhua Jin and Bocai Liu, for taking care of my daughter while I am in school and my beloved husband, Eric Fu, for his love and support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

The Constraint Satisfaction Problem (CSP) has been applied in different application areas to solve real life problems. Basically, a CSP problem consists of a set of variables and a set of constraints. Each variable is associated with a set of domain values. The objective of solving a CSP is to assign each variable with a domain value such that all constraints are satisfied. Problems such as workshop scheduling, resource allocation and visual image interpretation, *etc.*, can be formulated as constraint satisfaction problems and different constraint solving techniques can be applied to solve them.

One of the traditional algorithms to solve CSP problems is called the chronological backtrack search algorithm. In this algorithm, variables are assigned one at a time, such that the new assignment is compatible with all previous assignments. When no value can be found in a variable domain that is consistent with all previous variable assignments, the algorithm backtracks to the last assigned variable and a new value is chosen. The chronological backtrack search algorithm stops when either a solution is found or a conclusion is made that no solution exists. The worst-case complexity of this algorithm is exponential in terms of the number of variables, because in the worst case this algorithm has to generate all possible combinations of domain values to find a solution (or stop without a solution).

Value ordering heuristic has been used to guide the search algorithm towards areas of search space that are likely to contain solutions. An efficient value ordering heuristic would be to choose a value which appears in the most solutions. In this thesis, we use the marginal solution count to represent the number of times that a domain value appears in all solutions.

However, this information is only available after all solutions have been found. Thus we need to estimate these counts.

Various researches have been done on generating approximations of marginal solution counts. For tree-structured CSPs, Meisels *et al.* [16], Pearl [21], and Neopolitan [19] have shown that the exact marginal solution counts can be found in polynomial time. For general structured CSPs, Detcher and Pearl [5] suggested to first reduce the complete constraint network to single spanning tree and use the exact marginal solution counts obtained in the simplified problem as an approximation to the original problem. Vernooy and Havens [24] converted the original problem to multiple spanning trees. The approximated marginal solution counts for the whole problem are obtained by combining the marginal solution counts of each subproblems. Meisel *et al.* [16] suggested converting the constraint network to a Bayesian network such that the marginal solution counts are proportional to the marginal probabilities in the network. The solution probability is then obtained through probability updating in the network.

In these algorithms, loops in general CSP networks can cause gross over counting problems. They are ignored either by deleting the constraints that form a loop from the network [5] or by assuming independence among subproblems [24, 16]. These assumptions introduce errors to the estimations. Thus, the heuristic information obtained by these algorithms may not correspond to the ordering in the original problem.

The Probability Arc Consistency (pAC) algorithm introduced by Horsch and Havens [14] maintains the structure of the original CSP problem and deals with the over counting problem directly. In this algorithm, using local consistency technique, the approximated solution probabilities are obtained by iteratively propagating the number of constraints that are satisfiable. The program stops when the estimating process converges, *i.e.*, changes in the solution probabilities are less than a threshold, ϵ. Otherwise, the program halts without convergence when the total number of iterations has exceeded the predefined maximum number of iterations.

Experimental results showed that the ordering of domain values provided by the pAC algorithm is highly correlated with the ordering implied by the exact solution probabilities. The use of this value ordering as a heuristic to guide the search algorithm has been found to reduce the number of backtracks in random CSPs by as many as two orders of magnitude [12]. However the convergence of the algorithm can be very slow for moderate sized problems. For larger problems, pAC's runtime is competitive or superior to previous algorithms. When the problem is small, the cost of computing the pAC outweighs the benefit of reducing search cost.

In this thesis, we present two new algorithms, called Histogram Arc Consistency(HAC) and μ Arc Consistency (μAC). The HAC algorithm approximates marginal solution counts by propagating the number of constraints that are satisfiable. Local information is used to estimate these global properties. A major difference between HAC and pAC is that while the pAC algorithm propagates values of solution probabilities until they do not change, the HAC algorithm stops when no value is deleted from variable live domains thus guaranteeing convergence. We are aware that the marginal solution counts obtained by our algorithms are also not exact, but we argue that the exact marginal solution counts are not necessary when they are used as value ordering heuristics[12]. Our very coarse approximation is good enough to guide the search algorithm effectively.

The HAC algorithm is similar to Geelen's value ordering heuristic.[6] Geelen's algorithm calculates products of the remaining domain sizes of the future variables. It represents an upper bound on the number of possible solutions resulting from the assignment. The value with the highest valuation is chosen. The μAC algorithm takes one step further from the HAC algorithm. It assumes accuracy of marginal solution counts generated by the HAC algorithm. It removes all domain values with small HAC valuations from variable live domains and these forced domain reductions are propagated through the constraint network. Hopefully, by aggressively reducing the size of the search tree we can find a solution quickly and with less backtracks.

1.2 Objective of the Research

This thesis presents two new algorithms, Histogram Arc Consistency (HAC) and μ Arc Consistency (μAC). These algorithms are simplified versions of the pAC algorithm introduced by Horsch and Havens [12]. They generate rough estimations of the marginal solution counts for constraint satisfaction problems. This information is used as a value ordering heuristic to guide the search algorithm.

We want to show that, when used as value ordering heuristics, the estimated marginal solution counts generated by our algorithms, although not as accurate as those generated by the pAC algorithm, are good enough to guide the search algorithm and take less CPU time to compute.

These proposed algorithms are tested using randomly generated CSPs. Numbers of backtracks used to find first solutions and CPU times are collected. Performances of these two new algorithms are compared with the traditional arc consistency algorithm and with the pAC algorithm.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 introduces the background information of constraint satisfaction problems. In this chapter, we first give definitions of some preliminary concepts and then we introduce the traditional algorithm to solve CSPs. In Chapter 3, we review previous works on approximating the marginal solution counts. In Chapter 4, we present our proposed algorithms, HAC and μAC. We also discuss how to combine HAC and μAC algorithms with the search algorithm to solve CSPs. In Chapter 5, we show some experimental results. And finally, Chapter 6 concludes this thesis and points out the possible future work.

Chapter 2

Background Information

2.1 Constraint Satisfaction Problems

2.1.1 Definitions

In this section, we review some preliminary concepts of the constraint satisfaction problem as presented in [23].

A *Constraint Satisfaction Problem* (CSP) is a triple $< Z, D, C >$, where $Z = \{x_1, x_2, ..., x_n\}$ is a finite set of variables; $D = \{D_{x_1}, D_{x_2}, ..., D_{x_n}\}$ is a finite set of domains, in which $D_{x_i} \in D$ consists of all the possible values that can be assigned to variable x_i; C is a finite set of constraints on an arbitrary subset of variables in Z. For example, $C_{x_1,x_2,...,x_k}$ is a k-ary constraint which restricts values that $x_1, x_2, ...,$ and x_k can take simultaneously. For convenience but without loss of generality, we restrict constraints in this thesis to be unary and binary constraints [20].

A *label* is a variable-value pair $< x, v >$ representing the assignment of the value $v \in D_x$ to the variable x.

A *compound label* is the simultaneous assignments of values to a set of variables. We use the compound label $(< x_1, v_1 >< x_2, v_2 >, ..., < x_k, v_k >)$ to represent the simultaneous assignments: $x_1 = v_1, x_2 = v_2, ..., x_k = v_k$.

A *solution* of a CSP is a compound label for all variables in Z which satisfies all constraints.

A *marginal solution* of a CSP is the projection of the set of solutions to a CSP onto the combined domain of a subset of the variables. [14]. A *marginal solution count* is the number of times that the marginal solution appears in all the solutions.

2.1.2 CSP Formalization

To formalize a problem as a CSP, we need to identify the set of variables, Z, the set of domains, D, and the set of constraints, C.

Consider the n-queens problem as an example. The objective of this problem is to arrange n queens on an $n \times n$ chest-board such that no queens are attacking each other, *i.e.*, be arranged on the same row, or same column or same diagonal. A solution for the 8-queens problem is shown in Figure 2.1.

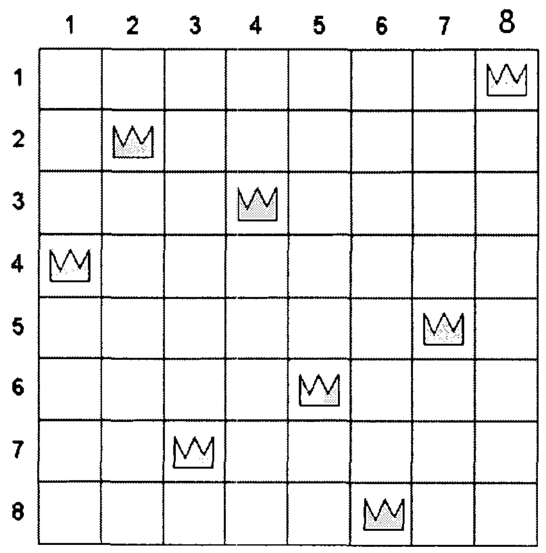


Figure 2.1: 8-queen Problem

To formulate the 8-queen problem, we can make each row a variable, that is, $Z = \{Q_1, Q_2, ..., Q_8\}$. Each of these eight variables can take one of the eight columns as its value, i.e., $D_{Q_1} = D_{Q_2} = ... = D_{Q_8} = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Constraints in this problem are represented using equations and inequalities. The constraint that no two queens are on the same column can be represented as : $\forall\ i, j,\ Q_i \neq Q_j$. To make sure that no two queens are on the same diagonal, we can use the following constraint: $\forall\ i,\ j$, if $Q_i = a$, $Q_j = b$, then $|(i - j)| \neq |(a - b)|$.

Another CSP example is called the map coloring problem, as shown in Figure 2.2. Given a map and a number of colors, can we color all regions with given colors such that neighboring regions are in different colors?

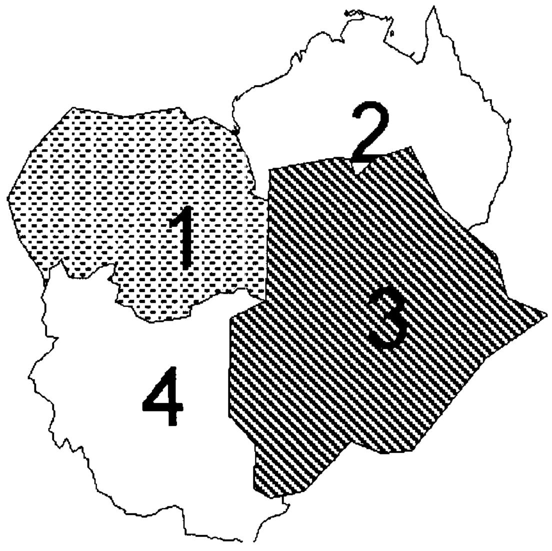


Figure 2.2: A Map Coloring Problem

To formalize this problem, we can make each of the regions a variable. So for the problem shown in Figure 2.2, there are 4 variables, $Z = \{x_1, x_2, x_3, x_4\}$. Each region is given three colors to choose from, {R-Red, G-Green, B-Blue}, that is, $D_{x_1} = D_{x_2} = D_{x_3} = D_{x_4} = \{R, G, B\}$. There is a "not-equal" constraint between every pair of adjacent regions, that is: $C = \{C_{x_1,x_2}, C_{x_1,x_4}, C_{x_1,x_3}, C_{x_2,x_3}, C_{x_3,x_4}\}$.

Sometimes, when constraints are too arbitrary to be formulated using equations or inequalities, we can use boolean matrix to represent them. In these matrices, tuples that are allowed are represented using 1's and tuples that are forbidden are represented using 0's.

To demonstrate, Figure 2.3 shows a boolean matrix representing the constraint C_{x_1,x_2} of the above map coloring problem.

¿From this constraint table we can see that for constraint C_{x_1,x_2}, allowed tuples are: $(< x_1, R >< x_2, G >), (< x_1, R >< x_2, B >), (< x_1, G >< x_2, R >), (< x_1, G >< x_2, B >), (< x_1, B >< x_2, R >), (< x_1, B >< x_2, G >)$.

A CSP can also be represented using *constraint graphs* in which each node represents

$$\begin{array}{cc|ccc} & & & X_2 & \\ & & R & G & B \\ \hline & R & 0 & 1 & 1 \\ X_1 & G & 1 & 0 & 1 \\ & B & 1 & 1 & 0 \end{array}$$

Figure 2.3: Constraint 01-table for the map coloring problem shown in Figure 2.2

a variable in Z, and each edge represents a constraint in C. The constraint graph for the above map coloring problem is shown in Figure 2.4.

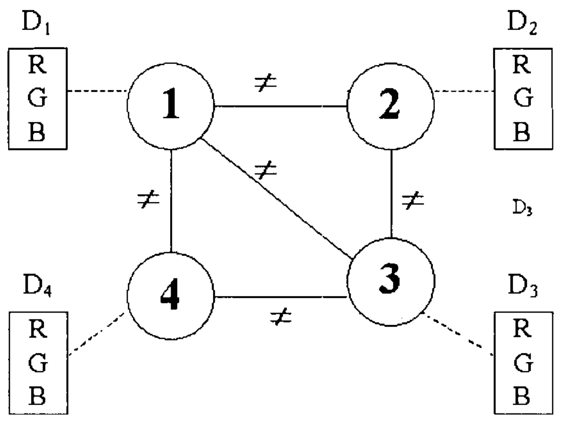


Figure 2.4: Constraint graph for the map coloring problem shown in Figure 2.2

2.2 The Chronological Backtrack Search Algorithm

A traditional algorithm to solve constraint satisfaction problems is called the chronological backtrack search algorithm [4]. In this algorithm, variables are assigned to their domain values one at a time in a predefined order. Assigning a value to a variable from its domain is called the *instantiation* of the variable. After a variable is instantiated, all constraints involving this variable are checked to make sure that this new assignment is compatible with all previous assignments. If no constraint violation is found, the algorithm goes ahead to instantiate the next variable in the order. Otherwise, the algorithm chooses a different value for the variable from its domain. If no such value can be found, the algorithm backtracks, which means that the last instantiated variable is revisited and another value, if available, is chosen. The algorithm ends when either a solution is found or a conclusion is made that no such solution exits.

The pseudo-code for the chronological backtrack search (BT) algorithm is shown in Figure 2.5. The $ChronologicalBacktracking(Z, D, C)$ procedure calls the BT procedure at line 2 with arguments U, P, D and C. U is a vector containing the set of variables waiting to be assigned to a value. Initially, U equals to Z. P is a vector containing the set of variable-value pairs which satisfy all constraints. Initially, P is empty. D is the set of domains for variables in Z and C is the set of constraints. The BT procedure starts with picking a variable x from the set of unassigned variables at line 5, and at line 7, it picks a value v from the domain of variable x. Line 9 checks if assigning the value v to the variable x violates any constraints. If no constraint violation is found, line 12 adds the label $< x, v >$ to the compound label P; deletes variable x from the set U and recursively calls the BT algorithm to instantiate another variable from the set U. If no value can be found in the variable domain that satisfies all constraints, the BT procedure returns NIL at line 16. The previous assignment is discarded and a new value is chosen. The procedure either stops at line 2 with a solution or it stops without a solution after exhausts all combinations of domain values.

In the worst case, the chronological backtrack algorithm has to try all possible combinations of domain values and all constraints have to be checked. The worst case complexity of the chronological backtrack algorithm is $O(d^n e)$, where n represents the number of variables, d represents the size of the variable domain and e is the number of constraints. When the size of the problem becomes larger, this algorithm becomes extremely slow.

2.3 Improving the Backtrack Search Algorithm

A number of algorithms have been developed to improve the efficiency of the chronological backtrack search algorithm. In this section, we first introduce the arc consistency algorithm, and then, we show you how to use heuristics to guide the search algorithm.

```
procedure ChronologicalBacktracking $(Z, D, C)$
begin
    BT(Z, {}, D, C);
end

procedure BT(U, P, D, C)
begin
    if U = {} then return $(P)$
    else
        begin
            Pick a variable $x$ from U
            repeat
                pick a value $v$ from $D_x$
                delete $v$ from $D_x$
                if P + {< $x, v$ >} violates no constraints then
                    begin
                        Result ←
                         BT(U - {$x$}, P + {< $x, v$ >}, D, C)
                        if Result ≠ NIL then return (Result)
                    end
             until $(Dx = \{\})$
             return NIL /*no solution*/
         end /*of else*/
end /*of BT*/
```

Figure 2.5: Psuedo code for the chronological backtrack search algorithm

2.3.1 Enforcing Arc Consistency

An arc (x_i, x_j), which corresponds to a binary constraint C_{x_i,x_j} in the constraint graph of a CSP, is *arc-consistent*(AC) if and only if for every value v_i in the domain of x_i, there is a value in the domain of x_j which is compatible with $< x_i, v_i >$. A CSP is arc consistent if and only if every arc in its constraint graph is arc-consistent.[23] There is also a lighter version of arc consistency in a CSP, called *directional arc consistency* (DAC) [5]. A CSP is directional arc consistent if and only if for every value v_i in the domain of x_i, there is a compatible label $< x_j, v_j >$, such that $i < j$.

The subset $D'_{x_i} \subseteq D_{x_i}$ is called the *live domain* of variable x_i which represents those values in its domain D_{x_i} which are consistent with the current compound label. [10]

By achieving arc consistency, we can remove values that can not be part of any solutions from variable domains, thus reduce the search space without ruling out any solutions. The REVISE(x_i, x_j) procedure, as shown in Figure 2.6, revises the live domain of variable x_i based on the live domain of variable x_j [15]. In this procedure, for each value v_i in the live domain of x_i, line 4 checks if it is supported by a value in the domain of its neighboring variable x_j. If it is not, line 6 removes the value v_i from the live domain of x_i.

```
precedure REVISE($x_i, x_j$)
1.  begin
2.      changed ← false
3.      for each $v_i \in D_i$ do
4.          if no $v_j \in D_j$ such that $C_{x_i,x_j}(v_i, v_j)$ then
5.              begin
6.                  $D_i \leftarrow D_i - \{v_i\}$
7.                  changed ← true
8.              end
9.      return changed
10. end
```

Figure 2.6: The REVISE procedure

Deleting a value from a variable domain may cause other values in its neighboring variables become inconsistent, thus should be deleted as well. To achieve arc consistency for the whole problem, the above REVISE procedure needs to be performed repeatedly until no value is deleted from any variable domain.

One of the AC achievement algorithms, AC-1, is shown in Figure 2.7. In this algorithm,

```
procedure AC1(Z, D, C)
1. begin
2.     changed ← true
3.     while (changed)
4.         changed ← false
5.         for each constraint $C_{x_i,x_j} \in C$ do
6.             changed ← changed ∨ REVISE($x_i, x_j$)
7. end
```

Figure 2.7: The AC-1 algorithm

arc consistency is checked for every pair of constraints in C. If any variable domain is changed during the REVISE procedure, all constraints will be reexamined.

The traditional arc consistency algorithm can also be represented using mathematical form as:

$$\forall a_i \in D_i, m(a_i) = \prod_{x_j \in N(x_i)} \sum_{a_j \in D_j} C_{i,j}(a_i, a_j)m(a_j)$$

where

- $m(a_i)$ is the membership value associated with the domain value a_i
- $\prod$ represents the boolean product, which is the logical AND operation, $\wedge$.
- $\sum$ represents the boolean sum, which is the logical OR operation, $\vee$.
- $N(x_i)$ represents neighboring variables of x_i.

In the above function, $m(a_i)$ is a boolean value associated with the domain value a_i of variable x_i. It represents the membership of value a_i in the live domain of variable x_i. Initially, all membership values are set to 1, meaning all values are in the live domain. C_{x_i,x_j} represents the constraint between variable x_i and x_j. $C_{x_i,x_j}(a_i, a_j)$ returns true if the tuple $\{x_i = a_i, x_j = a_j\}$ is allowed, and false otherwise. The logical OR operation tells us if there is ANY value in the neighboring variable x_j supporting the value a_i of the variable x_i. If the disjunction operation OR comes out to be false for any of the neighboring variables of x_i, then the conjunction operation AND will be false, which means that the value a_i is not supported, thus should be removed. To achieve arc consistency for the whole problem, these membership values are updated repeatedly until no values are deleted from any variable domains.

The arc consistency algorithm only checks for local consistency. It does not guarantee to find solutions. An efficient way to solve CSPs is to combine arc consistency with the search algorithm. The arc consistency algorithm can be used in the preprocessing to reduce the search space before search techniques are applied. It can also be used during search to prune off search space after each assignment is made.

There are different ways to combine arc consistency with the backtrack search algorithm [8, 18]. These algorithms differ from the degrees by which arc consistencies are achieved. In this thesis, we introduce the Full Look-ahead algorithm. This algorithm is also the base operation in our proposed algorithms.

In the Full Look-ahead algorithm, when a variable is labeled, all related constraints are checked for consistency. Domain values that are not compatible with committed labels are removed. The removal of values from variable domains are propagated through the constraint network. Full look-ahead algorithm ensures that for every value in every unlabeled variable, there is a compatible value in each of its neighboring variables. The pseudo code for the Full-Lookahead algorithm is shown in Figure 2.8. The Full-Lookahead algorithm searches for solutions through recursively calling the FL(U, P, D, C) procedure. In the FL procedure, line 3 performs arc consistency for every constraint. If none of the variable domains becomes empty after arc consistency is achieved, line 8 picks a value v_i from the live domain of variable x_i and assigns it to variable x_i. After this assignment, the live domain of variable x_i is updated using the UPDATE procedure, *i.e.*, $D'_{x_i} = \{v_i\}$. The updated live domain is propagated through the constraint network. This is done by the recursive call to the FL procedure at line 12 . If, after arc consistency propagation, any variable domain becomes empty, the program backtracks, which means that formerly deleted domain values are restored and the last assigned variable is re-assigned to a different value. The program either stops after every variable is assigned a value or exhausts the whole search space and concludes that no solution exists.

Experimental results show that [18], in most cases, the arc consistency algorithm is very efficient in reducing the search space. When combined with search algorithms, fewer number of backtracks are required to find solutions than using the chronological backtrack search algorithm alone. However, one disadvantage with the traditional arc consistency algorithm is that it only distinquishes domain values that are in variables' live domains from those

```
precedure Full-Lookahead($Z, D, C$)
1. begin
2.     FL($Z, \{\}, D, C$)
3. end

precedure FL($U, P, D, C$)
1. begin
2.     if (U = {}) then return (P)
3.     AC-x(Z, D, C)
4.     if no domain in D is empty then
5.         begin
6.             pick a variable $x_i$ from $U$
7.             repeat
8.                 pick a value $v_i$ from $D_{x_i}$
9.                 delete value $v_i$ from $D_{x_i}$
10.                $D' \leftarrow$ UPDATE($D, < x_i, v_i >$);
11.                Result $\leftarrow$
12.                    FL($U - \{x_i\}, P + \{< x_i, v_i >\}, D', C$)
13.                if(Result $\neq$ NIL) then return (Result)
14.            until $D_{x_i} = \{\}$
15.        end /*of if*/
16.     return NIL
17. end /*of FL*/

precedure UPDATE($D, < x, v >$)
1. begin
2.     $D' \leftarrow D$
3.     $D'_{x_i} \leftarrow \{v_i\}$
4.     return $D'$
5. end
```

Figure 2.8: The Full Look-ahead algorithm
AC-x() represents any arc consistency algorithm

that are not, it does not provide any heuristic ordering for these domain values.

A CSP example is shown in figure 2.9. The problem has three variables $Z = \{x_0, x_1, x_2\}$. Each variable has domain size three and values a, b and c. So, $D_{x_1} = D_{x_2} = D_{x_3} = \{a, b, c\}$. There are three constraints, $C = \{C_{x_0,x_1}, C_{x_0,x_2}, C_{x_1,x_2}\}$. These constraints are represented using 0-1 tables as shown in figure 2.10. In these tables, allowed tuples are represented by 1's and not-allowed tuples are represented by 0's.

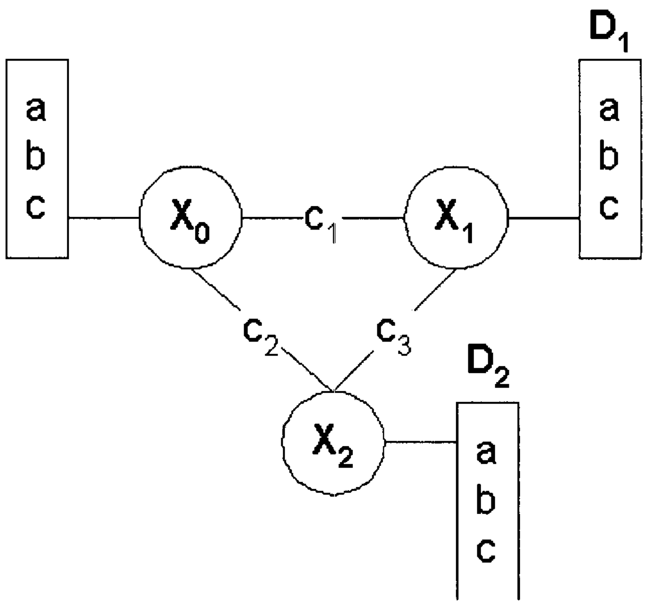
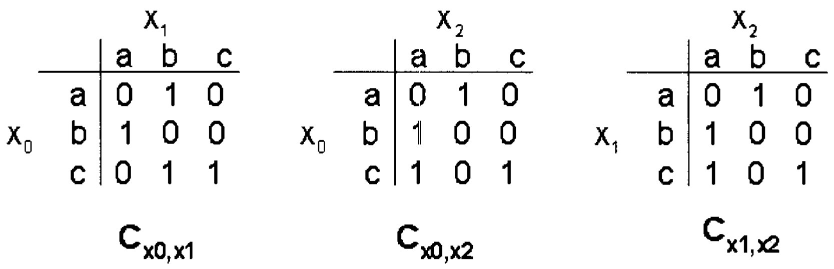


Figure 2.9: A csp example



	X_1	a	b	c
	a	0	1	0
X_0	b	1	0	0
	c	0	1	1

$C_{x0,x1}$

	X_2	a	b	c
	a	0	1	0
X_0	b	1	0	0
	c	1	0	1

$C_{x0,x2}$

	X_2	a	b	c
	a	0	1	0
X_1	b	1	0	0
	c	1	0	1

$C_{x1,x2}$

Figure 2.10: Constraint tables

Figure 2.11 shows an example of revising the live domain of variable x_0 base on live domains of it's neighbouring variables x_1 and x_2. As we can see, membership values associated with domain elements of x_0 are first projected from constraints C_{x_0,x_1} and C_{x_0,x_2} respectively using the logical OR operation. And then, these membership values are combined using the logical AND operation. To achieve arc consistency for the whole problem, these projection and combination operations need to be performed on every constraint repeatedly until no domain is changed.

The CSP that is arc consistent is shown in figure 2.12. From these membership valuations we can see that, after arc consistency propagation, all domain values are in their variable's live domains. Since the traditional arc consistency algorithm does not give these domain values any priority, to instantiate a variable, the search algorithm has to arbitrarily choose a value from the variable's live domain. For the above CSP, there are 3 solutions and they are: $\langle x_0 = c, x_1 = b, x_2 = a\rangle$, $\langle x_0 = c, x_1 = c, x_2 = a\rangle$, and $\langle x_0 = c, x_1 = c, x_2 = c\rangle$. To find a solution, the search algorithm first assigns $x_1 = a$. This assignment leads to a dead-end. The search algorithm backtracks. Next, the search algorithm assigns $x_1 = b$ and backtracks again. Finally, the search algorithm assigns $x_1 = c$ and finds a solution after 2 backtracks.

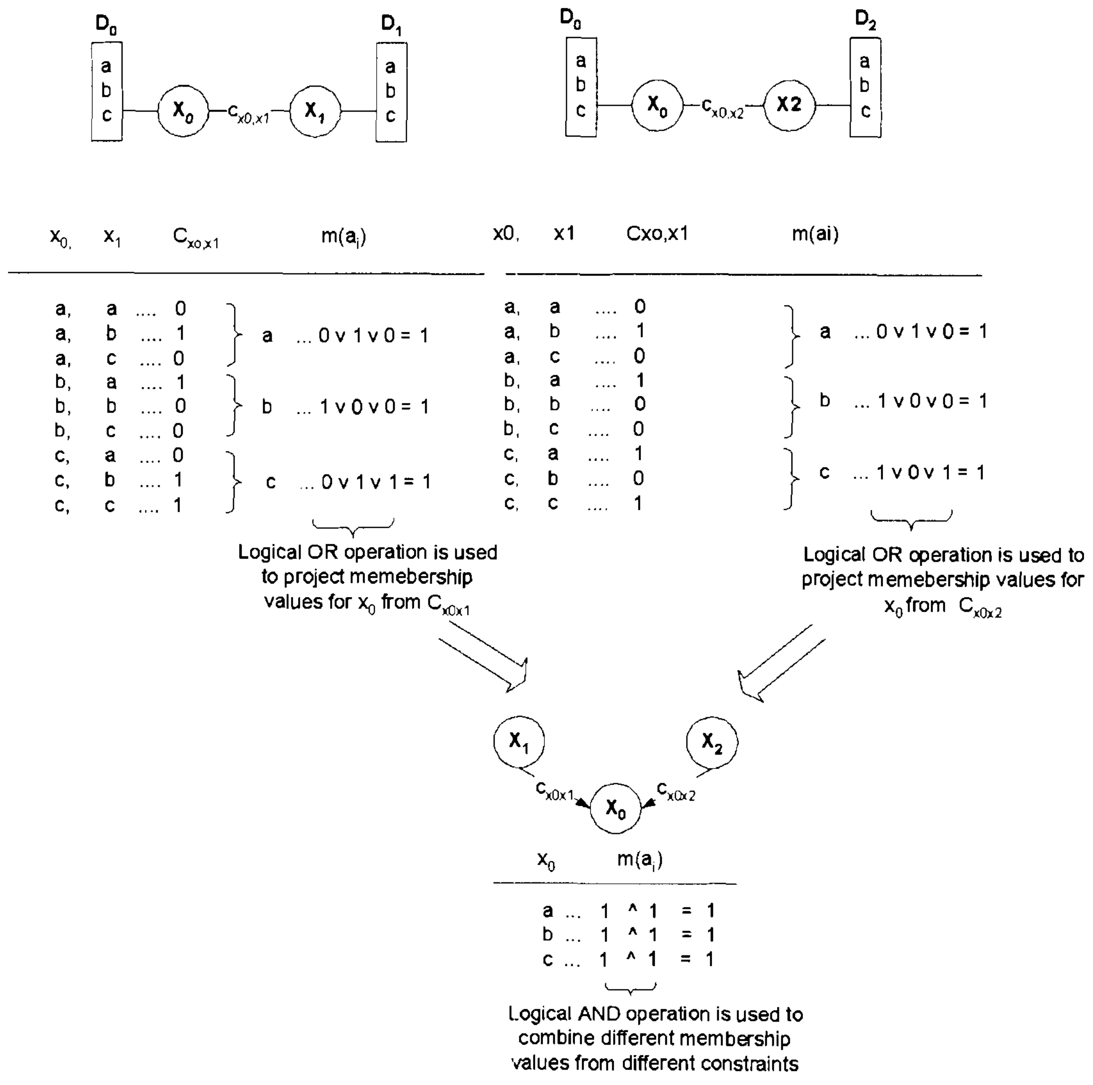
2.3.2 Using Heuristics During Search

There are two major decision points in the chronological backtrack search algorithm: [23]

1. choose which variable to instantiate next?
2. choose which value to assign to?

The ordering in which variables are labeled and values are chosen can affect the efficiency of the search algorithm significantly. Especially when the search algorithm is combined with consistency enforcing techniques. This is because different assignments may cause different amounts of search space to be pruned off.

A popular variable ordering heuristic is to choose the variable which is most likely to cause backtracks, or the *first-fail* principle [8]. The reason behind this heuristic is that if the current partial solution can not lead to a solution, the earlier we stop searching in this direction and backtrack the better. When combined with the consistency enforcing technique, the variable that is most likely to fail is identified as the one with the smallest



$x_{0,}$	x_1	$C_{x0,x1}$	$m(a_i)$
a,	a	0	
a,	b	1	a ... 0 v 1 v 0 = 1
a,	c	0	
b,	a	1	
b,	b	0	b ... 1 v 0 v 0 = 1
b,	c	0	
c,	a	0	
c,	b	1	c ... 0 v 1 v 1 = 1
c,	c	1	

Logical OR operation is used to project memebership values for x_0 from C_{x0x1}

x0,	x1	Cxo,x1	m(ai)
a,	a	0	
a,	b	1	a ... 0 v 1 v 0 = 1
a,	c	0	
b,	a	1	
b,	b	0	b ... 1 v 0 v 0 = 1
b,	c	0	
c,	a	1	
c,	b	0	c ... 1 v 0 v 1 = 1
c,	c	1	

Logical OR operation is used to project memebership values for x_0 from C_{x0x2}

x_0	$m(a_i)$
a ...	1 ^ 1 = 1
b ...	1 ^ 1 = 1
c ...	1 ^ 1 = 1

Logical AND operation is used to combine different membership values from different constraints

Figure 2.11: Boolean operations in the arc consistency algorithm

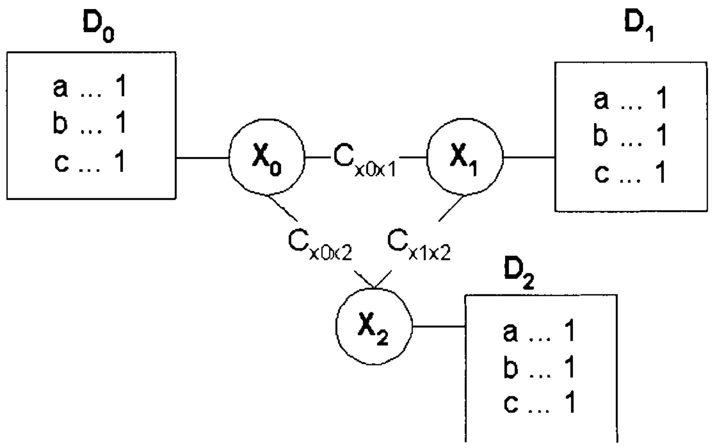


Figure 2.12: Membership values after arc consistency

live domain. For variables with the same number of elements in their live domains, the variable participated in the most constraints is considered most likely to fail.

The value ordering heuristic is a little more complicated. If the objective of the search algorithm is to find all solutions, then all possible combinations of domain values need to be examined. The ordering by which domain values are chosen has no impact on the efficiency of the search algorithm. However, in most cases, we are more interested in finding a solution quickly than the completeness of the algorithm. If this is the case, we can choose the value which is most likely to lead to a solution. However, how to identify such value is not an easy task. One suggestion is to choose the value that appears in the most solutions. However, this information is only available after all solutions are found. Although finding the exact marginal solution counts without a complete search is theoretically impossible, several algorithms have been developed to approximate the marginal solution counts.

In the next Chapter, we introduce some previous algorithms that estimate the marginal solution counts. In Chapter 4 we present our proposed algorithms to approximate marginal solution counts.

Chapter 3

Related Work

3.1 Counting the marginal solution counts

3.1.1 Single Spanning Tree Algorithm

Various research has been done on generating approximations of the marginal solution counts. For tree-structured CSPs, Meisels *et al.* [16], Pearl [21], and Neopolitan [19] have shown that the exact marginal solution counts can be found in polynomial time. For general structured CSPs, Detcher and Pearl [5] suggested to first reduce the complete constraint network to a single spanning tree of the tightest constraints and use the exact marginal solution counts obtained in the simplified problem as an approximation to the original problem.

Let x_j be the next variable to instantiate. To estimate the number of consistent solutions for each domain value of x_j, a single spanning tree rooted at x_j is formed. Consider a tree rooted at x_j as in Figure 3.1. X_s is a set of child nodes of x_j. The algorithm starts at leaves and progresses toward the root.

The marginal solution counts for leaf nodes are initialized to 1's. The number of solutions with variable x_j assigned to value v_j is calculated recursively as:

$$N(x_j = v_j) = \prod_S \sum_T N(x_s = v_s)$$

where

- $N(x_j = v_j)$ is the number of solutions with the variable x_j assigned to the value v_j.
- $S = \{s | x_s \text{ is the child node of } x_j\}$

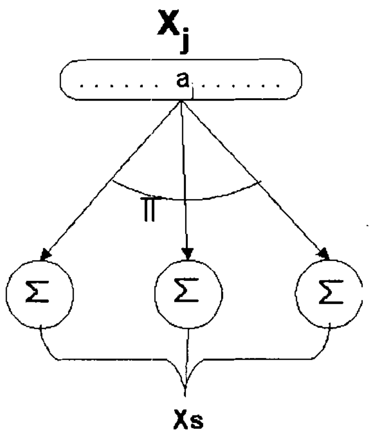


Figure 3.1: Computing the number of solutions for value a_j in variable x_j.

- $T = \{v_s \in D_{x_s} | C_{x_j,x_s}(v_j, v_s)\}$,

In the above formula, $C_{x_j,x_s}(v_j, v_s)$ is a function which returns true if the tuple ($< x_j, v_j >< x_s, v_s >$) is allowed, and false otherwise.

This algorithm is embedded within the directional arc consistency algorithm(DAC): a value that gets the count of 0 means that there is no support from its child nodes, thus can be eliminated. The algorithm stops when all values in the root variable are assigned counts.

One disadvantage of this algorithm is that when constraints of the original CSP are equally tight, the simplified tree-structure CSP may not correspond to the original CSP, thus affect the accuracy of the estimated marginal solution counts.[24].

3.1.2 Multiple Spanning Tree Algorithm

While Dechter and Pearl's algorithm reduces the original problem to Single Spanning Tree (SST), Vernooy and Havens [24] converted the original problem to multiple spanning trees. Approximations to the solution probabilities are obtained by combining the solution probabilities of each subproblems.

Let the original constraint graph be G, and the original constraint graph is decomposed into a set of subproblems C_i, such that

$$G = C_1 \cup C_2 \cup ... \cup C_N$$

where

$$C_1 \cap C_2 \cap ... \cap C_N = \phi$$

The approximated probability that $x = v$ is part of the global solution is calculated as:

$$P(x = v) = P_{C_1}(x = v) \cdot ... \cdot P_{C_N}(x = v)$$

One advantage of the MST algorithm is that all constraints from the original problem are maintained. However, the MST algorithm assumes independence among subproblems which is clearly not the case, because a global solution to C must also be a solution in each C_i. Experimental results show that the independence assumption starts to affect the performance of the MST algorithm as CSPs become more dense and less tree-like. [24].

3.1.3 Uniform Propagation Algorithm

Meisels *et al.* [16] calculated the marginal solution counts based on the probability updating in a Bayesian network.

The constraint network of a CSP is first converted to a directed acyclic graph (DAG) where an edge (x_j, x_i) between variables x_j and x_i is defined when $i < j$. x_j is called the predecessor of x_i and x_1 is always the *sink node* which has no successors.

An example of converting constraint graph with three variables to a directed acyclic graph is shown in Figure 3.2.

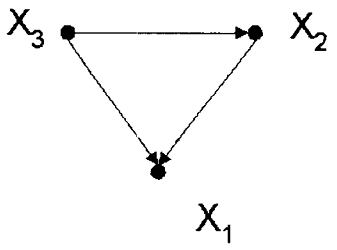


Figure 3.2: Directed acyclic graph for a three variable CSP

Starting at the source node (node with no predecessors) and moving down the ordered graph, the probability of a variable instantiation $< x_j, v_j >$ being part of the global solution, $P(x_j = v_j)$, is calculated recursively as

$$P(x_j = v_j) = \frac{1}{|D_{x_j}|} \prod_S \sum_T P(x_s = v_s)$$

where

- $S = \{s | x_s \text{ is the predecessor of } x_j\}$
- $T = \{v_s \in D_{x_s} | C_{x_j,x_s}(v_j, v_s)\}$

This algorithm assumes conditional independence between predecessors of node x_j. The ordering heuristic of this algorithm is found to perform slightly better than the Dechter and Pearl's SST algorithm [24]. Due to the same reason as the MST method, the independence assumption introduces error into the estimated probabilities.

3.1.4 Geelen's Value Ordering Heuristic [6]

Geelen's algorithm calculates products of the remaining domain sizes of the future variables. This value represents an upper bound on the number of different solutions after assigning $x_i = v_i$. The domain value with the highest upper bound value is chosen.

Given a set S of current compound label, CONSISTENT(S) is defined as *true* if and only if these assignments do not violate any constraints. Let x_i be a future variable that is as yet unassigned. The set of useful values for x_i is defined as

$$DOM_S(x_i) = \{v_i | v_i \in D_{x_i} \wedge CONSISTENT(S \cup \{x_i = v_i\})\}$$

The number of values that will still be useful for a future variable x_j after assigning v_i to x_i is defined as

$$LEFT_S(x_j | x_i = v_i) = |DOM_{S \cup \{x_i = v_i\}}(x_j)|$$

And finally, the *promise* value of an assignment is defined below. The domain value with the largest *promise* value is chosen.

$$promise_S(x_i = v_i) = \prod_{j \neq i} LEFT_S(x_j | x_i = v_i)$$

Geelen's algorithm constantly assure arc-consistency. If $LEFT_S(x_j | x_i = v_i) = 0$, then the assignment $x_i = v_i$ would leave x_j without useful values. So, v_i can be removed from the live domain of x_i. This domain reduction is propagated through the constraint network. All the related *promise* values are re-calculated.

3.1.5 Probabilistic Arc Consistency (pAC) Algorithm

The pAC algorithm introduced by Horsch and Havens [13] is a generalization of the arc consistency algorithm. It tries to determine how often each domain value appears in all solutions of a CSP.

Given a constraint graph, as shown in Figure 3.3, in which each node represents a variable of the CSP, the pAC algorithm works in a distributed fashion. Each node has a list of neighbors which are connected with it by constraints. The pAC process begins when each node tells all neighbors its current knowledge of the solution probabilities of its own domain values. After node i receives a message P_{ij} from node j, node i consults the constraint between node j, C_{ij}, and updates the solution probabilities for each of its domain values using the real plus operation, $+$. Messages from different nodes are combined using real multiplication, $\times$. After combining messages from all its neighbors, node i sends a new message to each of its neighbors and tells them its updated probabilities. The process repeats in this fashion until changes in the solution probabilities are less than a threshold, ϵ, or the total number of iterations has exceeded the predefined maximum number. To avoid the double counting problem, *i.e.,* same information being considered more than once, the updated message P'_{ji} sent from node i back to node j should not include the information that node j already knew.

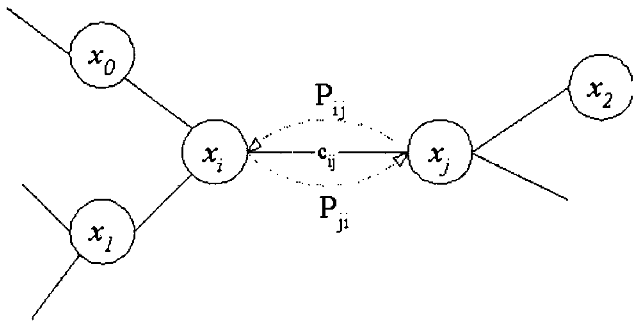


Figure 3.3: pAC algorithm. Message P_{ij} is received by node x_i and message P_{ji} is received by node x_j.

Using mathematical form, the probabilistic arc consistency is achieved by updating the following formula repeatedly until the probabilities in $P'_{ji}(a_i)$ becomes stable.

$$\forall a_i \in D_i, P'_{ji}(a_i) = \alpha \prod_{x_j \in N(x_i)} \sum_{a_j \in D_j} C_{ij}(a_i, a_j) \frac{P_{ij}(a_j)}{P_{ji}(a_i)} \tag{3.1}$$

in this formula:

- $P'_{ji}(a_i)$ is the revised probability that $x_i = a_i$.
- $\prod$ represents real product.
- $\sum$ represents real sum.
- $C_{ij}(a_i, a_j)$ is 1 if the pair $x_i = a_i, x_j = a_j$ is allowed by the constraint between x_i and x_j, and 0 otherwise.
- α normalizes the resulting probabilities such that they sum to 1.

In the above formula, the revised probability that $x_i = a_i$ in the updated message sent from node i back to node j, $P'_{ji}(a_i)$, is calculated as follows: for each neighbor j, the solution probabilities for value a_j of node j that are consistent with value a_i are added up using real sum operation, $\sum$. The solution probability from different neighbors are combined using real product operation, $\prod$. To avoid double counting problem, the probabilities in the previous message which are sent from node j to node i, $P_{ji}(a_i)$, is deleted from the updated message P'_{ji} sent from node i back to node j. For more detailed information, see [14].

Experimental results [12] show that there is a high correlation between the exact solution probabilities and the approximations computed by pAC. And when this information is used as a value ordering heuristic to guide the search algorithm, the number of backtracks required to find solutions is reduced by as many as two orders of magnitude. However the pAC algorithm exhibits slow convergence speed and in some cases may not converge at all. For larger problems, pAC's runtime is competitive or superior to other marginal solution counting algorithms. When the problem is small, the cost of computing the pAC outweighs the benefit of reducing search cost.

In Chapter 4, we introduce our proposed algorithms, HAC and μAC . These algorithms provide heuristic ordering for domain values while maintaining arc consistency throughout the whole process. In addition, our proposed algorithms intend to overcome the convergence problem of the pAC algorithm.

Chapter 4

HAC and μAC Algorithms

In this Chapter, we introduce our proposed algorithms, Histogram Arc Consistency(HAC) and μ Arc Consistency (μAC). These algorithms generate rough approximations of marginal solution counts. This information can be used as value ordering heuristic to guide the search algorithm.

4.1 Histogram Arc Consistency (HAC)

The Histogram Arc Consistency algorithm is a natural extension of both the traditional arc consistency algorithm and the probabilistic arc consistency algorithm. In the traditional arc consistency algorithm, each domain value is associated with a boolean value representing the membership status of this value which tells us whether the associated domain value is in the live domain or not. Constraints are combined using boolean operations, $\vee$ and $\wedge$. As pointed out in the previous section, the constraint value obtained by the traditional arc consistency algorithm distinguishes domain values that are in the live domain from those that are not, it does not give these values any likelihood of being in a consistent global solution. In the HAC algorithm, we change the data type of the constraint valuations from boolean to integer, and we change the operations from boolean operations to integer operations, *i.e.* $+$ and $\times$. Using mathematical form the Histogram Arc Consistency algorithm is represented as:

$$\forall v_i \in D_{x_i}, f(v_i) = \prod_{x_j \in N(x_i)} \sum_{v_j \in D_{x_j}} C_{x_i,x_j}(v_i, v_j) m(v_j)$$

where

- D_{x_i} is the current live domain of variable x_i.
- $f(v_i)$ is the HAC valuation for value v_i in x_i.
- $\prod$ represents integer product.
- $\sum$ represents integer sum.
- $N(x_i)$ represents neighboring variables of x_i.
- $C_{x_i,x_j}(v_i, v_j)$ is 1 if the pair $\{x_i = v_i, x_j = v_j\}$ is allowed by the constraint C_{x_i,x_j}, and 0 otherwise.
- $m(v_j)$ is the membership of v_j in the live domain of x_j. It is obtained by the following formula

$$m(v_j) = \begin{cases} 1 & f(v_j) > 0 \\ 0 & f(v_j) = 0 \end{cases}$$

In the above function, $f(v_i)$ is the HAC valuation associated with the assignment $< x_i, v_i >$. The sum operation tells us how many domain values in variable x_i's neighboring variable, x_j, supporting the assignment $< x_i, v_i >$. The number of supports from different neighbors are combined using the integer product operation, $\prod$. A value that gets the support count of 0 means that there is no support from its neighboring variables, and thus can be removed from the variable's live domain. Same as in the traditional arc consistency algorithm, removing domain values may cause other values in their neighboring variables to become inconsistent, thus should be deleted as well. To achieve Histogram Arc Consistency for the whole problem, these HAC valuations are updated repeatedly until no values are deleted from any variable domains.

The HAC-REVISE(x_i, x_j) procedure shown in Figure 4.1 revises HAC valuations for variable x_i based on the live domain of variable x_j. In this procedure, m_i is a vector containing the membership of domain values of variable x_i, as defined above. s_{ij} is a vector containing the number of supports for domain values in variable x_i from x_j. Initially, all HAC values are set to 1. At lines 2, and 3, previous memberships of domain values for variable x_i, and previous supports for domain values of x_i from neighboring variable x_j are saved into vector $\hat{m_i}$ and $\hat{s_{ij}}$ respectively. For each value v_i in the live domain of variable x_i, line 8 counts the new number of supports based on the current live domain of its neighboring variable x_j. Line 10 combines the supports for value v_i from different neighboring variables

```
precedure HAC-REVISE($x_i, x_j$)
1.  begin
2.      $\hat{m_i} \leftarrow m_i$
3.      $\hat{s_{ij}} \leftarrow s_{ij}$
4.      $s_{ij} \leftarrow \mathbf{0}$
5.      for $v_i \leftarrow$ each element in $D_{x_i}$ do
6.          for $v_j \leftarrow$ each element in $D_{x_j}$ do
7.              if $(C_{x_i,x_j}(v_i, v_j) \wedge m_j[v_j])$ then
8.                  $s_{ij}[v_i]$ ++;
9.      for each element $k$ in $f_i$ do
10.         $f_i[k] = f_i[k] * s_{ij}[k] / \hat{s_{ij}}[k]$
11.     $m_i \leftarrow$ UPDATEMEMBERSHIP($f_i$)
12.     return $(m_i \neq \hat{m_i})$
13. end

procedure UPDATEMEMBERSHIP($f_i$)
1.  begin
2.      $m_i \leftarrow \mathbf{0}$
3.      for each element k in $f_i$ do
4.          if $f_i[k] > 0$ then
5.              $m_i$[k] = 1
6.      return $m_i$
7.  end
```

Figure 4.1: The HAC-REVISE algorithm

using multiplication. To avoid the double counting problem, the old number of supports from variable x_j, $\hat{s_{ij}}$, is deleted from the updated support valuation.

The membership value for each domain value is updated base on the new HAC valuations as shown in the UPDATEMEMBERSHIP procedure. If any of the membership values becomes zero, the associated domain value is deleted from the variable's live domain. The deletion of domain values from live domains will certainly cause HAC valuations of neighboring variables to change. As in the traditional arc consistency algorithm, this deletion is propagated through the constraint network.

Figure 4.2 shows the pseudo code of the HAC algorithm which achieves histogram arc consistency for the whole CSP. The HAC algorithm is based on the AC-3 algorithm. First, the HAC-REVISE procedure is performed for each constraint C_{x_i,x_j}. If any value is deleted from the variable domain during the HAC-REVISE procedure, all affected constraints are

```
procedure HAC(Z, D, C)
1. begin
2.     $Q \leftarrow \{(i,j)|C_{x_i,x_j} \in C, i \neq j\}$
3.     while Q not empty do
4.         select and delete any constraint $C_{x_k,x_m}$ from $Q$
5.         if HAC-REVISE($x_k, x_m$) then
6.             $Q \leftarrow Q \cup \{(i,k)|C_{x_i,x_j} \in C, i \neq k, i \neq m\}$
7.     endwhile
8. end
```

Figure 4.2: The HAC algorithm

re-examined and valuations revised. The process terminates when no value is deleted from any variable domains. The result is a set of histogram valuations f_i for variable x_i that tell us how well each domain value in D_{x_i} is locally consistent with its neighboring variables. We interpret f_i as a rough approximation of the marginal solution counts. This information can now be used as value ordering heuristic to guide the search.

Although the HAC algorithm is essentially arc consistency, by changing the data type of the constraint valuations from boolean to integer, and changing additive and multiplicative operations from boolean operations to integer operations, we are able to gather more information from the HAC propagation process than from the traditional arc consistency algorithm. Like the traditional arc consistency algorithm, when combined with the backtrack search algorithm to solve CSPs, the HAC algorithm can be used in preprocessing before search to reduce the size of the search tree or used during search (after each assignment is made) to prune off search space. In addition, the HAC algorithm provides heuristic order for the domain values: choose a value in the variable's live domain with the largest HAC valuation.

Still using the CSP example as shown in Figure 3.5 and 3.6, Figure 4.3 shows the process of revising HAC valuations for variable x_0. The integer sum operation is first used to project the constraint values for the variable x_0 from constraints $C_{x_0x_1}$ and $C_{x_0x_2}$ respectively. Then, the integer product operator is used to combine these constraint values from different constraints. To achieve histogram arc consistency for the whole problem, these projection and combination operations need to be performed on every constraint repeatedly until no domain is changed. The CSP that is histogram arc consistent is shown in Figure 4.4.

¿From these HAC valuations we know that all domain values are still in their variable

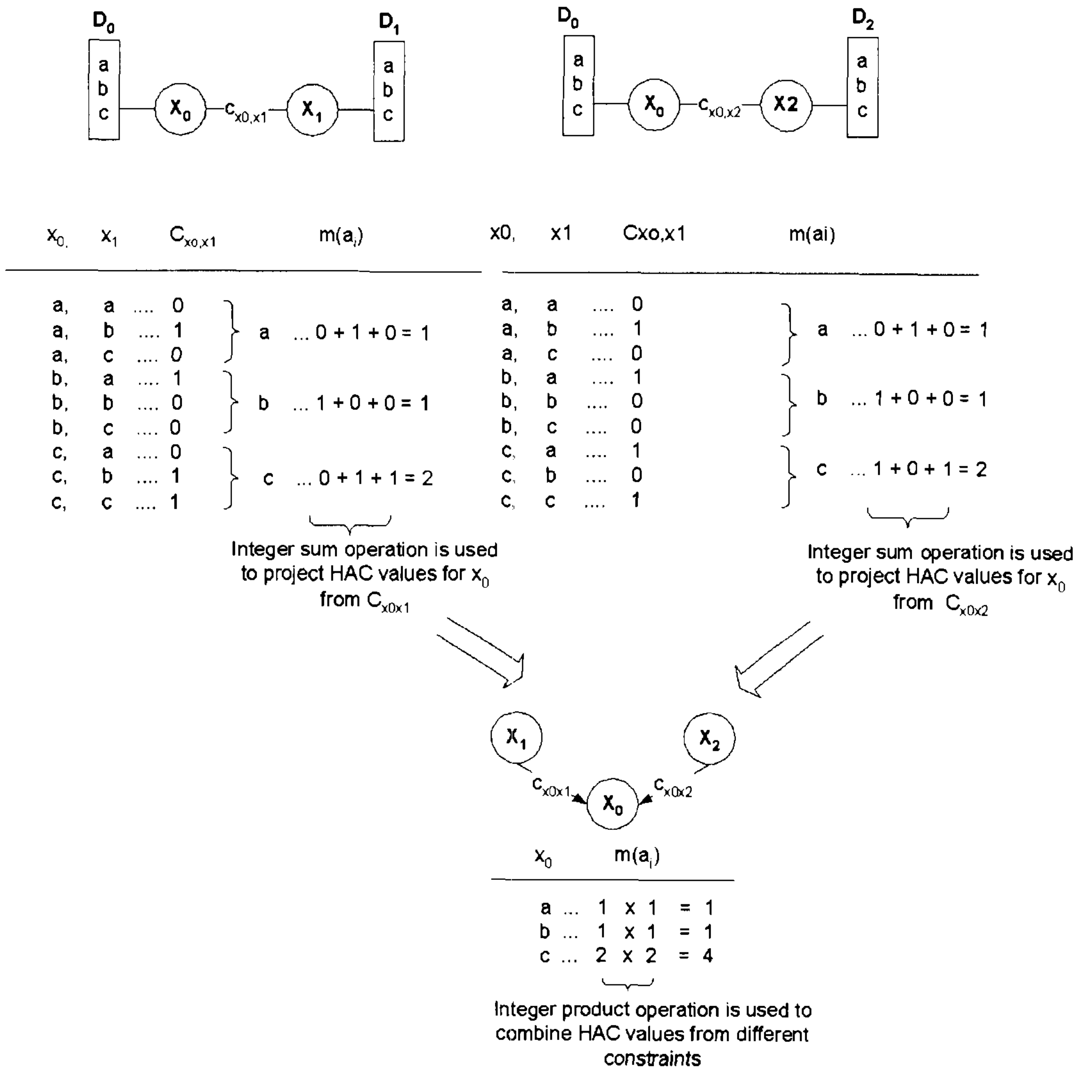


Figure 4.3: Integer operations in the HAC algorithm

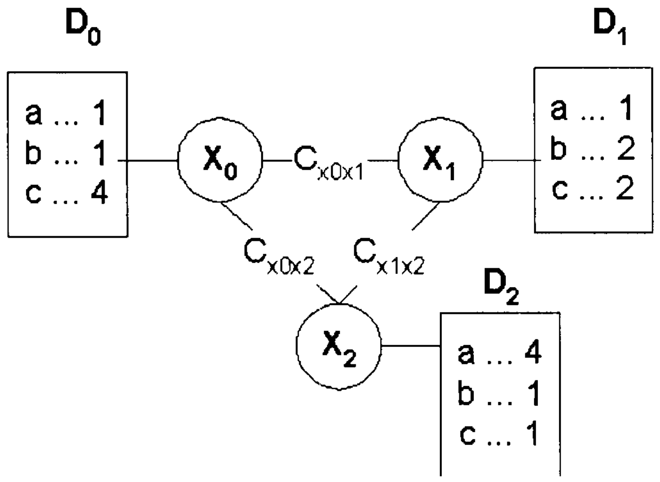


Figure 4.4: Constraint values after achieving HAC

live domains. In addition, these HAC valuations tell us that, for variable x_0, the value c may be the better choice than values a and b. And for variable x_1, values b and c are more locally consistent than the value a. And the best value for variable x_2 is the value c. Based on these HAC valuations, the search algorithm assigns $x_1 = c$ and finds a solution without any backtracks. From this example we can see that HAC provides a good value ordering heuristic. When this information is used to guide the search algorithm, backtracks are reduced.

4.2 μ Arc Consistency (μAC)

In most cases, we are more interested in finding a solution quickly than in the completeness of the algorithm. So the idea behind the μAC algorithm is: We aggressively reduce variable live domains by simply removing domain values with small HAC valuations from variable live domains. We quickly search through this very small search space. If no solution is found, we expand the search space a little more by putting more values back into live domains and search again. If eventually we have to put all values back into variable live domains and search, the μAC algorithm exhausts the entire search space. Thus μAC is guaranteed to find the solution if one exists. The worst case complexity of the μAC algorithm is $O(n + 2^n + 3^n + ... + d^n) = O(d^n)$. However, as pointed out in [9], if the heuristic is good, i.e., if HAC algorithm provides a good ordering of domain values, the worst scenario is

```
procedure $\mu$AC(Z, D, C, $\mu$)
1. begin
2.     HAC(Z, D, C)
3.     $D' \leftarrow \mu$-OPERATION(D, $\mu$)
4.     HAC(Z, $D'$, C)
5. end

procedure $\mu$-OPERATION(D, $\mu$)
1. begin
2.     for each domain $D_i \in D$ do
3.         $D'_i \leftarrow$
4.             delete all the elements with valuations less than $\mu$ from $D_i$
5.     return $D'$
6. end
```

Figure 4.5: μAC algorithm

unlikely to happen.

Basically, μAC is HAC with a μ parameter. In HAC, when any of the HAC valuations becomes zero the associated domain value is deleted from the variable's live domain and all affected constraints are reexamined. In μAC, 0 is replaced by μ. So if any HAC valuation becomes less than μ, the associated domain value is deleted from the variable's live domain and related constraints are reexamined.

One problem with this algorithm is that one can never know how big these HAC values will be after HAC propagation. For ease of implementation, we set the μ value to be proportional to the size of the variable's live domain.

The pseudo code of μAC algorithm is presented in Figure 4.5. The algorithm starts with performing an HAC propagation at line 2. After histogram arc consistency is achieved, the μ-OPERATION is called at line 3 with a predefined value μ as one of the arguments. This operation temporarily removes domain values with HAC valuations less than μ from variable live domains. And then, at line 4, another HAC propagation is performed to achieve μ arc consistency for the CSP.

4.3 Using HAC and μAC to Solve CSPs

4.3.1 The HAC Look-ahead Algorithm

We call the algorithm that combines HAC with the chronological backtrack search algorithm the HAC Look-ahead Algorithm. The pseudo code of the HAC Look-ahead algorithm is shown in Figure 4.6.

The HAC Look-ahead algorithm searches for solutions through recursively calling the HAC-L(U, P, D, C) procedure. In the HAC-L procedure, line 3 performs HAC propagation. If none of the variable domains becomes empty after HAC, Line 8 assigns the variable x to a value with the largest HAC valuation in its domain, say v. After this assignment, the live domain of variable x is updated using the *UPDATE* procedure. In this procedure, all the constraint values in the domain of variable x are set to 0 except for the value v which is set to 1. This updated domain D' is used in the recursive call of the HAC-L procedure at line 12 to instantiate another variable. If after the HAC propagation, any of the variable domains becomes empty, the program backtracks, which means that the last assigned variable is re-assigned to a value with the next highest HAC valuation in its domain. The program either stops after all variables are assigned a value or exhausts the whole search space and concludes that no solution exists.

4.3.2 The μAC Look-ahead Algorithm

While HAC Look-ahead is a complete algorithm, the μAC Look-ahead algorithm only looks for the first solution.

The pseudo code of μAC Look-ahead algorithm is shown in Figure 4.7. The $init(\mu)$ procedure initializes the μ value to be 90%. The μAC Look-ahead algorithm searches for solutions through recursively calling the μAC-L procedure. In the μAC-L procedure, line 3 performs μAC propagation with the μ parameter set to 90%. This means that, after HAC propagation, 90% of the domain values are temporarily removed from variable live domains and only top 10% of the domain values with the largest HAC valuations stay. If none of the variable domains becomes empty after μAC, Line 8 assigns variable x to a value with the largest HAC valuation in its domain, say v. After this assignment, the live domain of variable x is updated using the *UPDATE* procedure. In this procedure, all the constraint values in the domain of variable x are set to 0 except for the value v which is set to 1.

```
precedure HAC-Lookahead(k, D, C)
1.  begin
2.      HAC-L(Z, {}, D, C)
3.  end

precedure HAC-L(U, P, D, C)
1.  begin
2.      if (U = {}) then return (P)
3.      HAC(Z, D, C)
4.      if no domain in D becomes empty then
5.          begin
6.              pick one variable $x$ from U
7.              repeat
8.                  pick a value $v$ from $D_x$ with the largest HAC valuation;
9.                  delete $v$ from $D_x$
10                  $D' \leftarrow$ UPDATE(D, $< x, v >$)
11.                  Result $\leftarrow$
12.                      HAC-L($U - \{x\}, P + \{< x, v >\}, D', C$)
13.                  if(Result $\neq$ NIL) then return (Result)
14.               until $D_x = \{\}$
15.           end
16.       return NIL
17.  end

precedure UPDATE($D, < x, v >$)
1.  begin
2.      $D' \leftarrow D$
3.      for each element $a$ in $D'_x$
4.          if $a = v$
5.              f(a) = 1
6.          else f(a) = 0
7.      return $D'$
8.  end
```

Figure 4.6: The pseudo code of the HAC Look-ahead algorithm

This updated domain D' is used in the recursive call of the μAC-L procedure at line 12 to instantiate another variable. If after the μAC propagation, any of the variable domains becomes empty, the program backtracks, which means that the last assigned variable is re-assigned to a value with the next highest HAC valuation in its domain. The program either stops after all variables are assigned values or concludes that no solution exists with the given μ value. If after the μAC-L procedure returns to the μAC-Lookahead(Z, D, C) procedure at line 4 and no solution is found, original variable domains are restored and the μ value is decreased by a pre-defined amount $\triangle$, as shown at line 7 of the μAC-Lookahead(Z, D, C) procedure. This means that, in the next iteration, more values are allowed to stay in their variable live domains. The μAC-L procedure is performed repeatedly decreasing μ each time until a solution is found or until μ reaches zero thus concludes that no solution exists.

In the next Chapter, the proposed algorithms are tested using randomly generated CSPs. Numbers of backtracks used to find first solutions and CPU times are collected. The performances of these two new algorithms are compared with the traditional arc consistency algorithm and with the pAC algorithm.

```
precedure μAC-Lookahead(Z, D, C)
1.  begin
2.      init($\mu$)
3.      repeat
4.          Result ← $\mu$AC-L(Z, {}, D, C, $\mu$)
5.          if Result $\neq$ NIL then
6.              return Result
7.          $\mu = \mu - \triangle$
8.      until $\mu = 0$
9.      return NIL
10. end

precedure μAC-L(U, P, D, C, μ)
1.  begin
2.      if (U = {}) then return (P)
3.      $\mu$AC(U, D, C, $\mu$)
4.      if no domain in D becomes empty then
5.          begin
6.              pick one variable $x$ from U
7.              repeat
8.                  pick a value $v$ from $D_x$ with the largest non-zero HAC valuation
9.                  delete $v$ from $D_x$
10.                 $D' \leftarrow$ UPDATE(D, $< x, v >$)
11.                 Result ←
12.                     $\mu$AC-L$(U - \{x\}, P + \{< x, v >\}, D', C)$
13.                 if(Result $\neq$ NIL) then return (Result)
14.             until $D_x = \{\}$
15.         end
16.     return NIL
17. end

precedure UPDATE($D, < x, v >$)
1.  begin
2.      $D' \leftarrow D$
3.      for each element $a$ in $D'_x$
4.          if $a = v$
5.              f(a) = 1
6.          else f(a) = 0
7.      return $D'$
8.  end
```

Figure 4.7: The μAC Look-ahead algorithm

Chapter 5

Experimental Results

In this chapter, proposed algorithms are tested using randomly generated CSPs. Before we present experimental results, we will first introduce the algorithm used to generate random csps.

5.1 Generating Random CSPs

Randomly generated CSPs are widely used to test and compare algorithms. These random CSPs can be described by the tuple $< n, m, p_1, p_2 >$, where n represents the number of variables, m represents the uniform domain size, p_1 is a measure of the density of the constraint graph, and p_2 is a measure of the tightness of the constraints [7].

A random CSP generator generates a constraint graph G, in which each edge represents the constraint between a pair of variables. For each edge in G, a boolean matrix is generated by randomly choosing pairs of incompatible values. One simple way to generate random CSPs is: first, randomly select exactly $p_1 n(n-1)/2$ edges for G, and then for each selected edge randomly pick exactly $p_2 m^2$ pairs of values as incompatible. Achlioptas *et al.* identify a shortcoming of this model. They prove that if $p_2 \geq 1/m$ then, as $n \rightarrow \infty$, there almost always exists a flawed variable, for which every value is flawed. A value for a variable is flawed if it is not supported by any value in its neighboring variable. So, a CSP with a flawed variable can not have a solution. To overcome this problem, Gent *et al.* [7] introduce the flawless random CSP model. The basic idea of this model is to make sure that every value of a variable is supported by at least one unique value.

The random CSP generator used in this thesis is written by Michael Horsch. He used the flawless model introduce by Gent *et al* [7]. The algorithm is as follows:

1. randomly generate exactly $p_1 n(n-1)/2$ edges for G.
2. for each edge, the boolean matrix is initialized such that the diagonal elements are set to be true and all other elements are set to be false.
3. permute rows randomly.
4. randomly set more elements to be true until p_2 is correct.

5.2 Experimental Results

For small CSP problems, we test a set of random CSPs with numOfVar = 20 and domSize = 10. These test problems are further divided into 3 sets. For each set, the probability that a constraint exists between any pair of variables, represented as p_1, is set to be 0.2, 0.4 and 0.8 respectively. In other words, these three sets of CSPs represent three different levels of density of the constraint network. When p_1 equals to 0.2, the constraint network is sparsely connected and when p_1 equals 0.8, the network is densely connected. Within each constraint, the probability that a given pair of values is disallowed, called p_2, ranges from 0.01 to 1.0 with 0.01 increment. For each pair of p_1 and p_2, 200 different CSP instances are tested. To test performances of HAC and μAC for medium sized CSPs, we test a set of problems with n = 60 and m = 30. For each problem, p_1 is set to be 0.3 and p_2 ranges from 0.1 to 1.0 with 0.01 increment. For each pair of p_1 and p_2, 50 CSP instances are tested.

These test problems are solved using four different algorithms. They are: (1). backtrack search with traditional arc consistency (Full Look-ahead); (2). backtrack search with HAC as a value ordering heuristic (HAC Look-ahead); (3). backtrack search with μAC as a value ordering heuristic (μAC Look-ahead). μ values used in this algorithm are $\{0.8, 0.4, 0.2, 0\}$. (4). backtrack search with pAC as a value ordering heuristic (pAC Look-ahead). For each test problem, we record the number of backtracks required to find first solutions, and also CPU times taken by these algorithms. All tests are done using computer with AMD Athlon processor with 261MB RAM.

Experimental results show that our proposed algorithms provide valuable heuristic information. When HAC and μAC are used as value ordering heuristics to guide the search algorithm, fewer number of backtracks are required to find solutions than the traditional arc consistency algorithm. Our experimental results also show that our methods are quick to compute. As the size of the test problem increases, the advantage of using HAC and μAC as value ordering heuristics becomes more significant.

In Figure 5.1, 6800 problems with numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.2, p_2 (constraint tightness) $\in [0.46, 0.79]$ are selected. The figure plots numbers of backtracks used by each algorithm to find first solutions against constraint tightness (p_2). In Figure 5.2, we compare the total number of backtracks used to solve all test problems by each algorithm. Test results show that our proposed algorithms, HAC and μAC, outperformed the traditional arc consistency algorithm. On average, the HAC algorithm saved about 3.20% of backtracks from the traditional arc consistency algorithm, μAC saved 38.20% and pAC saved 45.52%.

In Figure 5.3, 7000 problems with numVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$ are selected. The figure plots numbers of backtracks used by each algorithm to find first solutions against constraint tightness (p_2). The total numbers of backtracks used to solve all test problems by each algorithm are shown in Figure 5.4. As far as the number of backtracks is concerned, μAC performed almost as well as the pAC algorithm. On average, HAC saved 22.21% of backtracks from the traditional arc consistency algorithm, μAC saved 44.74% and pAC saved 50.15%.

In Figure 5.5, 5200 problems with numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.8, p_2 (constraint tightness) $\in [0.1, 0.8]$ are selected. The figure plots numbers of backtracks used by each algorithm to find first solutions against constraint tightness (p_2). The total numbers of backtracks used to solve all test problems by each algorithm are shown in Figure 5.6. These results show that our proposed algorithms take fewer number of backtracks to find the first solution than the traditional arc consistency algorithm. In addition, the μAC algorithm uses fewer number of backtracks than the pAC algorithm. On average, HAC saved 20.7% of backtracks from the traditional arc consistency algorithm, μAC saved 43.5% and pAC saved 42.4%.

After comparing test results for all three sets of CSPs, we can see that as the density of the constraint network, p_1, increases, the performance of the HAC and the μAC algorithms improves. When $p_1 = 0.2$, the HAC algorithm only saved 3.20% of backtracks from the traditional arc consistency algorithm, and μAC saved 38.20%. When $p_1 = 0.8$, HAC saved 20.7% of backtracks from the traditional arc consistency algorithm, and μAC saved 43.5%. This is because the more constraints that are involved in a CSP, the more informative the

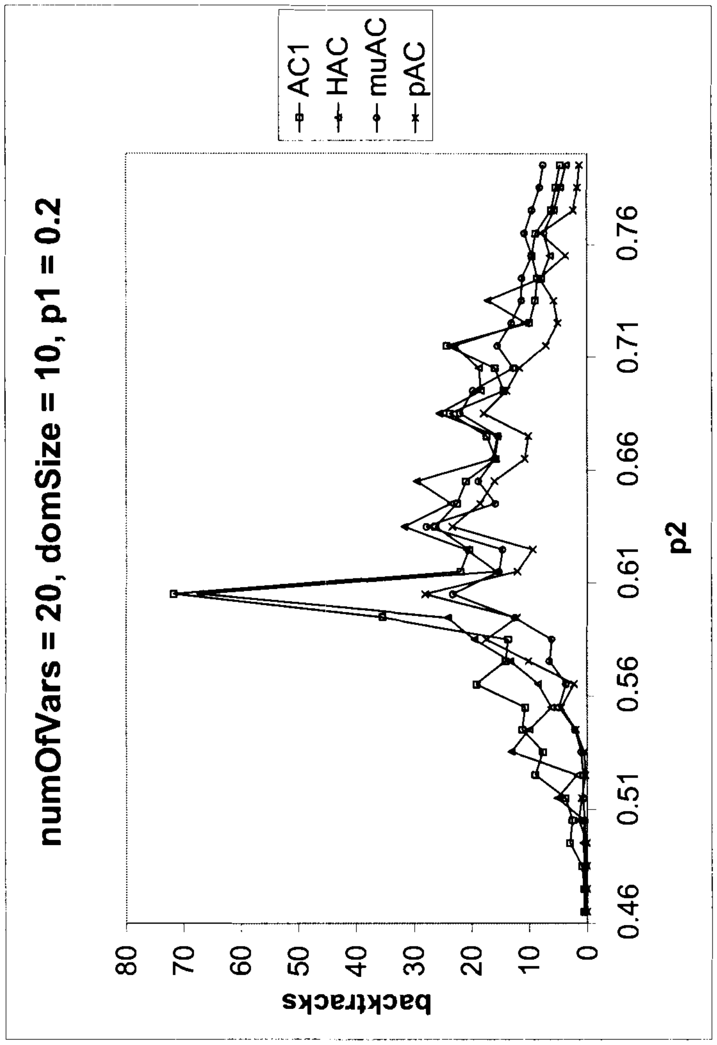


Figure 5.1: Number of backtracks used to find first solutions, numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.2, p_2 (constraint tightness) $\in [0.46, 0.79]$

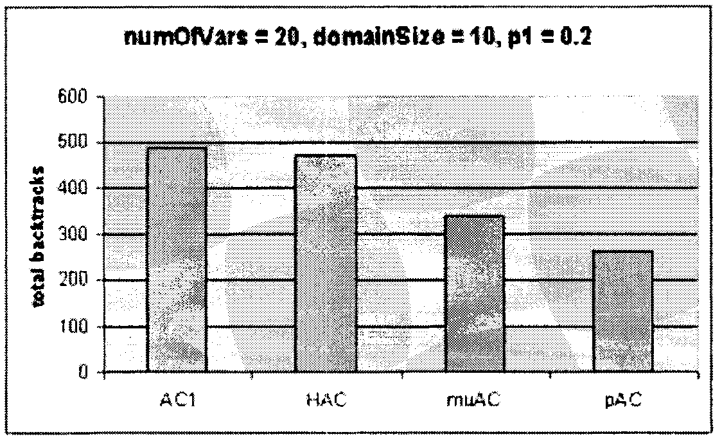


Figure 5.2: Total number of backtracks used to find first solutions, numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.2, p_2 (constraint tightness) $\in [0.46, 0.79]$

valuations will be after the HAC propagation. If most variables in the CSP are not connected by constraints, then after the HAC propagation, most of the valuations are close to 1. In this case, the HAC algorithm provides no more heuristic information than the traditional arc consistency algorithm.

We also compared CPU times taken to find a first solution for problems with numberVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$. As shown in Figure 5.7 and Figure 5.8, the pAC Look-ahead algorithm takes much longer CPU time than HAC Look-ahead and μAC Look-ahead algorithms. Due to the overhead of integer operations and the extra search steps to find the most promising value in a variable domain, HAC Look-ahead and μAC Look-ahead algorithms take longer CPU time than the Full Look-ahead algorithm.

To test the performance of HAC and μAC for larger CSP problems, we tested a set of problems with numOfVar = 60, domSize = 30, p_1 (constraint density) = 0.3 and p_2 (constraint tightness) $\in [0.1, 0.3]$. As shown in Figure 5.9, the pAC algorithm takes a large amount of CPU time to find solutions. HAC Look-ahead and μAC Look-ahead algorithms take less CPU time than the Full Look-ahead algorithm to find the first solutions. The reduced search cost compensates for the overhead incurred in the proposed algorithms.

In the next Chapter, we conclude this thesis and point out possible future works.

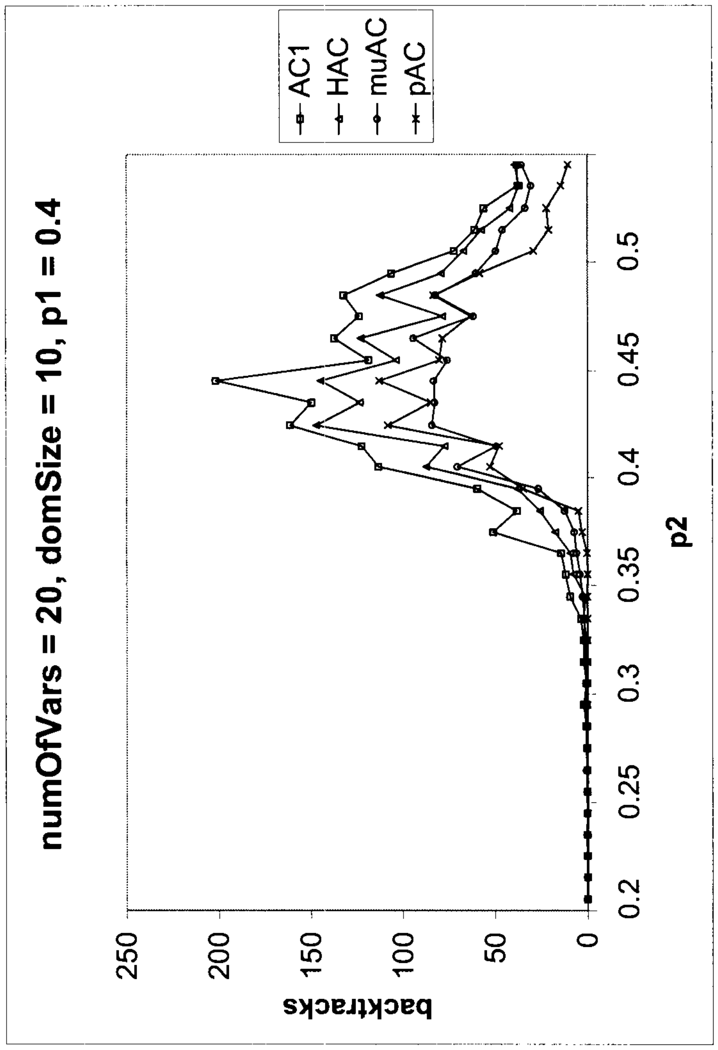


Figure 5.3: Number of backtracks used to find first solutions, numVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$

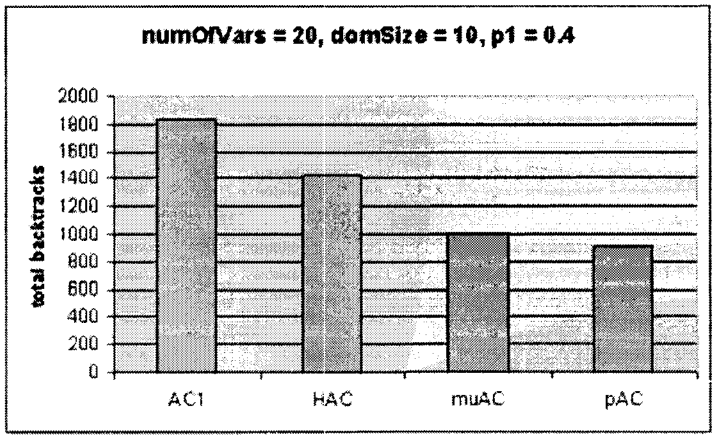


Figure 5.4: Total number of backtracks used to find first solutions, numVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$

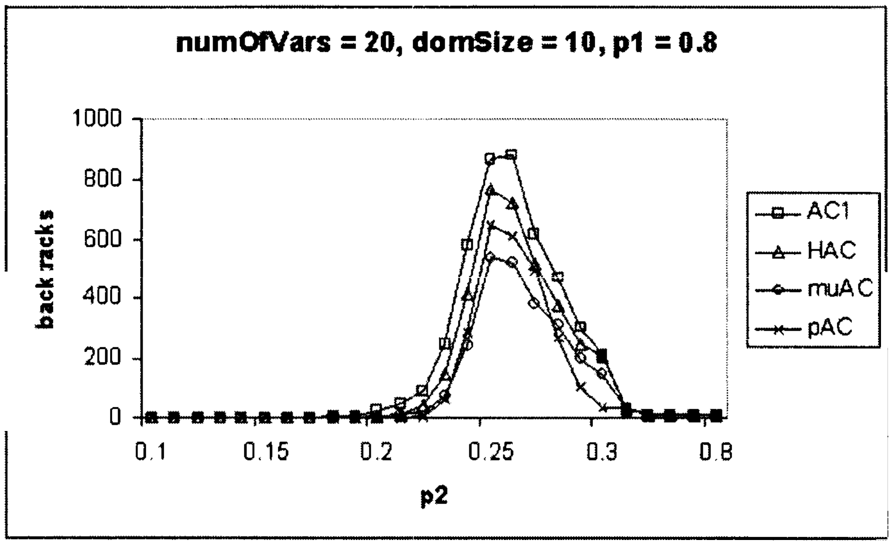


Figure 5.5: Number of backtracks used to find first solutions, numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.8, p_2 (constraint tightness) $\in [0.1, 0.8]$

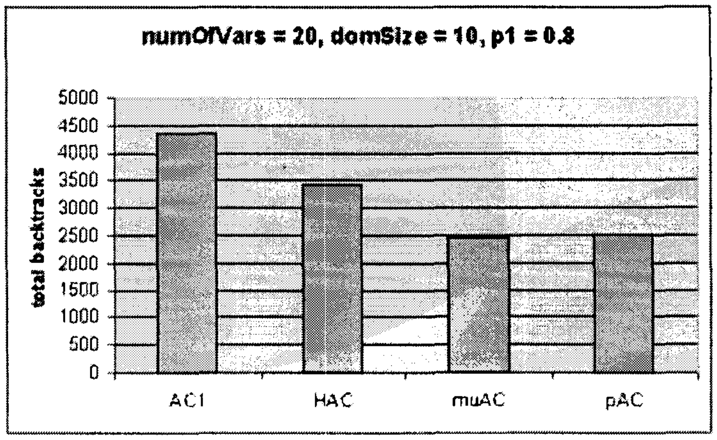


Figure 5.6: Total number of backtracks used to find first solutions, numOfVar = 20, domSize = 10, p_1 (constraint density) = 0.8, p_2 (constraint tightness) $\in [0.1, 0.8]$

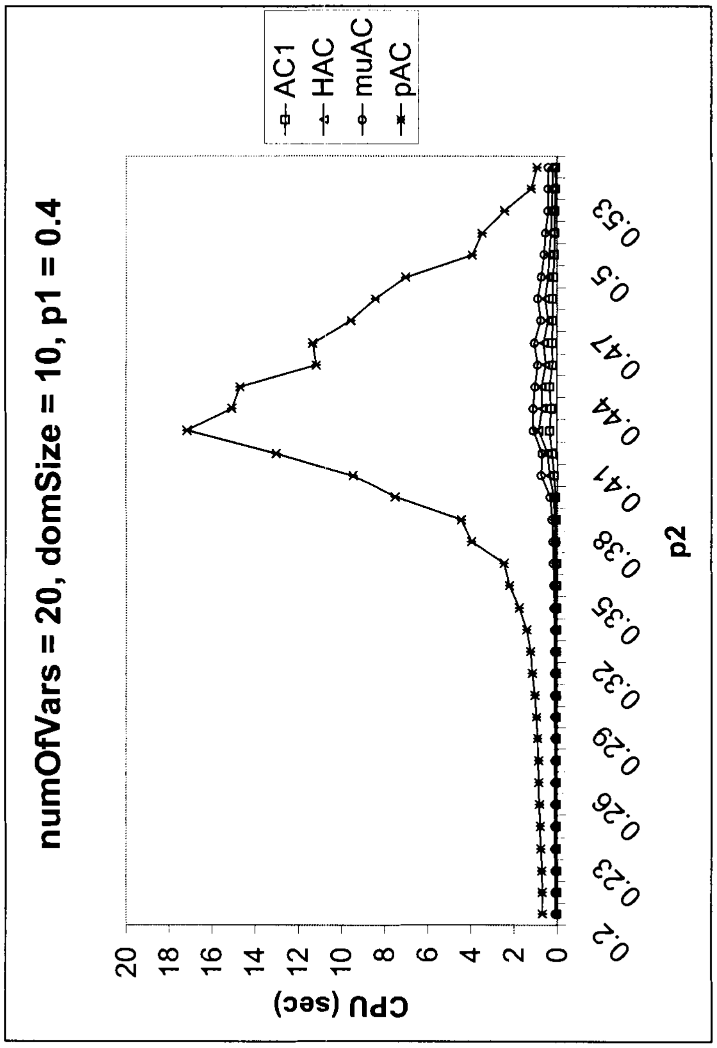


Figure 5.7: CPU taken to find first solutions, numberVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$

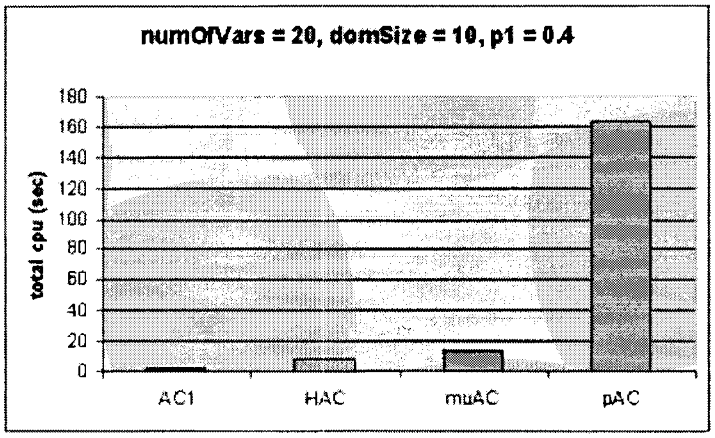


Figure 5.8: Total CPU taken to find first solutions, numberVar = 20, domSize = 10, p_1 (constraint density) = 0.4, p_2 (constraint tightness) $\in [0.2, 0.54]$

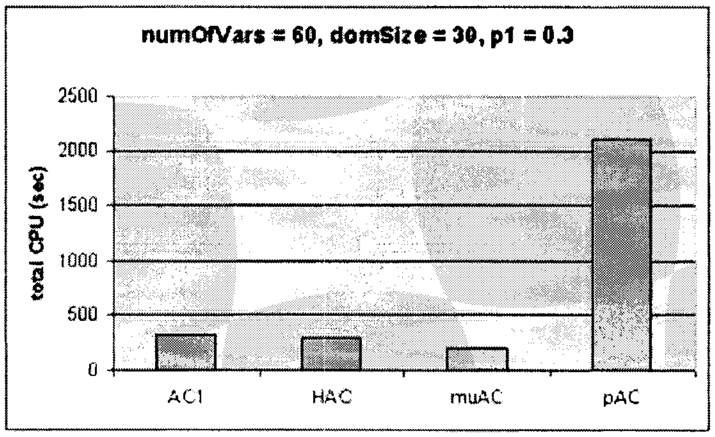


Figure 5.9: Total CPU taken to find first solutions for problems with numOfVar = 60, domSize = 30, p_1 (constraint density) = 0.3, p_2 (constraint tightness) $\in [0.1, 0.3]$

Chapter 6

Conclusions

6.1 Summary

In this thesis, we study the practical use of generalized propagation techniques. Heuristics generated by our algorithms, HAC and μAC, are derived from arc consistency operations directly.

When comparing HAC with previous algorithms which approximate the marginal solution counts on simplified CSPs [5, 16, 24], a major difference is that those algorithms ignore loops in the constraint network and assume single connection or independence among subproblems. The deletion of a domain value is not propagated through the constraint network. The resulting marginal solution counts are not locally consistent with their neighboring variables. Our algorithms maintain the original structure of the constraint graph. Deletions of locally inconsistent domain values are propagated through the constraint network while arc consistency is maintained throughout the whole process. The pAC algorithm does not ignore loops in the constraint graph. However, it tries to generate solution probabilities that are as accurate as possible. The pAC algorithm not only propagates the deletion of a domain value, it also propagates solution probabilities of these domain values until they do not change. Any small changes in the solution probabilities will trigger a mass constraint propagation. This causes an over counting problem and convergence problem. Our algorithms instead compute a quick estimation. We propagate the marginal solution counts until no value is deleted from any variable domain thus guaranteeing convergence.

Experimental results show that when used as value ordering heuristics, HAC and μAC algorithms performed almost as good as the pAC algorithm, however, take much less CPU time. The right balance is established between heuristic guidance, stability and efficiency.

Comparing HAC with Geelen's value ordering heuristic [6], the major difference is with respect to their implementation. Updating the marginal solution counts incrementally during the process of achieving arc consistency gets the same results as recalculating domain supports whenever the domain of a neighboring variable is changed but requires less computation.

The μAC algorithm takes one step further. It assumes accuracy of the marginal solution counts generated by HAC and removes all domain values with small marginal solution counts from variable live domains. These forced domain reductions are propagated through the constraint network. Experimental results show that, although the approximated marginal solution counts generated by the HAC algorithm provide good heuristic guidance for the search algorithm, it is the aggressive value pruning of the μAC algorithm that really improves the performance.

6.2 Discussion and Future Work

In this thesis, we have shown that our new algorithms overcome the shortcomings of the pAC algorithm. The estimated marginal solution counts generated by these algorithms are fast to compute and provide useful heuristic information. However, when the problem size is small, HAC Look-ahead and μAC Look-ahead algorithms take longer CPU time to find solutions than the traditional Full Look-ahead algorithm. The reason is that HAC and μAC algorithms use integer operations while the traditional arc consistency algorithm uses boolean operations. Furthermore, when choosing the next domain value to instantiate a variable, these algorithms need to search through each variable domain to find a value with the largest HAC valuation. This step takes longer CPU time compared to the Full Look-ahead algorithm, which chooses any value from a variable's live domain. Our experimental results show that, for larger CSP problems, the reduced search cost compensates for the overhead incurred in the proposed algorithms.

For future research directions, we suggest the following: 1). explore other generalized propagation schemes and combination operators for estimating the marginal solution counts. 2). further investigate the performances of HAC and μAC algorithms using real-world problems.

Bibliography

[1] D. Achlioptas, L. M. Kirousis, E. Kranakis, D. Krizanc, M. S. O. Molloy, and Y. C. Stamation. Random constraint satisfaction: A more accurate picture. In *Proceedings of CP97*, pages 107–120, 1997.

[2] C. Bessiè and M. O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 108–113, 1993.

[3] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaille, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties and comparison. *Constraints*, 4(3):199–240, 1999.

[4] J. R. Bitner and E. Reingold. Backtrack programming techniques. *Comm. ACM*, 18(11):651–656, 1975.

[5] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–34, 1988.

[6] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 31–35, 1992.

[7] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.

[8] R. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[9] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proc. of the Fourteen International Joint Conference on Artificial Intelligence(IJCAI-95)*, pages 607–615, 1995.

[10] W. S. Havens. Nogood caching for multiagent backtrack search. In *Proceedings of AAAI'97 Constraints and Agents Workshop*, 1997.

[11] P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[12] M. C. Horsch and W. S. Havens. An empirical evaluation of probabilistic arc consistency as an variable ordering heuristic. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 525–530, 2000.

[13] M. C. Horsch and W. S. Havens. How to count solutions to csps. Technical report, School of Computing Science, Simon Fraser University, 2000.

[14] M. C. Horsch, W. S. Havens, and A. Ghose. Generalized arc consistency with application to maxcsp. In R. Cohen and B. Spencer, editors, *Proceedings of Canadian Conference on AI*, pages 104–118, 2002.

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

[16] A. Meisels, S. E. Shimonoy, and G. Solotorevsky. Bayes networks for estimating the number of solutions to a csp. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 1–34, 1997.

[17] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[18] B. Nadel. Tree search and arc consistency in constraint-satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. New York: Springer-Verlag, 1988.

[19] E. Neopolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms.* Wiley, New York, 1990.

[20] B. Nudel. Consistent labeling problems and their algorithms: Expected complexities and theory-based heuristics. *Artificial Intelligence*, 21, 1983.

[21] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* San Mateo, CA: Morgan Kaufmann, 1988.

[22] N. M. Sadeh and M. S. Fox. Variable and value ordering heuristics for the job-shop scheduling constraint satisfaction problem. *Artificial Intelligence Journal*, 86(1):1–41, 1996.

[23] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[24] M. Vernooy and W. S. Havens. An examination of probabilistic value-ordering heuristics. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence*, 1999.