CONSISTENCY TECHNIQUES FOR LINEAR ARITHMETIC AND FUNCTIONAL CONSTRAINTS

ZHANG YUANLIN

NATIONAL UNIVERSITY OF SINGAPORE

1998

CONSISTENCY TECHNIQUES FOR LINEAR ARITHMETIC AND FUNCTIONAL CONSTRAINTS

ZHANG YUANLIN

(B.Eng, EAST CHINA INSTITUTE OF TECHNOLOGY)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

1998

Acknowledgements

First, I express my gratitude to my supervisor Joxan Jaffar for his introducing me to constraint programming which involves typical computer science and interesting mathematics, for valuable discussions and for his help during my study in NUS. As for research, Roland Yap is an incredible information resource for me. I benifit a lot from discussions with Roland Yap and Liu Bing.

I also thank the other members of the Programming Language and System Group for their creating a stimulating enviorenment.

Finally, I thank my wife and parents for their support in the past years.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Organization of the Thesis 3

2 Preliminaries on CLP over Finite Domain **4**

2.1 Constraint Logic Programming 4

2.2 Constraint Satisfaction Problem 8

2.2.1 Basics of Consistency 10

2.2.2 Variable and Value Ordering 12

2.3 CLP Over Finite Domains 13

2.4 Other Related Work 15

2.4.1 CLP(R) 16

2.4.2 Combining Consistency Techniques and Linear Programming . . . 17

3 Consistency on a Special Class of Constraints **19**

3.1 Background 19

3.2 Review of the Algorithms 21

3.2.1 A Generic Algorithm 21

3.2.2 Domain Structure 26

3.2.3 Algorithms for AFC and MC 26

3.3 Functional Constraints 29

3.3.1 Priliminaries of Functional Constraints 29

3.3.2 Domain Structure 33

3.3.3 Implementation of Consistency Enforcing on Functional Constraints 34

3.3.4 Properties of Algorithms for FCs, AFCs and MCs 39

3.3.5 Further Issues 42

4 Bound Consistency over n-ary Linear Constraints 49

4.1 Introduction 49

4.2 Bound Consistency 51

4.2.1 Basics of Interval Arithmetic 51

4.2.2 Bound Consistency on An n-ary Linear Constraint System 52

4.2.3 Bound Consistency Algorithms and Their Complexities 56

4.2.4 An Improved Version of BC-1 61

4.2.5 On Simple Transformation of Constraint System 64

4.2.6 Linear Equation System 67

4.2.7 Issues on Constraint Solver of a CLP over Finite Domain 68

4.3 Bound Consistency on Binary Equation System 69

4.3.1 The Euclidean Algorithm 69

4.3.2 Bound Consistency on Solved Form 71

4.3.3 Better Fixed Point on Solved Form 75

5 On Solving of n-ary Linear Constraints 80

5.1 Mathematical view of bound consistency 81

5.1.1 Fourier elimination 81

5.2 Algebraic Structure and Solving 83

5.2.1 A Puzzle 84

5.2.2 Variable Ordering in Linear Constraints 85

5.2.3 Algebraic Structure, Consistency and Search 86

6 Conclusion and Future Work **90**

6.1 Conclusion 90

6.2 Future Works 91

BIBLIOGRAPHY **93**

List of Figures

3.1 Generic algorithm AC-main. 23

3.2 Domain operations. 27

3.3 Domain structure. 28

3.4 Algorithm for AFCs. 29

3.5 Algorithm for MCs. 30

3.6 Modified domain structure. 33

3.7 *coordinate* structure. 34

3.8 *common* structure. 34

3.9 InitialCheck for FCs. 35

3.10 Algorithm OrientAndCheck. 36

3.11 Algorithm ReorientAndCheck. 37

3.12 Modified algorithm for MCs. 39

3.13 Structures required by the value link list. 43

4.1 Algorithm BC-1. 57

4.2 Algorithm BC-2. 58

4.3 Algorithm BC-1.1. 62

5.1 ALL WOOL 84

List of Tables

5.1 The fixed point achieved on different system 87

5.2 Searching on a dense system. 88

5.3 Number of backtracks on different form. 89

Summary

This thesis focuses on two kinds of important constraints: (n-ary) linear constraints and (binary) functional constraints over finite domains.

n-ary integer linear constraints have been actively used to prune search space by propagating bounds of domain of variables. In this thesis, we formalize this idea to bound consistency. Under the new formalism, we present several consistency enforcing algorithms and give their complexities. Our study shows that in some cases, algebraic manipulation will help to prune search space significantly. Our investigation also shows that by transforming a binary equation system into its solved form, the least fixed point and an efficient consistency enforcing algorithm can be achieved.

Algorithm enforcing arc-consistency on functional constraints with optimal time complexity has been invented. Our research shows that n-consistency can be enforced on such constraints with each functional constraint checked only once. The algorithm enforcing n-consistency on functional constraints has optimal space complexity and under some conditions, it also has optimal time complexity.

Chapter 1

Introduction

1.1 Introduction

The Constraint Programming (CP) paradigm combines the power of constraints (and constraint solving) and programming language for general problem solving. Under this new paradigm, programming is enhanced by the expressiveness of constraints and benefits from a wealthy of constraint solving techniques from AI, Operations Research and other mathematical area. The most prominent example of constraint programming is constraint logic programming (CLP) which greatly advances the development of CP. CLP is not only an elegant framework for a class of constraint programming languages based on Horn clause logic, but also has gained great success in many real-life applications. The commercial versions of CP like CHIP and ILOG have been spread widely in such applications as production planning, scheduling and resource allocation [vH89, Kum92 *etc*]. Specifically, CHIP and ILOG are representatives of constraint programming over finite domains which employ Constraint Satisfaction Problem (CSP) its underlying model, and embed CSP solving techniques in the constraint solvers.

The computation of CLP over finite domain usually involves solving integer linear equations and inequalities, dis-equations over natural number terms, ad-hoc symbolic constraints and user-defined constraints. The basic constraint solving techniques adopted by the constraint solver of CLP over finite domain involves mainly the consistency techniques inherited from CSP. The task confronting a constraint solver is how to effectively deal with those constraints. For the sake of efficiency, constraints are classified into basic constraints and non-basic constraints. Specifically, the basic constraints are a special class of binary constraints, that is functional constraints,anti-functional constraints and monotonic constraints. All the other constraints fall into the category of non-basic constraints. Actually the non-basic constraints can be further classified. The well-known n-ary integer linear equations and inequalities are treated separately in most constraint programming languages.

In this thesis, we will address the following two problems:

- How far can we go for the basic constraints?
- How to deal with n-ary integer linear constraints?

For arc-consistency enforcing on basic constraints, a special algorithm [VHDT92] with optimal time complexity has been proposed. However, under the same time complexity we find that n-consistency can be enforced on functional constraints, and furthermore, each functional constraint is checked only once. Our algorithm for consistency enforcing has optimal space complexity and under some condition has also optimal time complexity. A brief review on consistency techniques can be found in chapter 2 and chapter 3.

n-ary integer linear constraints have been actively used to prune search space by propagating bounds of domains of variables. In this thesis, we formalize this idea as bound consistency. Under the new formalism, we present several consistency enforcing algorithms and give an estimate of their complexities. The relationship between local consistency and global consistency is always an interesting and important topic in CSP.

We give a thorough study on properties of bound consistency for the case of a special linear system–binary equation system. We also give preliminary study on the influence of algebraic transformation of n-ary linear constraints on bound consistency and the solving of linear constraint system.

1.2 Organization of the Thesis

This thesis is organized as follows:

In chapter 2 we briefly describe how constraints are incorporated into rule-based programming languages, the fundamentals of Constraint Satisfaction Problem, the solver of a CLP over finite domain and other related work.

In chapter 3, after an overview of arc consistency techniques, we give detailed analysis of functional constraints and present algorithms for consistency enforcing on them.

Chapter 4 reviews the related area of bound consistency, gives the formal definition of bound consistency and consistency enforcing algorithms, analyses their complexities, and studies property of bound consistency on equivalent binary equation systems.

In chapter 5 we give a general view of bound consistency in a wider setting. Experiments are carried out to study the impact of Gaussian-Jordan elimination on bound consistency and the whole constraint solving process.

Chapter 6 summarizes the results in this thesis and discusses the future work and related research.

Chapter 2

Preliminaries on CLP over Finite Domain

2.1 Constraint Logic Programming

The constraint logic programming [JL87, JM94] defines a class of languages based upon the paradigm of rule-based constraint programming. All the languages of this class are based on a uniform framework of formal semantics. Each logic-based constraint programming language, its declarative semantics, its operational semantics and the relationship between these semantics could all be parameterized by a choice of domain of computation and constraints. The parameters can be characterized by a quadruple $(\sum, \mathcal{D}, \mathcal{L}, \mathcal{T})$. Here $\sum$ is a signature, $\mathcal{D}$ is a $\sum$-structure, $\mathcal{L}$ is a class of $\sum$-formula, and $\mathcal{T}$ is a first-order $\sum$-theory.Intuitively, $\sum$ is the set of the predefined predicate and function symbols and their arities, $\mathcal{D}$ is the structure over which computation is performed, $\mathcal{L}$ is the class of constraints, and $\mathcal{T}$ is an axiomatization of properties of $\mathcal{D}$.A constraint domain can be understood as a tuple $(\mathcal{D}, \mathcal{L})$.

An instance of CLP can be obtained by specifying the constraint domain. The following are some CLP systems over different constraint domains. The traditional logic programming [Llo87] can be viewed as CLP over the constraint domain of finite trees. CLP(R) [JMSY92] has linear arithmetic constraints and computes over real numbers. Nonlinear arithmetics is treated by delay mechanism. Prolog III [Col90] involves several constraint domains including the two valued boolean algebra, linear arithmetic over rational numbers and a domain of strings. CHIP [DVHS88] also has several domains among which the most prominent is finite domains (linear arithmetic constraints and some ad hoc constraints over bounded subsets of integer). There are now several languages dealing with finite domain: $cc(FD)$ [VHSD93], $CLP(FD)$ [CD96], Echidna [HSSJO92] and Flang [Man93].

Let $\sum$ denote the set of function and predicate symbols defined in the constraint domain, and Π the set of predicate symbols defined by a program.A primitive constraint has the form of $q(t_1, \cdots, t_n)$, where $t_1, \cdots, t_n$ are terms over $\sum$ and $p \in \sum$ is a predicate symbol. An atom has the form of $p(t_1, \cdots, t_m)$, where $t_1, \cdots, t_m$ are terms and $p \in \Pi$ is a predicate symbol. A CLP program is a collection of rules of the form

$$a \leftarrow c, B$$

where c is the conjunction of constraints in the body, B is the conjunction of atoms in the body, and a which is called head of the rule is an atom. A goal G is a rule without head. A fact is a rule with only constraints in its body.

The operational semantics,based on the top-down execution, of a CLP program can be described by a transition system on states. Each state is a triple $< A, C, S >$, where A is a multi-set of atoms and constraints, C and S are multi-sets of constraints which are also referred to constraint store upon which a constraint solver acts. A goal G initiates a transition sequence by giving the starting state $< G, \emptyset, \emptyset >$. Here we assume there exists a computation rule which selects a transition type and an appropriate element of A for each state. In order to define a transition, we also assume a predicate *consistent* and a function *infer* which can be considered as two aspects of constraint solving.

The transitions of the transition system are :

$$< A \cup \{a\}, C, S > \rightarrow_r < A \cup B, C, S \cup (a = h) >$$

if a,an atom, is selected by the computation rule, $h \leftarrow B$ is a rule of P, and h and a have the same predicate symbol. $a = h$ means the conjunction of equations between corresponding arguments of a and h.

$$< A \cup \{a\}, C, S > \rightarrow_r fail$$

if a is selected by the computation rule, and there is no rule of P whose head does not have the same predicate as a.

$$< A \cup \{c\}, C, S > \rightarrow_c < A, C, S \cup c >$$

if c is a constraint and it is selected by the computation rule.

$$< A, C, S > \rightarrow_i < A, C', S' >$$

if $(C', S') =$ *infer*(C, S).

$$< A, C, S > \rightarrow_s < A, C, S >$$

if *consistent*(C).

$$< A, C, S > \rightarrow_s fail$$

if $\neg$*consistent*(C).

A derivation is a sequence of transitions

$$< A_1, C_1, S_1 > \rightarrow \cdots \rightarrow < A_i, C_i, S_i > \rightarrow \cdots$$

A state is called final state if it can not be further rewritten. A derivation is successful if it is finite and the final state has the form of $< \emptyset, C, S >$. If a goal G with free variables $\tilde{x}$ initiates a successful derivation whose final state is $< \emptyset, C, S >$, $\exists_{-\tilde{x}} C \cap S$ is called the answer constraint of the derivation.Given a goal G, program P and the computation rule, the search space of all possible derivations starting from $< G, \emptyset, \emptyset >$ can be thought of as a computation tree. The problem of finding answers to a query can be seen as the problem of searching the computation tree to obtain a successful derivation.Most CLP systems employ a depth-first search with chronological backtracking [JM94].

In the view of the operational semantics, there are several operations on constraints to be implemented. They are satisfiability test to implement *consistent* and *infer*, entailment test to implement guarded goals,and the projection of constraint store onto a set of variables to compute the answer constraint from the final state. The constraint solver must be able to undo the effects of newly added constraints when the inference engine backtracks.

Here we are mainly concerned with satisfiability test : *consistent* and *infer*.The *consistent(C)* is a test of satisfiability of C.*consistent(C)* iff $\mathcal{D} \models \tilde{\exists} C$. Function $infer$ computes from the current set of constraints a new set of active constraints and passive constraints. It varies widely from system to system. In general, there is a tradeoff between a complete *consistent* and an incomplete *consistent*, and between a weak *infer* and a strong *infer* in order to achieve satisfactory efficiency. The implementation of CLP requires that the algorithm for constraint solving must be incremental.

2.2 Constraint Satisfaction Problem

A large number of problems in Artificial Intelligence (AI) and other areas of Computer Science can be expressed as constraint satisfaction problems. Some example applications are machine vision,belief maintenance, scheduling, temporal reasoning,graph coloring, circuit design,boolean satisfiability and logic puzzle solving. CSP has been extensively studied in the area of AI ([Wal72], [Mon74] and [Mac77]). [Tsa93] gives a detailed description on this relatively new area. The definitions in this section follow [Mac77].

Definition 1 A Constraint Satisfaction Problem (CSP) *is a triple* (V, D, C)*, where* $V = \{x_1, \cdots, x_m\}$ *is a finite set of variables of the problem,* $D = \{D_1, \cdots, D_m\}$ *is a collection of finite set of objects and* x_i *can only take values from* D_i*, and* $C = \{C_1, \cdots, C_e\}$ *is a finite set of constraints each of which is defined over a subset of* V*.*

An instantiation of a variable x_i is to assign a value in D_i to x_i. A solution of a constraint $C_i(x_{i_1}, \cdots, x_{i_k})$ is a tuple of instantiations $(v_{i_1}, \cdots, v_{i_k})$ of variables such that $C_i(v_{i_1}, \cdots, v_{i_k})$ is true. A constraint C_i normally takes two forms. One form is represented by explicit tuples of instantiations of subject variables which is a subset of Cartesian product $D_{i_1} \times \cdots \times D_{i_k}$. Another one is represented implicitly, for example, by mathematical expressions.

The domain D_i in fact is a unary constraint on x_i. It is singled out because for any CSP, there always exists such a constraint on each variable. It is this feature together with the finite number of constraints that makes a general solving strategy for CSP possible. The simplest one is the generate-and-test paradigm in which each possible combination of instantiations of all variables is systematically generated and then tested to see if it satisfies all the constraints. An immediate improvement over that paradigm is the backtracking paradigm in which the basic idea is that if an instantiation of some variables can not satisfy the constraints all the extensions of the partial instantiation to the whole set of variable can not satisfy the constraints and thus those extensions need not be explored. Given a CSP (V, D, C),an illustrative algorithm for backtracking paradigm is as follows

0. $k \leftarrow 0$.

1. If all variables in V are instantiated, a solution has been found and the algorithm terminates.Otherwise pick an uninstantiated variable x. Set S_k to be the set of all values of x which are compatible with instantiations so far.

2. If S_k is empty, go to step 3. Otherwise,pick one value for x and remove it from S_k, $k \leftarrow k+1$, and go to 1.

3. $k \leftarrow k-1$. If $k < 0$, there is not solution and algorithm terminates.Otherwise go to 2.

Although the backtracking paradigm is better than a generate-and-test method, its run-time complexity is still exponential. However,the backtracking paradigm is useful because most CSPs are NP-complete. There are many techniques ([Mac77], [Fre78], [Nil80], [HE80] and [DP92] etc.) which have been developed to improve the practical efficiency of the backtracking paradigm. Among those we are interested in the following two approaches.

One way to improve the efficiency of a backtracking-based searching procedure is to prune the search space a priori. The combination of pruning techniques with a basic searching procedure can be distributed on a wide spectrum from one extreme that there is no pruning at all to the other extreme that no backtrack is needed.

Another way is to find a good ordering of variables to be instantiated and the values to be chosen for a variable, which has been proved very useful in practical problems [HE80, Tsa93].

2.2.1 Basics of Consistency

A CSP is called binary CSP if each constraint in C is either unary or binary. A CSP with constraints not limited to unary and binary is referred to general CSP or n-ary CSP. The pruning of searching space is achieved by various levels of consistency ([Wal72], [Mon74], [Mac77], [Fre82], [Fre90], [MF85] and [Dec90]). In this section we will present basic concepts of consistency in the context of binary CSP where the constraint over variables $x_i, x_j (i < j)$ is denoted by C_{ij} for clarity. A binary CSP (V, D, C) can be depicted by a constraint graph (N, E) where $N = V, E = \{(x_i, x_j) \mid C_{ij} \in C\}$. Generally, the edge (x_i, x_j) is understood as two directed arcs $< x_i, x_j >$ and $< x_j, x_i >$ in most consistency enforcing algorithms.

Intuitively, the consistency techniques are used to remove those instantiations on variables, which lead to no solution by making an active use of constraints.

Definition 2 *A CSP (V, D, C) is* node-consistent *iff for each variable x_i all values in its domain D_{x_i} satisfy all the unary constraints on x_i.*

Definition 3 *An arc $< x_i, x_j >$ is* arc-consistent *iff for every value $a \in D_i$ satisfying all the unary constraints on x_i, there exists a value $b \in D_j$ such that b satisfies unary constraints on x_j and $C_{ij}(a, b)$ holds.*

A CSP is arc-consistent *iff every arc in its constraint graph is arc-consistent.*

Definition 4 *Given a path* $p = (x_{i_0}, \cdots, x_{i_k})$*,an instantiation tuple* $< v_{i_0}, v_{i_k} >$ *for* x_{i_0} *and* x_{i_k} *is* path consistent *with regard to* p *iff there exists* $(v_{i_1}, \cdots, v_{i_{k-1}})$ *such that* $v_{i_l} \in D_{i_l} (1 \leq l \leq k-1)$ *and* $C_{i_l i_{(l+1)}}(v_{i_l}, v_{i_{l+1}})(0 \leq l < k)$.

A path $p = (x_{i_0}, \cdots, x_{i_k})$ *is* path consistent *iff for all* $v_{i_0} \in D_{i_0}$ *and* $v_{i_k} \in D_{i_k}$ *with* $C_{i_0 i_k}(v_{i_0}, v_{i_k})$, $< v_{i_0}, v_{i_k} >$ *is path consistent with regard to* p.

A CSP is path consistent *iff every path in the associated constraint graph are path consistent.*

The ideas of node, arc and path consistency are generalized by Freuder [Fre78] as k-consistency.

Definition 5 *A CSP is* 1-consistent *iff for any variable* x_i *and any value* $v_i \in D_i$ *the unary constraints on* x_i *hold.*

A CSP is k-consistent *iff given any instantiation*$(v_1, \cdots, v_{k-1})$ *of any* $k-1$ *variables which satisfies all constraints over those variables, there exists an instantiation of any* k*-th variable such that* $(v_1, \cdots, v_k)$ *satisfies all constraints over the* k *variables.*

A CSP is strongly k-consistent *iff it is* j*-consistent for all* $j \leq k$.

Node,arc and path consistency correspond to 1-, 2-, and 3-consistency respectively. Generally speaking, the stronger the level of consistency one achieves, the more computation one requires and the more useless searching space one can prune. It is obvious that when a CSP is strongly m-consistent a solution can be found without backtrack [Tsa93].

Numerous algorithms have been invented to enforce certain level of consistency over a CSP. The theme is how to improve the efficiency of the consistency enforcing algorithms generally and specifically. In chapter 3 we will give more details on consistency enforcing algorithms.

2.2.2 Variable and Value Ordering

Assuming tree search with chronological backtracking is used, the possible gains of a special ordering of variable could be

1. Failures could be detected earlier than other orderings.

2. With some consistency technique, larger portion of the search space can be pruned off than other orderings.

Many heuristics for variable ordering have been accumulated [Tsa93]. Among them are minimal width ordering (MWO) heuristic,minimal bandwidth ordering (MBO) heuristic and first fail principle (FFP). MWO [Fre82] orders variables before search by exploiting the topology of the nodes in its associated constraint graph. MBO [Zab90] obtains the variable ordering by exploiting the structure of the associated constraint graph. FFP [HE80,BP81] is a very general heuristic for searching. It suggests that the task which is most likely to fail should be performed first. It may be used to produce both dynamic and static variable ordering. Chapter 5 will present a heuristics based on this principle. In practice, a good ordering of variables can drastically improve the efficiency of the search procedure.

It has long been suggested that the efficiency of a search procedure for general search problems can be greatly affected by the ordering in which the branches denoting possible values for each variable are explored [Nil80]. In choosing a value to assign to the chosen variable, a good general strategy is to choose, if possible, a value which is likely to lead to a solution and thus reduce the possibility of having to backtrack on this variable and try an alternative value.

2.3 CLP Over Finite Domains

Constraint Logic Programming over finite domains has been investigated in [DVHS88], [vH89], [CD96] and [VHSD93] etc.. The paradigm of CLP over finite domains provides a perfect language to describe and solve constraint satisfaction problems. It has been shown that many real-life CSPs can be solved with an efficiency comparable to procedural programs, yet with a much smaller development time [DVHS87].

The intended application area of CLP over finite domain is to solve those problems which can be modeled by CSPs. Typical constraints in such a language include linear arithmetic constraints and ad hoc symbolic constraints [VHD91]. Each variable in this language is associated with a subset of integers, which is called the domain constraint. The essential computation step involves checking the satisfiability of the constraint store $< C, S >$. Obviously,there exists a complete constraint solver to determine *consistent*(C, S) as is shown in section 2.1. However such an approach on one hand is too time consuming (exponential in time complexity) to be afforded by the solver, on the other hand is not appropriate for a large class of problems which require different solutions and have special properties which can be exploited to efficiently solve the problems at hand. An alternative is to employ an incomplete solver which is equipped with some consistency technique [vH89].

In the constraint store $< C, S >$, let C be the set of domain constraints, and S be the set of all the other constraints. The predicate *consistent*(C) is true if and only if no domain in C is empty. The function *infer* can be implemented as consistency enforcing to achieve certain level of consistency. In order to improve efficiency and flexibility, the constraint store can be classified into different categories for each of which there is an efficient algorithm to implement *infer*. Without considering the symbolic constraints, the linear constraints can be separated into basic constraints and non-basic constraints. The basic constraints include

$$ax \neq b, ax = b, ax = by + c, ax \geq by + c, ax \leq by + c.$$

where $a, b, c \geq 0$ and $a \neq 0$. All the other binary constraints and n-ary constraints fall into the class of non-basic constraints.

For basic constraints, the arc-consistency combined with *consistent()* is sufficient to guarantee a complete solver [VHDT92]. In this case *infer*$(C, S) = (C', S)$. The role of *infer* is twofold. One is to help test the satisfiability and the other one is to prune the search space of the problem at hand. In order to achieving more pruning, the path even k-consistency can be used to implement *infer*. Chapter 3 gives an implementation of arc-consistency which achieves optimal space and time complexity and higher consistency for a special class of constraints.

The arc-consistency of an n-ary constraints [MM88b] can be obtained by a natural generalization of that of binary constraint. However, for n-ary (non-binary) constraints, such a generalization is too expensive to be afforded. Here, the semantics of the constraint has to be considered to obtain a relaxation of arc-consistency. The basic idea is to relax the domain of the variable to real number and at the same time keep the bounds of the domain integer. We call such a relaxation bound consistency. Chapter 4 gives a formal discussion on bound consistency and related consistency enforcing algorithms.

Arc-consistency enforcing algorithms are incremental in nature. Once a constraint is added to store, the *infer* will be triggered and sequentially the *consistent()* is invoked to check the satisfiability. Generally, the function *infer* deals with different levels of of consistency corresponding different constraints. The new store (C', S) satisfies all the consistency requirements.

A methodology of solving constraint satisfaction problem in CLP is

$P(\cdots)$:-

$\cdots$ Primitive constraints $\cdots$,

$\cdots$ Constraints expressed with predicates $\cdots$,

$\cdots$ Generators for the variables.

A CLP language normally provides built-in generators so that heuristics such as variable and value ordering can be fully exploited.

2.4 Other Related Work

The linear constraints considered in the previous section actually can be described as an integer programming problem as follows

$$Ax \diamond b$$

$$A \in Z^{m \times n}, x \in Z^n, b \in Z\text{'}\diamond \in =, \leq$$

A typical technique to solve the above problem is to relax integer domain of variable to real domain so that standard linear programming techniques can be used. An interesting observation is that if the relaxed problem does not have solution the original problem will not have solution too, and if an feasible solution of the relaxed problem is integral it is also the solution of the original problem. The disadvantage of such a relaxation is that satisfiability of linear constraints over real domain does not mean its satisfiability over integer domain. The following section will show the linear programming techniques incorporated in CLP(R).

2.4.1 CLP(R)

The constraint domain of CLP(R) is linear constraints over real numbers (Non-linear constraint is also allowed in CLP(R) but it is omitted here because of it is irrelevance to our concerns). The success of CLP(R) has shown that the linear programming techniques can be efficiently implemented in a CLP language.

The satisfiability test of linear constraints over real domain in CLP is a special case of the linear problem

$$\min cx$$

$$s.t. Ax = b, \qquad x \in R^n.$$

which will minimize the objective function cx where x is subject to linear constraints $Ax = b$.

The basic idea of solving the problem ([Dan63]) is as follows. The set of linear constraints defines a polyhedron. An optimal solution to the linear program is located on one of the vertices of the polyhedron. The simplex method finds an optimal solution by moving from a vertex to an adjacent one with a smaller value for the objective function. An adaption of first phase simplex method is used in CLP(R) [JMPY92] to test the satisfiability of a set of linear constraints.

Specifically, the linear constraints in CLP(R) are partitioned into two classes: one for equations and the other for inequalities. Gaussian elimination is used to solve the equations and the adapted first phase simplex method is employed to solve the inequalities. A mechanism is provided to coordinate the two different solvers. In addition to achieving satisfiability test, the constraint solvers can also detect fixed variables (which are grounded) as soon as possible and always maintain a feasible solution for the current active constraint store.

Note the difference between constraint solving techniques used in CHIP and CLP(R).

The arc-consistency techniques consider each constraint individually and the interaction between different constraints is achieved through the domain of shared variable(s). In linear programming, linear constraints are treated as a whole and a global solution can be obtained. Consider example

$$x_1 + x_2 = 3$$
$$x_1 - x_2 = 5$$
$$-3 \leq x_1, x_2 \leq 3$$

Unsatisfiability can not be detected by arc consistency, but can be detected by simplex method(for the above system Gaussian elimination can help us find the satisfiability easily). The combination of these two approaches of CSP and linear programming obviously detects unsatisfiability earlier and thus achieves more pruning of search space. As usual, such a combination incurs an overhead which is not negligible.

2.4.2 Combining Consistency Techniques and Linear Programming

[RWH97] proposes a system which employs the finite domains solvers of ECLiPSe and a mixed integer programming solving system CPLEX. The system still uses CLP as its language because of the ease of CLP to model combinatorial optimization problems. Because CPLEX uses only linear programming as its input, a CLP program has to be transformed to the standard form recognized by CPLEX.

The kernel of the new system is a hybrid algorithm. Essentially the hybrid algorithm is a backtracking algorithm combined with consistency enforcing at each search step and simplex based satisfiability test at certain steps. A CLP language ECLiPSe is employed to implement the backtracking mechanism and consistency enforcing. At certain search node CPLEX is triggered to check the satisfiability of the constraint store and if constraint store is satisfiable, provides a relaxed solution for the constraint store. Note that CPLEX may not be incremental.

Their experiment results show that for some difficult problems, for example the Progressive Party Problem, which cannot be solved in reasonable time by CLP or CPLEX alone, can be efficiently solved by the combined system.

Another work is done by Beringer and Backer [BDB95]. Motivated by the constraint, called mixed constraint, which spans over integer domain and real domain,they implemented both finite domain solver and a revised simplex based linear solver in ICE. Those mixed constraints are sent to both solvers. The authors' concern is how to make full use of the linear solver and finite domain solver. Their strategy is that when the bound of a variable is changed (by finite domain solver), the new bound will be sent to linear solver. The intuition behind this is that the finite domain solver has done some rounding on some bounds such that the new bounds can help linear solver to make a more accurate decision. As soon as a CLP employs both finite domain solver and linear solver, the idea of [RWH97] can be easily implemented in such a language by allowing programmer to put one finite domain constraint to both solvers when necessary.

In addition to abovementioned advantages of the combination, we are specially concerned with the influence of the algebraic transformation of a system on consistency techniques and the searching procedure.Specifically, Chapter 5 and part of chapter 4 give an rudimentary study about the influence of Gaussian-Jordan elimination on consistency techniques and searching efficiency.

The most related work to us is CIAL [CL94]. From the point view of constraint solving,CIAL incorporates the consistency techniques and Gaussian elimination tightly,which means that the consistency is enforced on the transformed linear constraints rather than the original ones. However, no further analysis on such a combination is given .

Chapter 3

Consistency on a Special Class of Constraints

3.1 Background

As a key technique in solving constraint satisfaction problem, consistency technique has been studied extensively in past two decades. In particular, originating from the Waltz filtering algorithm [Wal72], a number of arc consistency algorithms have been proposed. Among them are AC-1 to AC-3, which are summarized and refined by Mackworth [Mac77], AC-4 [MH86],which is an optimal algorithm, AC-5 [VHDT92], which is a generic algorithm and can be specialized to AC-3 and AC-4 separately ,and AC-6 [BC93],which improve the space complexity of AC-4. All these algorithms are designed for general CSPs. AC-3 has the optimal space complexity of $\mathcal{O}(e+nd)$ [MH86], and AC-4 and AC-6 have the optimal time complexity of $\mathcal{O}(ed^2)$, where e is the number of constraints and d the size of the greatest domain. There are also some specialized algorithms [Lau78] [MM88] [VHDT92] that are designed by considering the semantics of constraints and thus more efficient.

In this chapter, we are specially interested in algorithms for a class of binary constraints, precisely the functional constraint (FC), anti-functional constraint (AFC) and monotonic constraint (MC). Actually these three kinds of constraints correspond to linear constraint and dis-equation which play an important role in a CLP language. A specialization of AC-5 [VHDT92] for FCs, MCs and AFCs achieves optimal time complexity $\mathcal{O}(ed)$ with space complexity $\mathcal{O}(e+nd)$. In [Liu96], a new kind of constraint,increasing functional constraint (IFC), is identified and an efficient algorithm with optimal space complexity for IFC,MC and AFC is proposed. In this chapter we will present an algorithm for FC, AFC and MC which can be optimal in both time and space complexity and all the functional constraints need to be checked only once as in [Liu95] for IFC. Furthermore, the algorithm achieves strongly n-consistency on functional constraints.

Hereafter, the CSP is restricted to binary CSP and we assume that there is only one constraint over a pair of variables, that is in the associated graph of a CSP there is only one edge (two directed arc) between two nodes. $arc(G)$ is used to denote the set of directed arcs in G. For simplicity, the variable x_i is denoted by i. We further assume that there is a total ordering on the domain.

Definition 6 [VHDT92] *A constraint C_{ij} is* functional *with respect to domain D_i and D_j iff for all $v \in D_i$ (respectively $w \in D_j$) there exists at most one element $w \in D_j$ (respectively $v \in D_i$) such that $C_{ij}(v,w)$.*

If C_{ij} is functional, for any $v \in D_i$(respectively $w \in D_j$) if there exits $w \in D_j$(respectively $v \in D_i$) such that $C_{ij}(v,w)$ holds, we use $f_{ij}(v)$ (respectively $f_{ji}(w)$) to represent w(respectively v). An example of functional constraint in CLP is an equation $x = -y$ with $x \in \{1,2,3\}, y \in \{-1,-2,-3\}$.

Definition 7 [Liu95] *A functional constraint $C_{ij}$is* increasing *with respect to domain*

D_i and D_j iff for any $u, v \in D_i$ such that $f_{ij}(v)$ $f_{ij}(u)$ exist in D_j, $u < v$ implies $f_{ij}(u) < f_{ij}(v)$.

The equation $x = y$ is an increasing functional constraint.

Definition 8 [VHDT92] *A constraint C_{ij} is* anti-functional *with respect to domain D_i and D_j iff $\neg C_{ij}$ is functional with respect to D_i and D_j.*

[VHDT92] The dis-equation $x \neq y$ in CLP is anti-functional because $x = y$ is functional.

Definition 9 *A constraint C_{ij} is* monotonic *with respect to domain D_i and D_j iff there exists a total ordering on D_i and D_j such that for all $v \in D_i$ and $w \in D_j$, $C_{ij}(v, w)$ implies $C_{ij}(v', w')$ for all $v' \leq v$ and $w' \geq w$.*

The inequality $x \leq y$ in CLP is monotonic. Note $x \leq -y$ is not monotonic.

Before we give special techniques to attack functional constraints, we recall the algorithms for IFC,FC,AFC and MC in next section.

3.2 Review of the Algorithms

3.2.1 A Generic Algorithm

All algorithms for arc-consistency employ a queue as basic structure. The queue contains the necessary information for consistency enforcing. The queue of AC-3 contains the arc $< i, j >$ which need to be rechecked. The queue of AC-4 constraints (i, v) , where i is a node and v is a value, which has an associated support set. The element of queue of AC-5 is of (i, j, w) where $< i, j >$ is an arc and w is a value that has been removed from D_j and justifies the need to reconsider arc $< i, j >$. In AC-5*, the element of the queue takes two forms: one is of (i, j, w), and the other is of $< i, j >$.

One important observation about AC-5 and its specialization is that for AFCs and MCs, value w is not used and only the minimum and maximum values and the domain size are used.As for functional constraints, w plays a key role in achieving an $\mathcal{O}(ed)$ algorithm in [VHDT92]. Our observation is that without the help of w, an $\mathcal{O}(ed)$ algorithm can still be achieved.For FC, AFC and MC, the form $< i, j >$ is enough and (i, j, w) is reserved for other general constraints. Due to the $< i, j >$ structure, for FCs, AFCs and MCs, the algorithm can also achieve the optimal space complexity. Because we are only concerned with FC, AFC and MC, we use only queue element structure $< i, j >$ in our description.

The basic operations on the queue are Enqueue, which add element(s) to the queue, and Dequeue, which remove an element from the queue.

```
**Procedure** Enqueue(**in** $i$,**inout** $Q$)
  **begin**
    $Q \leftarrow Q \cup \{< k, i > \mid < k, i > \in arcs(G)\}$
  **end**
```

```
**Procedure** Dequeue(**inout** $Q$, **out** $i, j,$)
  **begin**
    delete $< i, j >$ from $Q$
  **end**
```

Corresponding to the change of queue structure, AC-5 is slightly changed as shown in figure 3.1. AC-main is parameterized by two open procedures InitialCheck and ReCheck.

Intuitively, InitialCheck will check the consistency of each constraints once and prepare necessary data for ReCheck. ReCheck will check the necessary constraints according to the data provided by InitialCheck. $\triangle$ is the set of values in D_i which are not supported by D_j under constraint Cij.Remove($\triangle, i$) removes those invalid values $\triangle$ from

```
Algorithm AC-main
1 begin
2   $Q \leftarrow \emptyset$;
3   for each $< i, j > \in arc(G)$ do
4     begin
5       InitialCheck($i, j, \triangle$);
6       Enqueue($i, Q$);
7       Remove($\triangle, D_i$);
8     end;
9   while $Q \neq \emptyset$ do
10    begin
11      Dequeue($Q, i, j$);
12      ReCheck($i, j, \triangle$);
13      Enqueue($i, Q$);
14      Remove($\triangle, D_i$);
15    end
16end

Procedure InitialCheck(in $i, j$. out $\triangle$ )
  $\triangle \leftarrow \{v \in D_i \mid \forall w \in D_j \ \ \neg C_{ij}(v, w)\}$
Procedure ReCheck(in $i, j$. out $\triangle$ )
  $\triangle \leftarrow \{v \in D_i \mid \forall w \in D_j \ \ \neg C_{ij}(v, w)\}$
Procedure ReCheck(in $(i, j, w)$. out $\triangle$ )
  $\triangle_1 \subset \triangle \subset \triangle_2$
    where$\triangle_1 \leftarrow \{v \in D_i \mid C_{ij}(v, w) \ \ \forall w' \in D_j \ \ \neg C_{ij}(v, w')\}$
      $\triangle_2 \leftarrow \{v \in D_i \mid \forall w' \in D_j \ \ \neg C_{ij}(v, w')\}$
```

Figure 3.1: Generic algorithm AC-main.

the domain of i. Note lines 5-7 may be integrated in one procedure as shown in later implementation of functional constraints.

As for AC-main, we have following results.

Proposition 1 [VHDT92]

(1) the AC-main is correct.

(2) If the time complexity of InitialCheck is $\mathcal{O}(d^2)$ and the time complexity of ReCheck is $\mathcal{O}(d)$, then the time complexity of AC-main is $\mathcal{O}(ed^2)$.

(3) If the time complexity of InitialCheck is $\mathcal{O}(d)$ and the time complexity of ReCheck is

$\mathcal{O}(\triangle)$, then the time complexity of AC-main is $\mathcal{O}(ed)$.

Proof:

(1) refer to [VHDT92].

(2) the proof can be obtained from the proof of (3) in the above proposition.

Here, we give an alternative proof of (3) which will be useful in later analysis of the complexity of the proposed algorithms.

a. The algorithm can terminate.

b. Obviously, the complexity of first loop (**for** loop) is $\mathcal{O}(ed)$. Consider the **while** loop. After rearrangement of the expansion of the **while** loop (almost all proofs in this chapter follow this way), we have the following sequence (R for ReCheck and E for Enqueue):

$$\begin{array}{llll}
\cdots \; R(i-1,j') & \overbrace{\overbrace{R(i,j_1),\cdots,R(i,j_1)}^{m_{<i,j>}}\cdots R(i,j_{d_i})\cdots R(i,j_{d_i})}^{i\text{th node}} & R(i+1,j'') & \cdots \\
\cdots \; E(i-1) & \underbrace{E(i)\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots E(i)}_{n_i} & E(i+1) & \cdots
\end{array}$$

Assume total number of iterations of **while** is γ, for arc $< i,j >$ ReCheck is invoked $m_{<i,j>}$ times, and for each invocation of ReCheck for $< i,j >$ there are $\mid \triangle_k \mid$ elements deleted and the complexity of ReCheck is β_k,where

$$\beta_k = \begin{cases} 1 & if \mid \triangle \mid_k = 0, \\ \mid \triangle_k \mid & otherwise \end{cases} \tag{3.1}$$

$$\sum_{k=1}^{m_{<i,j>}} \mid \triangle_k \mid \leq d \tag{3.2}$$

$$\sum_{<i,j> \in arc(G)} m_{<i,j>} = \gamma \tag{3.3}$$

$$m_{<i,j>} \leq d \tag{3.4}$$

Let n_i be the number of times Enqueue is invoked for node i and α_k be the complexity of Enqueue.We have

$$n_i = \sum_{l=1}^{d_i} m_{<i,l>}.\alpha_k = \begin{cases} d_i & \text{if some element has been deleted,} \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

where d_i is the degree of the node i in graph G.

Enqueue can cause at most dd_i arcs enter Q for node i because each invocation of Enqueue d_i arcs will be entered the Q. For all nodes, there are at most $\sum_{i\in V} dd_i = 2ed$ arcs entering Q. That in each iteration Dequeue will remove one arc from Q implies $\gamma \leq 2ed$.

The total time consumed by ReCheck for $< i, j >$ is

$$CR_{<i,j>} = \sum_{1}^{m_{<i,j>}} \beta_k \leq d + m_{<i,j>}.$$

So, for G,by (3.5) the time spent on ReCheck is

$$\sum_{<i,j>\in arc(G)} CR_{<i,j>} \leq ed + \gamma \leq 3ed.$$

For Enqueue, the time consumed for i is

$$CE_i = \sum_{1}^{n_i} \alpha_k \leq dd_i + n_i.$$

So, the complexity of Enqueue is $\sum_1^m CE_i \leq 4ed$. Therefore, the AC-main is $\mathcal{O}(ed)$. □

For general constraints, $\mathcal{O}(ed^2)$ is the optimal time complexity. Intuitively, it has to take d^2 step to check the consistency of each constraint and totally we have e constraints. One way to further reduce the complexity is to exploit the semantics of constraints. In [VHDT92], FC, AFC and MC are identified and specializations of InitialCheck and

ReCheck for FCs, AFCs and MCs have been implemented in $\mathcal{O}(ed)$ (see also [MM88a]). [Liu95] proposes an InitialCheck for IFCs so that IFCs need not be considered in ReCheck. In addition, [Liu95] achieves optimal space complexity for IFCs, AFCs and MCs. Here, we further develop the idea of [Liu95] so that all FCs need to be considered only in InitialCheck and the optimal space complexity can be achieved for FCs, AFCs and MCs. The challenge here is how to implement InitialCheck for functional constraints without help of the removed value w and the implementation should not affect the time complexity of treatment of other constraints.

3.2.2 Domain Structure

Because of three different kinds of constraints, the domain structure and operations are more involved than those usually required by arc-consistency algorithms.

The implementation of InitialCheck and ReCheck requires such operations on domain as shown in figure 3.2.

The function Size is used by anti-functional constraints. Functions Min,Max,Succ,and Pred are used by monotonic constraints. In order to achieve the goal that all the operations above should take constant time (or reasonable time), [VHDT92] gives following domain data structures (see figure 3.3) for sparse domains.

All fields of domain are maintained by RemovElem(). It is obvious that all the primitive operations on domain take constant time under the above data structure for domain (the expected time of domain membership test, under reasonable assumption, is constant [CLR90]).

3.2.3 Algorithms for AFC and MC

An AFC is consistent as long as none of D_i and D_j has less than two values (see

```
**Function** Size(**in** $D$): Integer
  **begin**
    Size = $|D|$.
  **end**
**Procedure** RemovElem(**in** $v$, **inout** $D$)
  **begin**
    $D = D - \{v\}$
  **end**
**Function** Member(**in** $v$,$D$):Boolean
  **begin**
    Member = $(v \in D)$
  **end**
**Function** Min(**in** $D$): Value
  **begin**
    Min = **min** $\{v \in D\}$
  **end**
**Function** Max(**in** $D$): Value
  **begin**
    Max = **max** $\{v \in D\}$
  **end**
**Function** Succ(**in** $v, D$): Value
  **begin**
    **if** $\exists v' \in D \;\; v' > v$
      Succ = **min** $\{v' \in D \mid v' > v\}$
    **else**
      Succ = $-\infty$
  **end**
**Function** Pred(**in** $v, D$): Value
  **begin**
    **if** $\exists v' \in D \;\; v' < v$
      Succ = **max** $\{v' \in D \mid v' < v\}$
    **else**
      Succ = $+\infty$
  **end**
```

Figure 3.2: Domain operations.

Let the initial domain be $D_{i_0} = \{v_1, \cdots, v_l\}$ with $v_k < v_{i_{k+1}}$
Let current domain be $D_i = \{v_{i_1}, \cdots, v_{i_p}\}$ with $v_{i_k} < v_{i_{k+1}}$ and $i_k < i_{k+1}$

Syntax

$D_i.size$:$\{1, \cdots, l\}$
$D_i.min$:$\{1, \cdots, l\}$
$D_i.max$:$\{1, \cdots, l\}$
$D_i.element$:set of couples$(e, index)$ with $e \in D_{i_0}$ and $index \in \{0, \cdots, l\}$,
organized as a hash table on key e.
$D_i.value$:array $[1..l]$ of elements of D_{i_0}
$D_i.succ$:array $[1..l]$ of integers of $\{1..l\}$
$D_i.pred$:array $[1..l]$ of integers of $\{1..l\}$

Semantics

$D_i.size$: p
$D_i.min$: i_1
$D_i.max$: i_p
$D_i.element\ (v) =$ i_k, if $\exists k\ \ v = v_{i_k}$
0, otherwise
$D_i.value[i_k] = v_{i_k}$
$D_i.succ[i_k] = i_{k+1}\ \ 1 \leq k < p$
$D_i.succ[i_k] = +\infty$
$D_i.pred[i_{k+1}] = i_k\ \ 1 < k \leq p$
$D_i.pred[i_{k+1}] = -\infty$

Figure 3.3: Domain structure.

figure 3.4).

For MC, we associate each arc $< i, j >$ with three functions f_{ij}, $last_{ij}$ and $next_{ij}$, and a relation $\succ_{ij}$ to define an ordering of values for the domains D_i and D_j. Given a constraint C_{ij}, for arc $< i, j >$ the functions and relations are defined as:

$$f_{ij}(w) = max\{v \mid C_{ij}(v, w)\},$$

$$last_{ij} = Max, \quad next_{ij} = Pred,$$

$$\succ_{ij} = >$$

```
Procedure InitialCheck(in $i, j$ ,out $\triangle$)
  begin
    $s \leftarrow Size(D_j)$;
    $w_1 \leftarrow Min(D_j)$;
    if s=1
      $\triangle \leftarrow \{f_{ji}(w_1)\} \cap D_i$
    else
      $\triangle \leftarrow \emptyset$
  end
Procedure ReCheck(in $i, j$ ,out$\triangle$)
  begin
    InitialCheck($i, j, \triangle$)
  end
```

Figure 3.4: Algorithm for AFCs.

while for $< j, i >$ they are defined as

$$f_{ji}(v) = min\{w \mid C_{ij}(v, w)\},$$

$$last_{ij} = Min, \quad next_{ij} = Succ,$$

$$\succ_{ij} = < .$$

The point here is that the check always starts from the Min or Max which guarantee the complexity of InitialCheck is proportional to the number of deleted values. In the following procedure (figure 3.5), all the subscripts are omitted for simplicity.

Now, we are ready to present the implementation of functional constranits.

3.3 Functional Constraints

3.3.1 Priliminaries of Functional Constraints

Given a CSP, let G be the associated graph as before, and G' be the subgraph of G which is associated with all the functional constraints and involved variables of the CSP. $V(G)$

```
**Procedure** InitialCheck(**in** $i, j$ ,**out** $\triangle$)
  **begin**
    $\triangle \leftarrow \emptyset$;
    $v \leftarrow$ *last* $(D_i)$;
    **while** $v \succ f(last(D_j))$
      **begin**
        $\triangle \leftarrow \triangle \cup \{v\}$;
        $v \leftarrow$ *next* $(v, D_i)$
      **end**
  **end**
**Procedure** ReCheck(**in** $i, j$ ,**out** $\triangle$)
  **begin**
    InitialCheck$(i, j, \triangle)$
  **end**
```

Figure 3.5: Algorithm for MCs.

denotes the set of nodes of G and $E(G)$ the set of arcs of G.

Definition 10 *Given a CSP, each connected subgraph of G' is called a* functional block.

Without loss of generality, we assume G' is connected in the following presentation and thus G' is a functional block.

Definition 11 *The* coordinate *of a value $w \in D_j$ with regard to D_i under f where $f : D_i \to D_j$ is a bijection , is $f^{-1}(w) \in D_i$. The domain D_i is called the* reference domain *of D_j with regard to f. The variable i is called the* reference variable.

Property 1 *The reference domain is reflexive, symetric and transitive.*

Definition 12 *A domain D_i is* oriented *by D_j under f if D_j is selected as the reference domain of D_i with regard to f.*

Because of our assumption that there is only one relation between any two variables, hereafter when it is not confusing, we say "D_i is oriented by D_j" by omitting f.

After arc-consistency enforcing on G' we have the following result:

Property 2 *Each functional constraint will be a bijection, and thus domains in the same functional block have the same size.*

If there is a path from i to j, the domain D_j can be oriented by D_i under the composition of functions along the path, and thus all domains in the block can be oriented by the same domain.

Definition 13 *We can always designate a reference domain for all domains in a functional block. This domain is called the* origin *of the block.*

Once the origin of a block is determined, sometimes we abuse the origin to represent the block.

In an arc consistent graph G', from node i to j there may exist more than one path, which means that domain D_j can be oriented by the same domain under different functions. The coordinate of value of D_j may be different with different functions. For that case, we have following proposition.

Proposition 2 *An instantiation v_i of x_i can be extended to a solution of G' if and only if v_i has a unique coordinate with regard to the origin of G'.*

Proof

We give the proof of the necessity by contradition. The sufficiency can be proved similarly. Assume $v_i \in D_i$ has two different coordinates w, w' with regard to reference domain D_j. There must exist two different paths p_1 and p_2 from D_i to D_j in G'.Assume $f_1 : D_i \to D_j$ and $f_2 : D_i \to D_j$ are composition of functions along p_1 and p_2 respectively. We have $f_1(v) = w$ and $f_2(v) = w'$, which implies that given v, constraints on p_1 and p_2 can not be satisfied simutaniously. So, v_i will never be able to be extended to a solution of G' and thus a contradiction is obtained. $\square$

According to the above proposition, values with different coordinate under different functions will not appear in any solution of G' and thus the solution of G.Such values should be removed in arc-consistency enforcing. Of course, without the help of special techniques, those values can not be removed by the normal arc-consistency enforcing algorithms.

Corollary 1 *For G', if for each domain D_j, each value of D_j has unique coordinate with regard to the origin of G', G' is strongly n-consistent.*

The origin plays threefold role in our algorithms. The first is to achieve n-consistency, the second is to save space and the third is to make it possible to check each functional constraint once.

Definition 14 *An arc $< i, j > \in G$ is called* a related arc *of a functional block G' iff $j \in V(G')$ and $< i, j > \in E(G) - E(G')$.*

Example:

Given five variables $\{1, 2, 3, 4, 5\}$. Let the domain of the variables be the same $\{a, b, c\}$ and four constraints be

$$C_{12} : \{(a, c), (c, a)\}, C_{23} : \{(a, a), (b, c)\}, C_{34} : \{(a, a), (a, b)\}, C_{25} : \{(a, a), (a, b)\}.$$

The functional block G' has $\{1, 2, 3\}$ as its nodes and $\{< 1, 2 >, < 2, 1 >, < 2, 3 >, < 3, 2 >\}$ as its arcs. Let $i.c$ denote the value c of the domain of variable i. Supposing D_1 be the origin of G', $2.a$ and $3.a$ have the same coordinate $1.c$. Arcs $< 5, 2 >$ and $< 4, 3 >$ are related arcs of G'. Any removal of values with coordinate $1.c$ will cause the check of of the set of related arcs of G',$\{< 4, 3 >, < 5, 2 >\}$.

3.3.2 Domain Structure

In terms of the property of functional block, the domain structure of AC-5 is modified so that all domains in one block share such attributes as $D_i.size$, the origin of the block, and the set of related arcs of the functional block. In the domain structure all values with the same coordinate share the same structure *coordinate* so that once *coordinate.indicator* is **false** all the values will be considered being removed from corresponding domains.

The modified domain structure is listed in figure 3.6, figure 3.7 and figure 3.8.

Let the initial domain be $D_{i_0} = \{v_1, \cdots, v_l\}$ with $v_k < v_{i_{k+1}}$
Let current domain be $D_i = \{v_{i_1}, \cdots, v_{i_p}\}$ with $v_{i_k} < v_{i_{k+1}}$ $i_k < i_{k+1}$
Syntax
$D_i.min$:$\{1, \cdots, l\}$
$D_i.max$:$\{1, \cdots, l\}$
as a hash table on key e.
$D_i.succ$:array $[1..l]$ of integers of $\{1..l\}$
$D_i.pred$:array $[1..l]$ of integers of $\{1..l\}$
$D_i.element$:set of couples$(e, index)$ with $e \in D_{i_0}$ and $index \in \{0, \cdots, a\}$,
organized as a hash table
$D_i.value$: array $[1..l]$ of elements of D_{i_0}
$D_i.coordinate$: array $[1..l]$ of *coordinate_str* structure
$D_i.common$: *common_str* structure
Semantics
$D_i.min$: i_1
$D_i.max$: i_p
$D_i.succ[i_k] = i_{k+1}$ $1 \leq k < p$
$D_i.succ[i_k] = + \infty$
$D_i.pred[i_{k+1}] = i_k$ $1 < k \leq p$
$D_i.pred[i_{k+1}] =$ - ∞
$D_i.element(v)$= i_k, if $\exists k$ $v = v_{i_k}$
0, otherwise
$D_i.value[i_k] = v_{i_k}$
$D_i.coordinate[i_k] =$ the pointer to a *coordinate_str* structure
$D_i.common =$ the pointer to a $common_str$ structure

Figure 3.6: Modified domain structure.

Unlike [Liu96], here we only let domain structure share essential information such as

Syntax

$coordinate\_str.indicator$:boolean

Semantics

$coordinate\_str.indicator$: **true** if $v_{i_k} \in D_i$
false otherwise

Figure 3.7: *coordinate* structure.

Sytax

$common\_str.origin$:integer
$common\_str.size$:integer
$common\_str.arcs$:a set of arcs

Semantics

$common\_str.origin$: o if D_i has been oriented.
0 otherwise.
where o is the origin of the functional block to which domain of D_i belongs.
$common\_str.size$: p
$common\_str.arcs$: the set of related arcs of current block

Figure 3.8: *common* structure.

size, *origin*, and *arcs*. The structure *loc* in [Liu96] is also replaced by a more general structure *coordinate* because *loc* heavily depends on the stored order of values of a domain.

3.3.3 Implementation of Consistency Enforcing on Functional Constraints

This section discusses the algorithm for functional constraints (see figure 3.9). The implementation is still based on the traditional arc-consistency techniques as used in AC-3.One minor difference is that we will consider a functional constraint as one arc rather than two directed arcs.For ease of presentation, we merge the three procedures InitialCheck , Remove and Enqueue into one procedure which is still called InitialCheck.

```
Procedure InitialCheck(in $i, j$, inout $Q$)
1begin
2  if $D_j.common.origin = 0$
3     OrientAndCheck($i, j, Q$);
4  else
5     if $D_i.common.origin = 0$
6        OrientAndCheck($j, i, Q$);
7     else
8        ReorientAndCheck($i, j, Q$);
9end
```

Figure 3.9: InitialCheck for FCs.

In addition to removing those inconsistent values, the main task of InitialCheck is to orient all the domains and obtain coordinate of each value of each domain involved in a functional block. There are two different operations in InitialCheck. OrientAndCheck is applied when either D_i or D_j has not been oriented. ReorientAndCheck is applied when both D_i and D_j have been oriented.

The OrientAndCheck(i, j) (see figure 3.10) accomplishes two things: orienting and checking the domain. In OrientAndCheck, the D_j is oriented by D_i which is implemented by sharing the *common* structure with D_j(line 25) (if D_i is not oriented too, it is oriented by itself at line 22 under the artificial identity function) and the coordinates of values of D_j is obtained in line 10. Before the sharing of *common* structure, the related non-functional arcs of variable j should be added to the set of related arcs of the functional block $D_i.common.origin$ (see line 24). Line 6 removes those inconsistent values in D_i. If the domain D_i changes, the set of related arcs of current functional block $D_i.common.origin$ will be enqueued. Lines 13 to 21 check the consistency of arc $< j, i >$ (the constraint C_{ji}).

The ReOrientAndCheck (see figure 3.11) is more complicated than OrientAndCheck.

```
Procedure OrientAndCheck(in $i, j$, inout $Q$)
    begin
1      DELETE $\leftarrow$ false;
2      for each $v \in D_i$
3         if $f_{ij}(v) \notin D_j$
4            begin
5               RemovElem($v, D_i$);
6               DELETE $\leftarrow$ true
7            end
8         else $D_j.coordinate[element(f_{ij}(v)] \leftarrow D_i.coordinate[element(v)]$
9      if DELETE
10        $Q \leftarrow Q \cup D_i.common.arcs$;
11     DELETE $\leftarrow$ false;
12     for each $w \in D_j$
13        if $f_{ji}(w) \notin D_i$
14           begin
15              DELETE $\leftarrow$ true;
16              RemovElem($v, D_j$); // $D_j.element(w) \leftarrow 0$
17           end
18     if DELETE
19        $Q \leftarrow Q \cup D_j.common.arcs$;
20     if $D_i.common.origin = 0$
22        $D_i.common.origin = i$;
22     $D_i.common.arcs \leftarrow D_i.common.arcs \cup D_j.common.arcs$;
23     $D_j.common \leftarrow D_i.common$;
    end
```

Figure 3.10: Algorithm OrientAndCheck.

The problem here is that both D_i and D_j have been oriented.Here, we have two cases.

The first case occurs when both D_i and D_j are oriented by the same reference domain (lines 2 to 16),where it is only necessary to check if the pair of values $(v, w), v \in D_i, w \in D_j$, which have the same coordinate, is consistent under functional constraint C_{ij}. If (v, w) does not satisfy C_{ij}, both v and w will be deleted. The reason is as follows. If for each $v \in D_i$ there does not exist $w' \in D_j$ such that $C_{ij}(v, w')$ holds, v should be deleted and thus w will be deleted automatically because they have the same coordinate. If there exists $w' \in D_j (w' \neq w)$ such that $C_{ij}(v, w')$ holds, v will have the same coordinate with

```
Procedure ReorientAndCheck(in $i, j$, inout $Q$)
  begin
1   if $D_i.common.origin = D_i.common.origin$
2     begin
3       DELETE $\leftarrow$ false
4       for each $v \in D_i$
5         if $f_{ij}(v) \notin D_j$
6           begin
7             RemovElem($\{v\}, D_i$);
8             DELETE $\leftarrow$ true
9           end
10        else
11          if not ($D_i.coordinate[element(v)] = D_j.coordinate[element(v)]$)
12            begin
13              RemovElem($v, D_i$); $//D_i.coordinate[element(v)]- > indicator \leftarrow$ false
14              DELETE $\leftarrow$ true
15            end
16    end
17  else
18    begin
19      if Check(i,j) then $Q \leftarrow Q \cup D_i.common.arcs$;
20      if Check(j,i) then $Q \leftarrow Q \cup D_j.common.arcs$;
21      $D_i.common.arcs \leftarrow D_i.common.arcs \cup < j, i >$
22      $D_j.common.arcs \leftarrow D_j.common.arcs \cup < i, j >$
23    end
  end
Function Check(in $i, j$)
  Begin
    DELETE $\leftarrow$ false;
    for each $v \in D_i$
      if $f_{ij}(v) \notin D_j$
        begin
          RemovElem($\{v\}, D_i$);
          DELETE $\leftarrow$ true
        end
    Return DELETE;
```

Figure 3.11: Algorithm ReorientAndCheck.

w' whose coordinate must be different from w. In other words, the coordinate of v is not unique. So, v should be deleted according to proposition 2 . An interesting point here is that checking is only needed for either D_i or D_j. Because they are oriented by the same reference domain, D_i and D_j have the same size and further if one value in e.g. D_i is deleted, a corresponding value in D_j is automatically deleted and if the value is valid, a corresponding value in D_j will be kept. Note also that D_i and D_j actually share the same structures *coordinate* and *common* and thus there is no need to adjust them.

The second case is that D_i and D_j are oriented by different domains, which is called conflict of orienting. The constraint causing the conflict is called the conflict finder. Acutually, because of the existence of functional constraints C_{ij}, the functional block $D_i.common.origin$ and functional block $D_j.common.origin$ are two part of one functional block. The problem now is that one functional block has two origin ! The direct consequence is that coordinate of value no longer makes sense. One approach is to take these two sub-blocks as seperate parts and thus the constraint C_{ij} will be treated as a general constraint (lines 18 to 23).

As pointed by [Liu96], the conflict of orienting can be avoided in some circumstance. If G' is known a piori, the domain can be oriented in a special order which is the same as the order of traversing a spanning tree of G'. However, in an incremental computing context, such as the constranit solver of CLP, conflict of orienting can not be avoided.

Refer to the section after next section for further discussion on conflict of orienting.

In our fashion to deal with the FC, the RemovElem does not maintain

$$D_i.succ,\ D_i.pred,\ D_i.min,\ D_i.max$$

for all domains in a functional block (however it can maintain the *common.size* for all domains in constant time). Actually, these four attributes are supplied uniquely for implementation of monotonic constraints.The maintenance of these attributes of D_i has to be shifted to InitialCheck and ReCheck for MCs. The algorithm for MCs can be implemented as follows (see figure 3.12). Precisely, only the $D_i.min$ and $D_i.max$ need to be maintained (see lines from 3 to 6) thanks to the special property of MCs.

```
**Procedure** InitialCheck(**in** $i, j$ ,**out**$\triangle$)
  **begin**
1    $\triangle \leftarrow \emptyset$
2    **while** $last(D_i) \notin D_i$
3      $last(D_i) \leftarrow next(last(D_i))$;
4    **while** $last(D_j) \notin D_j$
5      $last(D_j) \leftarrow next(last(D_j))$;
6    $v \leftarrow last\ (D_i)$
7    **while** $v \succ f(last(D_j))$
8      **begin**
90       if $v \in D_i$
10         $\triangle \leftarrow \triangle \cup \{v\}$;
11       $v \leftarrow next\ (v, D_i)$
12     **end**
  **end**
**Procedure** ReCheck(**in** $i, j$ ,**out**$\triangle$)
  **begin**
    InitialCheck($i, j, \triangle$)
  **end**
```

Figure 3.12: Modified algorithm for MCs.

3.3.4 Properties of Algorithms for FCs, AFCs and MCs

Proposition 3 *The algorithm for FC achieves strong n-consistency on G' if there is no conflict of orienting.*

The immediate usefullness of the above property is that given a binary CSP with only functional constraints the InitialCheck actually implicitly obtains all solutions, which can not be achieved in both [VHDT92] and [Liu96]. For example, consider

$$\begin{cases} C_1: & x+y=10 \\ C_2: & y+z=10 \\ C_3: & x+z=10 \\ & x,y,z \in \{1..9\} \end{cases}$$

For the above constraints, both ordinary arc-consistency enforcing algorithm and our special implementation for FCs will terminate with each constraint checked once. After the application of normal consistency enforcing algorithm, no value will be removed from any domain. Assuming the origin of the functional block is the domain of x our implementation works as follows.

For C_1 (each pair connected by the vertical line has the same coordinate)

$$\begin{array}{cccccc} x & 1 & \cdots & 5 & \cdots & 9 \\ & | & \cdots & | & \cdots & | \\ y & 9 & \cdots & 5 & \cdots & 1 \end{array}$$

For C_2,

$$\begin{array}{cccccc} x & 1 & \cdots & 5 & \cdots & 9 \\ & | & \cdots & | & \cdots & | \\ y & 9 & \cdots & 5 & \cdots & 1 \\ & | & \cdots & | & \cdots & | \\ z & 1 & \cdots & 5 & \cdots & 9 \end{array}$$

For C_3, both x and z have been oriented by the same origin. Under the functional constraint C_3, the value $z.1$ has a new coordinate $x.9$ while its old coordinate is $x.1$. So, both $x.9$ and $z.1$ will be removed. After the ReOrientAndCheck of C_3, only value 5 is left in the domain of x,y and z.

Proposition 4 *If it has been involved in an orienting, any functional constraint will need not to be checked in ReCheck.*

Proof

In orienting a domain, this function is employed to obtain the coordinate of the domain. As long as the coordinate is kept for this domain, the functional constraint will be automatically satisfied. In following computation the coordinate either is deleted or kept and is never changed by other constraints as is shown by the algorithm and the example (for example, an attempt by any constraint to change the coordinate of a value will cause the deletion of all values with the same coordinate).$\square$

Corollary 2 *If conflict of orienting can be avoided, all functional constraints need to be checked only once.*

Proposition 5 *The algorithms for FCs, AFCs and MCs has optimal space complexity $\mathcal{O}(e + nd)$ for a priori known 'G'.*

This property is not achieved by [VHDT92] and partially by [Liu96].

Proposition 6 *If there is no conflict of orienting, the time complexity of implementations for FCs, AFCs and MCs is $\mathcal{O}(ed)$ which is optimal.*

Proof:

We only need to prove the **while** loop of AC-main is $\mathcal{O}(ed)$. For ReCheck for MCs (see figure 3.12) we have three loops: first loop (from line 2 to line 3), second loop (from line 4 to 5) and third loop (from 7 to 12). Replace the body of ReCheck to the expansion of AC-main. Using the similar method of analyzing the total complexity of ReCheck in proof of proposition 1 , we obtain that the total complexity of each loop in ReCheck is

$\mathcal{O}(ed)$.Note, in third loop, once v is checked, it will not be checked any more (If $v \in D_i$ v will be deleted. Otherwise, it will be swept out of consideration by the first loop in the next invocation of ReCheck). Similar proof can be obtained for implementations for AFCs. $\square$

This property is achieved by [VHDT92] and partially achieved by [Liu96].

From the proof above, the time complexity will remain the same even if the maintenance of $D_i.succ$ and $D_i.pred$ is done in InitialCheck for MCs. Whether the maintenance should be included depends on practical issues. If the D_i is frequently checked, it may be worth of maintaining them.

3.3.5 Further Issues

On Domain Structure

Actually, the data structures and algorithms presented in previous sections are not the only choice to achieve the results of last section. If other constraints than FC, AFC and MC heavily depends on the attributes,

$$D_i.succ, \quad D_i.pred, \quad D_i.min, \quad D_i.max.$$

for the sake of generality, RemovElem still should be responsible for maintaining the four attributes. In that case, an auxiliary link list, called value link list, whose size is $n'd$, is needed to link all values with the same coordinate (see figure 3.13).

Syntax
  $D_i.link$:array $[1..l]$ of structure $nextv$
Semantics
  $D_i.link[i_k].domain = j$
  $D_i.link[i_k].value = j_k$
    next node (D_j, j_k) after (D_i, i_k) in the value link list.
Syntex
  $coordinate.last$: structure of $nextv$
Semantics
  $coordinate.last.domain = j$
  $coordinate.last.value = j_k$
    the last node (D_j, j_k) of value link list of this coordinate.
Syntex
  $nextv.domain$:integer
  $nextv.value$: integer
Semantics
  $nextv.domain = j$
  $nextv.value = j_k$
    the node is the value j_k of domain D_j, denoted by (j, j_k).

Figure 3.13: Structures required by the value link list.

The newly introduced fields is used by RemovElem and maintained by InitialCheck for FCs. The link list can be maintained in constant time. Specifically, line 8 of OrientAndCheck, will be modified as follows (here we abuse the value v_{i_k} and its hash value i_k):

```
Procedure OrientAndCheck(in $i, j$, inout $Q$)
  begin
      ⋮
3     if $f_{ij}(v) \notin D_j$
        ⋮
8     else
        begin
          $j' \leftarrow D_i.coordinate[v].domain$;
          $i'_k \leftarrow D_i.coordinate[v].value$;
          $D_{j'}.link[i'_k] \leftarrow (j, f_{ij}(v))$;
          $D_i.coordinate[v].domain \leftarrow j$;
          $D_i.coordinate[v].value \leftarrow f_{ij}(v)$;
          $D_j.coordinate[f_{ij}(v)] \leftarrow D_i.coordinate[v]$;
        end
      ⋮
  end
```

ReorientAndCheck can be modified similarly. Once RemovElem assumes the duty to maintain those four attributes the implementations for MCs in [VHDT92] can be adopted without any alteration.

Proposition 7 *Under the above implementation of RemovElem, the time complexity of AC-main is $\mathcal{O}(ed)$ with space complexity $\mathcal{O}(e + nd)$.*

Consider the fact that the complexity of RemovElem is proportional to the number of deleted values. By applying the same principle in proof of AC-main, we have the total time complexity of RemovElem of $\mathcal{O}(n'd + ed)$ where $n' = V(G') \leq e$. Hence force the above result holds.

Conflict of Orienting

When conflict of orienting occurs, the advantage of the method presented in last section is its simplicity. The disadvantage is that the functional block is not exploited to the greatest extent. One extreme is that a functional block may be separated into many small blocks and thus the strongly n-consistency does not make sense for so small functional block. Another disadvantage is that the time complexity to deal with conflict finder C_{ij} will be $\mathcal{O}(d)$.

Proposition 8 *In the occurrence of conflict of orienting, the complexity of the algorithm AC-main is $\mathcal{O}((e' - n'/2)d^2 + (e - e')d)$ where $e' = E(G)$ and $n' = V(G)$.*

Proof:

By expanding the **while** loop of AC-main, for a conflict finder C_{ij}, we have

$$\cdots Check(i,j),\ ...,\ Check(i,j),\ Check(j,i),\ ...,\ Check(j,i),\cdots.$$

The worst case occurs when each deletion of an element of D_j causes an $< i,j >$ entering queue once as shown in the above expansion while every recheck on $< i,j >$ does not delete any element in D_i. Under that case, the total time spent on $< i,j >$ will be $\mathcal{O}(d^2)$(According the algorithm, $< i,j >$ can enter the queue at most d times because only when some element(s) in D_j is eliminated is it possible for $< i,j >$ to enter queue once and an element of D_j will never deleted more than twice). For FC, in the above case, each rechecking of $< i,j >$ will delete at least one value from D_i. The time spent on $< i,j >$ should be $(d+1)d/2$. The number of such arcs are at most $e' - n'/2$, which together with the complexity on one arc $< i,j >$ implies the result. $\square$

If we use the queue element $(< i,j >, w)$ for conflict finder C_{ij}, the algorithm can achieve time complexity $\mathcal{O}(ed)$ with space complexity $\mathcal{O}((e' - n'/2)d + (e - e' + n'/2) + nd)$.

When conflict of orienting can not be avoided, to implement ReorientAndCheck there is an alternative which is to reorient one sub block by the origin of the other block.

The advantage of re-orienting sub-block is that the functional block will not be separated and no functional constraints will be dealt with separately. The disadvantage is that the implementation is more complicated The focus of this approach is how to maintain the *common* and *coordinate* structure in re-orienting. A natural way is to extend the structure of *common* and *coordinate* to link lists. After each access of $D_i.common$ (respectively, $D_i.coordinate[i_k]$), the pointer of $D_i.common$ (respectively, $D_i.coordinate[i_k]$) will be adjusted to the last node of *common* (respectively, $D_i.coordinate[i_k]$) list if it does not point to the last node.For the above approach, we have

Proposition 9 *The complexity of the above mentioned algorithm will be $\mathcal{O}(n' \log n'd + ed)$ with space complexity $\mathcal{O}(e + nd)$.*

Proof:

We consider the **for** loop and **while** loop of AC-main separately. The main concern here is that the membership test can not be implemented in constant time.

It is easy to verify that the length of link list of *common* and *coordinate* is less than or equal to $\log n'$.

Consider **for** loop of AC-main for FCs.D_i is checked d'_i times , the degree of node i in G'.Let λ_k be the length of the *coordinate*(resp. *common*) list at kth InitiCheck on D_i. For λ_k we have

$$\sum_{1}^{dd'_i} \lambda_k \leq d \log n'.$$

Note, when $\lambda_k = 0$, the membership test will take constant time. So, in the following estimation we use $\lambda_k + 1$ to represent the complexity of each membership test. The time consumed on D_i by InitialCheck is

$$\sum_{1}^{dd'_i} (\lambda_k + 1) \leq d \log n' + dd'_i$$

where $n' = |V(G')|$. The total time spent on D_i is

$$\sum_{1}^{n'}(\log n'd + d'_i d) = n' \log n'd + 2e'd$$

where $e' = |E(G')|$. Similarly, we can obtain the complexity of **for** loop in algorithms for AFCs and MCs.

Consider **while** loop of AC-main for MCs. Again we expand **while** loop in terms of ReCheck. As for ReCheck itself, it is enough to consider only one loop, which we called sub-loop. Now, we estimate the time spent on domain D_i by ReCheck in the expansion. Let λ_k be the same as that in the proof of **for** loop. The complexity of kth execution of ReCheck on D_i is as follows

$$\alpha_k = \begin{cases} (\lambda_k + 1)| \triangle_k | & \text{if } | \triangle_k |(> 0) \text{ element(s) has been removed} \\ \lambda_k + 1 & \text{Otherwise} \end{cases}$$

The total number of times of execution of ReCheck on D_i is less than or equal to dd_i where d_i is the degree of node i in G. The total time consumed on D_i by ReCheck is

$$\beta_i = \sum_{1}^{dd_i} \alpha_k \leq \sum_{1}^{dd_i}(\log n'| \triangle_k | + \lambda_k + 1) \leq d \log n' + d \log n' + dd_i.$$

The total time consumed on all domains in G' by ReCheck is

$$\sum_{1}^{n'} \beta_i \leq n' \log n'd + 2ed.$$

Similarly, we can get the complexity of **while** loop of AC-main for AFCs.

So, the complexity of AC-main for FCs, AFCs and MCs is $\mathcal{O}(n' \log n'd + ed)$. The size of the *common* list and *coordinate* list is $\mathcal{O}(n')$ and $\mathcal{O}(n'd)$ respectively because both list can be thought of as a complete tree with n' leaves, and thus the space complexity is $\mathcal{O}(nd + e)$. $\square$

Another approach is to refresh the pointer to *coordinate* and *common* of all domains of block *common.origin* each time some domain of the block is reoriented (Of course, we select to refresh the smaller one between two sub-blocks). For this approach, we have

Proposition 10 *The complexity of this approach is* $\mathcal{O}(n' \log n'd + ed)$.

Proof

The refreshment happens in the check of functional constraints. The worst case occurs when the domains in $V(G')$ are (re)oriented in an order of a complete binary tree. In this case, each re-orienting causes $n'/2$ domains refreshed whose complexity is $n'd/2$. After $\lceil \log n' \rceil - 1$ times re-orienting, no conflict of orienting will happen. The total complexity of refreshment is $n' \log n'd/2$. So, the complexity of the algorithm is $\mathcal{O}(n' \log n'd + ed)$.$\square$

In summary, for space complexity $\mathcal{O}(e + nd)$

1. if there is no conflict of orienting time complexity of $\mathcal{O}(ed)$ can be achieved .

2. when there is conflict of orienting

    (a) $\mathcal{O}((e' - n'/2)d^2 + (e - e')d)$ can be achieved with conflict finder treated as general constraint.

    (b) $\mathcal{O}(n' \log n'd + ed)$ can be achieved so that n-consistency can be achieved.

For space complexity $\mathcal{O}(ed + nd)$, we have the following conjecture.

Conjecture 1 *For functional constraints, there exists an algorithm with space complexity* $\mathcal{O}(ed + nd)$ *and time complexity* $\mathcal{O}(ed)$ *such that* n*-consistency can be achieved when conflict of orienting occurs.*

Chapter 4

Bound Consistency over n-ary Linear Constraints

4.1 Introduction

In last chapter, we are mainly concerned with binary CSPs. Actually, in most constraint programming languages, a class of n-ary constraints,specially linear equation and linear in-equation, can be easily expressed [DVHS88,ILOG].The modeling (expressive) ability and usefulness of this class of constraints have been shown by the great success of (integer) Linear Programming [Dan63] [NW88] in practice. These constraints have been fully studied theoretically and practically in the community of (integer) linear programming. Unfortunately, those constraint solving techniques can not be directly used in the constraint solver of a constraint programming language because the constraint programming language normally has to face a more general setting where linear constraints are not the only constraints to be of concern and other requirements imposed on constraint solving by the language ([JM94], [VHD91] and [vH89]). Because of the general purpose property of a language, underlying most constraint programming languages over finite domain is the CSP model ([vH89] and [ILOG]). As is shown in previous chapter, general techniques(e.g. arc consistency) in CSP can be improved by considering special properties of constraints (the case for FC, AFC and MC). The problem now is how to efficiently deal with the n-ary linear constraint in a constraint programming language.

Some work has been done in CSP to attack general n-ary constraints. A general method is given by [RPD89] to convert a CSP with n-ary constraints to a binary one. From the point of view of consistency, [MM88] generalized the arc consistency [Mac77] of binary CSP and proposed an algorithm GAC-4 whose counterpart in binary CSP is AC-4 [MH86]. In GAC-4, a constraint is regarded as a set of explicitly known tuples.However, for linear constraints,the tuples are not trivially explicit. Even they can be obtained, the size of the set of tuples will be too large to be useful because an n-ary linear constraint is generally weak. That is why GAC-4 is not practical to be used in the solver of a general language. An important theoretical work in CSP is to study the relationship between local consistency and global consistency([Fre82] [Fre90] [Dec92] [DP92]). [DB95] presents a new definition of consistency on n-ary constraints under which the results for binary CSP can be easily generalized to n-ary CSP. However, the new definition and related algorithms mainly serve as conceptual tools. Algorithms presented in [DB95] are not practical to deal with linear constraints.

Actually, linear constraints have been dealt with in [Lau78], [vH89], [CC93] and [Cod96] by propagating bounds of domains of variables. The essence of the idea is to relax the traditional arc consistency to bound consistency(see next section). As for bounds of variables, the directly related area is interval constraint logic programming(ICLP) ([Cle87], [OV90], [OV93], [OB93], [BHM94], [BO97], and [VHMD97]) where the interval (lower and upper bounds) of a variable is the basic operand. The strength of ICLP comes from interval arithmetic [Moo63].The theme of ICLP is to generalize functional interval arithmetic to relational interval arithmetic and consistency techniques in CSP. Because constraints of interest in ICLP are so general that although CLP(BNR) ([BO97]) can deal with integer constraints, no special attention is paid to linear constraints. [Lho93] defines a special class of CSP,which is called numerical CSP, where the interval arithmetic is employed as the foundation of problem solving and consistency techniques similar to ICLP are defined. Like ICLP,the domain of variable in numerical CSP is continuous. We also note the treatment of real linear constraint in CIAL ([CL94]) where a generalized Gaussian elimination method is employed to speed up the solving of linear equation systems.

The topic of this chapter is motivated by how to efficiently deal with linear constraints over finite domains in the context of constraint programming.

4.2 Bound Consistency

4.2.1 Basics of Interval Arithmetic

We use the usual mathematical notations to represent intervals. In interval arithmetic, a variable always takes an interval as its value. For convenience to present, we introduce two kinds of notations for the bounds of the domain of a variable.

The first is

$$[x] = [Lb, Ub] \tag{4.1}$$

where Lb and Ub is the lower bound and upper bound of the domain of x. We use $Lb(x)$ and $Ub(x)$ to denote the lower bound and upper bound of x respectively.The interval arithmetic operations imposed on $[x]$ are defined as follows[Moo63]:

$$[x] + [y] = [Lb(x) + Lb(y), Ub(x) + Ub(y)],$$

$$[x] - [y] = [Lb(x) - Ub(y), Ub(x) - Lb(y)],$$

$$[x] - a = [Lb(x) - a, Ub(x) - a],$$

$$a[x] = \begin{cases} [aLb(x), aUb(x)], a > 0 \\ [aUb(x), aLb(x)], a < 0 \end{cases}.$$

Additionally,

$$[x] \cap [y] = \begin{bmatrix} max(Lb(x), Lb(y)) \\ min(Ub(x), Ub(y)) \end{bmatrix}.$$

The other notation is the vector form of the bounds of a variable:

$$\langle x \rangle = \begin{pmatrix} Lb \\ Ub \end{pmatrix} \tag{4.2}$$

all the operations on $\langle x \rangle$ will be considered as vector ones.For example,

$$\langle x \rangle \pm \langle y \rangle = \begin{pmatrix} Lb(x) \pm Lb(y) \\ Ub(x) \pm Ub(y) \end{pmatrix}$$

We also abuse operator $[\ ]$ and $\langle\ \rangle$. Applying $[]$ to $\langle\rangle$ means to change the vector to an interval and vice versa.

4.2.2 Bound Consistency on An n-ary Linear Constraint System

In this section we give the definitions of bound consistency on n-ary linear constraint over finite domains.

Definition 15 A linear constraint $cs(x_1, \cdots, x_n)$ *is*

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \diamond\ b$$

$$x_i, a_i, b \in Z \qquad \diamond \in \{=, \leq, \geq\}.$$

We use $vars(cs)$ and $|cs|$ to denote respectively the set and the number of variables that occur in cs.

Definition 16 *A* linear constraint system *is a triple* (V, D, C) *where* $V = \{x_1,\ x_2,\ \cdots,\ x_m\}$ *denotes a set of variables,* $D = \{D_1, D_2, \cdots, D_m\}$ *a set of domains,*D_i *being the finite integer domain of* x_i*, and* $C = \{cs_1, cs_2, \cdots, cs_e\}$ *a set of constraints,* cs_i *being linear constraint.*

Hereafter, m, e, n and d refer to the number of variables,number of constraints, $max\{|cs_i|\ ,\ cs_i \in C\}$ (the maximum number of variables in any constraints) , and $max\{|D_i|\ ,\ D_i \in D\}$ (the maximum domain size) respectively.

Definition 17 *A* Z-interval *is*

$$[a, b] = \{r \in R | a \leq r \leq b, a, b \in Z\}.$$

$\tilde{x} = ([x_1], \cdots, [x_n])$ *is a* Z-interval vector *if each* $[x_i]$ *is a* Z*-interval.*

Let $\mathcal{Z}$ be the set of all Z-intervals on which $\leq$ is defined as $\subseteq$. The arithmetic operations on Z-interval is the same as those in last section.

Definition 18 *The* Z-interval representation *of a set* $S \in R$ *is*

$$\Box S = max\{[a, b] \in \mathcal{Z} | [a, b] \subseteq S\}.$$

Definition 19 *The* projection function π_i *of a constraint cs on* x_i *is*

$$\pi_i(cs) = \frac{-1}{a_i}(a_1x_1 + \cdots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \cdots + a_nx_n - b).$$

The natural interval extension of $\pi_i(cs)$ *is*

$$\Pi_i(cs) = \frac{-1}{a_i}[a_1[x_1] + \cdots + a_{i-1}[x_{i-1}] + a_{i+1}[x_{i+1}] + \cdots + a_n[x_n] - b].$$

Definition 20 *The* projection *of a constraint cs on variable* x_i *is a set*

$$\{v_i \in R \mid \exists v_k \in [x_k], k \neq i \ \ \textit{such that} \ \ cs(v_1, \cdots, v_n) \ \ \textit{holds}\}.$$

The projection of cs on variable x_i is exactly

$$Proj_i(cs) = \begin{cases} \Pi_i(cs) & \text{if } \diamond' \text{ is } = \\ [-\infty, Ub(\Pi_i(cs))] & \text{if } \diamond' \text{ is } \leq \\ [Lb(\Pi_i(cs)), +\infty] & \text{if } \diamond' \text{ is } \geq \end{cases}$$

where

$$\diamond' = \begin{cases} \leq & \text{if } a_i \text{ is negative and } \diamond \text{ is } \geq \\ \geq & \text{if } a_i \text{ is negative and } \diamond \text{ is } \leq \\ \diamond & \text{otherwise} \end{cases}$$

Definition 21 *A constraint cs is* balanced *with respect to* $([x_1], \cdots, [x_m])$ *iff*

$$\forall x_i \in vars(cs) \ [x_i] \subseteq Proj_i(cs).$$

A constraint system is balanced *with respect to* $([x_1], \cdots, [x_m])$ *iff every* $cs_j \in C$ *is balanced.*

Note. In the above definition, $[x_i]$ may be a real interval.

Definition 22 *A constraint cs is* bound consistent *with respect to Z-interval vector* $([x_1], \cdots, [x_m])$ *iff* $\forall x_i \in \text{vars}(cs) \;\; [x_i] \subseteq \Box Proj_i(cs_j)$.

A constraint system is bound consistent *with respect to Z-interval vector* $([x_1], \cdots, [x_m])$ *iff every* $cs_j \in C$ *is bound consistent.*

Property 3 *Any linear constraint cs with initial domain* $([x_1], \cdots, [x_m])$ *is balanced with respect to*

$$([x_1] \cap Proj_1(cs), \cdots, [x_m] \cap Proj_m(cs)).$$

This can be easily proved by the intermediate value theorem. Note,this property remains true in ICLP because approximation always rounds outward [BO97]. However,the above property is not applicable to bound consistency because of the requirement for the Z-interval representation of the projection of a constraint.In this case, some real value are removed by Z-interval representation, which may make the bound of some other variable invalid and thus breaks the consistency of cs. An example is

$$3x = 4y$$
$$[x] = [y] = [1..10].$$

Because of the Z-interval representation of domain of variable, bound consistency can not be described by the concept of *narrowing function* as defined in [BO97]. By the way, according to the *idempotence* property of narrowing function, the narrowing algorithm in [BO97] can be improved as follows. Let $\overrightarrow{\rho}$ be the narrowing function defined by a constraint ρ. When $[x_i]$ in $vars(\rho)$ is changed by $\overrightarrow{\rho}$, the constraint ρ need not to be rechecked. If the narrowing algorithm remains untouched, the *narrowing function* can be generalized to a projection function together with any rounding strategy(inward or outward).

Definition 23 *Given a constraint system (V, D, C), a Z-interval vector $\tilde{x} = ([x_1], \cdots, [x_n])$, where $[x_i] \subseteq \Box D_i$, is a* fixed point *of the system if the constraint system is bound consistent with respect to $\tilde{x}$. $\tilde{x} \geq \tilde{y}$ if and only if $\forall i \in 1..n \quad [x_i] \geq [y_i]$*

Note.When any $[x_i]$ is empty, the fixed point $\tilde{x}$ is regarded as an empty vector. Any two fixed point of a system (V, D, C) is comparable.

4.2.3 Bound Consistency Algorithms and Their Complexities

There are two approaches to enforce bound consistency on a constraint system (V, D, C). One approach, as is shown in algorithm BC-1(see figure 4.1), is a natural extension of AC-3 [Mac77]. BC-1 uses a generalized queue element structure of AC-3. The queue element (x_i, x_j) of AC-3 brings the information that the domain of x_i will be refined by the binary constraint between x_i and x_j. As for bound consistency, the bounds of variable x_i is refined by an n-ary constraint cs_j where x_i occurs. So, in algorithm BC-1, the queue element is (x_i, cs_j). The other approach,as is shown in algorithm BC-2(see figure 4.2), uses a new queue element structure. In algorithm BC-2, the queue contains constraints on which bound consistency will be enforced .Therefore the Revise has been modified to treat a constraint cs_j as a whole. The Revise here can be thought of as a relaxation of bound consistency (see property 3). Both approaches make use of the property of linear constraint that the valid interval of variables in a constraint cs_j can be easily and efficiently obtained by $\Pi_i(cs_j)$.

By the definition of bound consistency it is easy to prove the following two propositions .

Proposition 11 *Both algorithms BC-1 and BC-2 are correct.*

```
Algorithm BC-1
begin
  $Q \leftarrow \{< x_i, cs_j > | \forall cs_j \in C, \forall x_i \in vars(cs_j)\}$;
  while(Q not empty)
  begin
    select and delete $< x_i, cs_j >$ from $Q$;
    if Revise($< x_i, cs_j >$)
      $Q \leftarrow Q \cup \{< x_l, cs_k > | \forall l, cs_k \;\; x_l, x_i \in vars(cs_k), l \neq i\}$
  end
end
function Revise($< x_i, cs_j >$)
begin
  if not ($[x_i] \subseteq \Box Proj_i(cs_j)$)
    begin
      $[x_i] \leftarrow [x_i] \cap \Box Proj_i(cs_j)$;
      return true
    end
  else return false
end
```

Figure 4.1: Algorithm BC-1.

Proposition 12 *Both algorithms always reach the maximal fixed point of constraint system (V, D, C).*

Proposition 13 *For the constraint system (V, D, C), the worst case complexity of algorithm BC-1 is $\mathcal{O}(n^3ed)$.*

Proof:

The total number of iterations of while() loop depends on the number of pairs that have ever entered the queue. Two cases contribute to the growth of Q. One is the initialization of Q and the other is the success of Revise(), that is Revise() returns `true`.

For initialization, Q has at most ne pairs because each constraint produces at most n pairs and there is totally e constraints in the system.

Now consider maximum number of success of the function Revise.Each success of

Algorithm BC-2

```
begin
  $Q \leftarrow \{cs_i | cs_i \in C\}$;
  while ($Q$ not empty)
  begin
    select and delete $cs_i$ from $Q$;
    Revise($cs_i, Q$);
  end
end
procedure Revise($cs_j, Q$)
begin
  for each $x_i \in vars\ (cs_j)$
  begin
    if $[x_i] \not\subseteq \Box Proj_i(cs_j)$
      begin
        $[x_i] \leftarrow [x_i] \cap \Box Proj_i(cs_j)$;
        $Q \leftarrow \{cs_k \in C \mid x_i \in vars\ (cs_k)\}$
      end
  end
end
```

Figure 4.2: Algorithm BC-2.

Revise at least reduces the size of domain of some variable x_i by 1. So, at most Revise will succeed md times. For each variable x_i, if its domain is revised, we assume there will be d_i pairs entering Q. Therefore, there are at most $\sum_{i=1}^{md} d_i$ pairs ever entering Q.

Consider the graph $G = (N, E)$ of (V, D, C) which is defined as

$$N = V$$

$$E = \{(x_i, x_j)_k \mid \exists k \;\; x_i, x_j \in vars(cs_k)\}.$$

It is observed that the d_i is exactly the degree of node i of G and the number of edges of G is at most n^2e. Therefore

$$\sum_{i=1}^{md} d_i \leq 2n^2ed.$$

The complexity of Revise procedure is obviously linear of n because of $Proj_i(cs_j)$.

Hence forth the complexity of BC-1 is $\mathcal{O}(n^3ed)$. $\square$

The advantage of Algorithm BC-2 is that when the constraint cs_j is treated as a whole, much time can be saved by fully exploiting the property of the constraint.

Proposition 14 *The Revise procedure of BC-2 can be implemented in linear time of* n.

Proof:

Consider cs_j

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n \diamond b_j.$$

Let

$$f_j(x) = a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n - b_j$$

where

$$x = (x_1, \cdots, x_n)$$

and its natural interval extension be $F_j(X)$ where $X = (X_1, \cdots, X_n)$ and each X_i is interval variable.

Now, we have

$$\Pi_i(cs_j) = -\frac{1}{a_{ji}}[\langle F_j([X])\rangle - \langle a_{ji}[x_i]\rangle]$$

where

$$[X] = ([x_1], \cdots, [x_n]).$$

Recall

$$Proj_i(cs_j) = \begin{cases} \Pi_i(cs_j) & \text{if } \diamond' \text{ is } = \\ [-\infty, Ub(\Pi_i(cs_j))] & \text{if } \diamond' \text{ is } \leq \\ [Lb(\Pi_i(cs_j)), +\infty] & \text{if } \diamond' \text{ is } \geq \end{cases}$$

Therefore, Revise(cs_i) can be implemented in linear time of n.$\square$

The use of interval and vector greatly simplifies the above proof. A motivating illustration is as follows. Consider equation

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b.$$

For the ease of presentation, All coefficients are classified into a positive set, $\{a_1^+, ..., a_k^+\}$, and a negtive set,$\{a_{k+1}^-, \cdots, a_n^-\}$.

Let

$$\Delta_{max} = \sum_{j=1}^{j=k} a_j^+ Ub(x_j^+) + \sum_{j=k+1}^{j=n} a_j^- Lb(x_j^-) - b,$$

$$\Delta_{min} = \sum_{j=1}^{j=k} a_j^+ Lb(x_j^+) + \sum_{j=k+1}^{j=n} a_j^- Ub(x_j^-) - b.$$

For x_i^+, the least value of the projection function of cs on x_i^+ is

$$I_{low} = \frac{\Delta_{max}}{a_i^+} - Ub(x_i^+),$$

and the greatest value is

$$I_{upper} = \frac{\Delta_{min}}{a_i^+} - Lb(x_i^+).$$

The new bounds of x_i^+ should be

$$[x_i^+] = [x_i^+] \cap \square[I_{low}, I_{upper}].$$

Similarly, for x_i^-, we have

$$I_{lower} = \frac{\Delta_{min}}{-a_i^-} + Lb(x_i^-)$$

$$I_{upper} = \frac{\Delta_{max}}{-a_i^-} + Ub(x_i^-)$$

$$[x_i^-] = [x_i^-] \cap \Box[I_{low}, I_{upper}].$$

Proposition 15 *Given a constraint system (V, D, C), the complexity of algorithm BC-2 is $\mathcal{O}(mned)$*

Proof:

As in proposition 13, we need to estimate the maximal length of the Q. The Revise(cs_i) may success at most md times and each success will cause at most e constraints to enter Q. By proposition 14,the complexity of Revise(cs_i) is $\mathcal{O}(n)$ and thus complexity of BC-2 is $\mathcal{O}(emnd)$. $\Box$

4.2.4 An Improved Version of BC-1

As implied by proposition 14, revising together the bounds of all variables in one constraint is more efficient than revising them separately. The idea behind proposition 14 can be employed to improve the efficiency of BC-1.

For system (V, D, C), consider constraint $cs_j \in C$:

$$a_{j1}x_1 + \cdots + a_{jn}x_n = b_j.(1)$$

In BC-2, $F_j([X])$ has to be evaluated each time. Here, we introduce an auxiliary variable y_j for cs_j and x_{ji} for variable x_i so that the function Revise() of BC-1 can be implemented in constant time. The modified algorithm BC-1 is listed in figure 4.3.

In order to emphasize that cs_j is no longer used in Revise, we use the queue elment

```
Algorithm BC-1.1
begin
1 for each $cs_j \in C$
2    begin
3       $[y_j] \leftarrow F_j(\Box D_1, \cdots, \Box D_m)$;
4       for each $i \in vars(cs_j)$
5          begin
6             $[x_{ji}] \leftarrow \Box D_i$;
7             $[x_i] \leftarrow \Box D_i$;
8          end
9    end
10 $Q \leftarrow \{< x_i, j > | \forall cs_j \in C, \forall x_i \in vars(cs_j)\}$;
11 while(Q not empty)
12 begin
13    select and delete $< x_i, j >$ from $Q$;
14    Revise($< x_i, j >$,$Q$)
15 end
end
function Revise($< x_i, j >$)
begin
16 $\Pi_i(cs_j) \leftarrow -\frac{1}{a_{ji}}[\langle y_j \rangle - \langle a_{ji}[x_{ji}] \rangle]$
17 if not ($[x_i] \subseteq \Box Proj_i(cs_j)$)
18    begin
19       $[x_i] \leftarrow [x_i] \cap \Box Proj_i(cs_j)$;
20       if $\Box\Pi \neq \Pi$
21          $Q \leftarrow Q \cup \{< x_l, k > | \forall l, k \;\; x_l, x_i \in vars(cs_k), l \neq i\}$
22       else
23          $Q \leftarrow Q \cup \{< x_l, k > | \forall l, k \;\; x_l, x_i \in vars(cs_k), l \neq i \;\; k \neq j\}$
24       for all $k$ such that $x_i \in vars(cs_k)$
25          begin
26             $[y_k] \leftarrow [\langle y_k \rangle - \langle a_i[x_{ki}] \rangle] + a_i[x_i]$;
27             $[x_{ki}] \leftarrow [x_i]$
28          end
29    end
end
```

Figure 4.3: Algorithm BC-1.1.

(x_i, j). Lines 1-9 intialize y_j's and x_{ji}'s. Lines 24-28 modify corresponding y_j (that is $F_j([X])$) whenever $[x_i]$ changes. Line 16 makes use of y_j to obtain the projection of cs_j on x_i. The invariant that

$$[y_j] = F_j([x_{j1}], \cdots, [x_{jn}])$$

is preserved in the algorithm, which guarantee the correctness of line 16. Lines 20-23 enqueue all the affected pairs $< x_l, k >$ as a result of the change of $[x_i]$. Line 20 checks whether there is rounding on the projection. If there is no rounding, cs_j remains consistent (property 3) and thus no pair $< x_i, j >$ needs to be considered again. This idea is also applicable to refine the Revise in BC-2.

Proposition 16 *For system (V, D, C), the complexity of algorithm BC-1.1 is $\mathcal{O}(n^2ed)$.*

Proof:

From the proof of proposition 13,the complexity of the above algorithm is $\mathcal{O}(n^2ed)$. The total complexity of **for** loop (lines 24-28) can be proved by similar method used to prove the total complexity of Enqueue in the last chapter.$\square$

Example.

Consider constraint

$$2x_1 + 5x_2 + x_3 = 5,$$

$$[x_1] = [x_2] = [x_3] = [-10, 10].$$

Evaluate $[y_1]$ first

$$[y_1] = \begin{bmatrix} -20 \\ \\ 20 \end{bmatrix} + \begin{bmatrix} -50 \\ \\ 50 \end{bmatrix} + \begin{bmatrix} -10 \\ \\ 10 \end{bmatrix} - 5 = \begin{bmatrix} -85 \\ \\ 75 \end{bmatrix},$$

$$[x_{11}] = [x_{12}] = [x_{12}] = [-10, 10].$$

$$[x_1] = -\frac{1}{2}[\langle y_1 \rangle - \langle 2[x_{11}] \rangle] \cap [-10, 10] = [-10, 10],$$

$$[x_2] = -\frac{1}{5}[\langle y_1\rangle - \langle 5[x_{11}]\rangle] \cap [-10, 10] = [-7, 5],$$

Actually, because no rounding occurs here, there is no pair entering Q.

$$[x_3] = -\left[\begin{pmatrix} -75 \\ 80 \end{pmatrix} - \begin{pmatrix} -10 \\ 10 \end{pmatrix}\right] \cap [-10, 10] = [-10, 10].$$

□

4.2.5 On Simple Transformation of Constraint System

It is interesting to study the propagation on a simple equivalent system of linear system (V, D, C). In CLP(FD) ([CD96]), an n-ary constraint is translated into 3-ary constraints.

The constraint cs_j

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n = b_j$$

can be translated into the following system cs'_j by introducing intermediate variables $\{y_{j1}, \cdots, y_{jn-2}\}$ [CC93]:

$$\begin{array}{lcll} a_{j1}x_1 & + & a_{j2}x_2 & = y_{j1} \\ y_{j1} & + & a_{j3}x_3 & = y_{j2} \\ & \vdots & & \\ y_{jn-2} & + & a_{jn}x_n & = b_j \end{array}$$

Let the system obtained from (V, D, C) by the above rule be

$$(V', D', C')$$

where

$$V' = V \cup \{y_{ij} | i \in 1..e, j \in 1..|vars(cs_i)| - 2\}$$

$$D' = D \cup \{D_{y_{ij}} | i \in 1..e, j \in 1..|vars(cs_i)| - 2\}$$

$$C' = \{cs'_i | i \in 1..e\}$$

Proposition 17 *The maximal fixed point of system (V', D', C') is the same as that of (V, D, C).*

Proof:

Assume the fixed points of (V', D', C') and (V, D, C) are $([x'_1]$, $\cdots$, $[x'_n]$, $[y_{11}]$, $\cdots$, $[y_{nn-2}])$ and $([x_1]$, $\cdots$, $[x_n])$ respectively.

For any $\bar{x}_i \in [x_i]$ and any constraint cs_j in which x_i appears, there exists $\bar{x}_1$, $\cdots$, $\bar{x}_{i-1}$, $\bar{x}_{i-1}$, $\cdots$, $\bar{x}_n$ such that cs_j is satisfied,that is

$$a_{j1}\bar{x}_1 + \cdots + a_{i-1}\bar{x}_{i-1} + a_{ji-1}x_{ji-1} + \cdots + a_{in}\bar{x}_n = b_j.$$

Obviously, $\exists \bar{y}_{j1}, \cdots, \bar{y}_{jn-2}$ such that each constraint of cs'_j is satisfied and thus $\bar{x}_i \in [x'_j]$ and $\bar{y_{jl}} \in [y_{jl}]$ $(l : 1..n-1)$ (because of the continuity of the constraints) . So, $[x_i] \subseteq [x'_i]$ for any i.

For any $\bar{x}_i \in [x'_i]$ and any cs'_j that contains x_i, there exists $\bar{y}_{ji-2}$ and $\bar{y}_{ji-1}$ such that the equation $\bar{y}_{ji-2} + a_{ji}\bar{x}_i = \bar{y}_{ji-1}$ holds. Because cs'_j is bound consistent, there exist

$\bar{x}_1, \cdots, \bar{x}_{i-1}, \bar{x}_{i+1}, \cdots, \bar{x}_n$ such that $\bar{x}_l \in [x'_l]$ $l \in \{1..n\}$ and

$$\begin{aligned}
a_{j1}\bar{x}_1 &+ a_{j2}\bar{x}_2 = \bar{y}_{j1} \\
&\vdots \\
\bar{y}_{ji-3} &+ a_{ji-2}\bar{x}_{i-1} = \bar{y}_{ji-2} \\
\bar{y}_{ji-2} &+ a_{ji-1}\bar{x}_i = \bar{y}_{ji-1} \\
\bar{y}_{ji-1} &+ a_{ji}\bar{x}_{i+1} = \bar{y}_{ji} \\
&\vdots \\
\bar{y}_{jn-2} &+ a_{jn}\bar{x}_n = b
\end{aligned}$$

Therefore

$$\sum_{i=1}^{n} a_{ji}\bar{x}_i = b_j$$

and thus $([x'_1], \cdots, [x'_n])$ is a fixed point of (V, D, C), which implies

$$([x'_1], \cdots, [x'_n]) \leq ([x_1], \cdots, [x_n]).$$

So, the two systems have the same fixed point.□

Proposition 18 *Bound consistency can be enforced on (V', D', C') by Algorithm BC-1 in $\mathcal{O}(ned')$ where*

$$\begin{aligned}
d' &= \max_j \min \left\{ \textstyle\sum_{i=1}^{(l-2)/2} |a_{ji}|d, \quad \sum_{i=l/2}^{l-2} |a_{ji}|d \right\}, \\
l &= |vars(cs_j)|.
\end{aligned}$$

Proof:

The number of the constraints in C' will be ne, and each constraint has at most 3 variables. By proposition 13, the complexity is $\mathcal{O}(ned')$.From the translation rule, it is easy to obtain the size of the greatest domain in the new system. □

The proposition shows that d' is related to n and the magnitude of the coefficients in C. However, the introduction of intermediate variables simplify the relationship between variables in one constraints, which helps make the estimation of time complexity more accurate. For example, in constraint cs_j, x_1 is related to all the other variables. The auxiliary variable y_{i1} separate x_1 and x_2 from $x_3, \cdots, x_n$.

4.2.6 Linear Equation System

Definition 24 *A linear constraint system (V, D, C) is called a* linear equation system *if C constraints only linear equations.*

Definition 25 *Given a linear equation system (V, D, C), C is said in* solved form *if it is of the form*

$$bx_B = ax_N + c$$

where a, b, c are constant vectors and x_B, x_N are variable vectors such that

$$x_B \cup x_N = V$$
$$x_B \cap x_N = \emptyset$$

The variable in x_B is called subject variable *while the variable in x_F is called* free variable.

For a linear equation system (V, D, C), we can always obtain an equation system (V, D, C') where C' is in solved form by Gaussian-Jordan elimination. For the new system we have following conclusion which is a direct corollary of proposition 16 and proposition 15.

Corollary 3 *Algorithm BC-1 can achieve bound consistency on (V, D, C') in $\mathcal{O}(n'^2(m-e)d)$, and algorithm BC-2 can achieve bound consistency on (V, D, C') in $\mathcal{O}(e(m-e)n'd)$,* *where*

$$n' = \max_j |vars(cs'_j)|$$

.

From the above result we can see that the efficiency of consistency enforcing algorithm on new system may not be better than on the old system. For example, let (V, D, C) be

$$x_1 = x_2 + x_3 + x_4$$
$$x_5 = x_1$$
$$x_1, ..., x_4 \in [-10, 10]$$
$$x_5 \in [-20, 20].$$

One solved form of C is

$$x_1 = x_2 + x_3 + x_4$$
$$x_5 = x_2 + x_3 + x_4$$

Here, the consistency enforcing algorithm will take longer time on C' than on C. Furthermore, the maximal fixed point of (V, D, C') is greater than that of (V, D, C).

4.2.7 Issues on Constraint Solver of a CLP over Finite Domain

Naturally,for linear constraints,the *infer* function of the constraint solver of a CLP language can be implemented as bound consistency enforcing. Actually, that is what CLP(FD) ([CD96]) does with linear constraints although in a different implementation fashion. Naturally, bound consistency can be generalized to (integer) nonlinear constraints where the projection function can be numerically calculated by Newton methods(see [VHMD97]).

4.3 Bound Consistency on Binary Equation System

Binary linear constraint system is an important case in finite domain constraint programming [DVHS88]. Efforts such as [VHDT92] have been made to improve the efficiency of consistency enforcing algorithm on binary system. Here, we focus on binary equation system.

A binary equation system may have many equivalent systems which have the same solution set. It is desirable to find a system on which an efficient consistency algorithm can be constructed and a smaller fixed point can be achieved.

In this section we will study the impact of algebraic manipulation,especially the Gaussian-Jordan elimination, on bound consistency on binary equation system.

4.3.1 The Euclidean Algorithm

In this section we will recall some necessary materials in solving an integer equation. The Euclidean Algorithm plays an important role in a class of methods. For the sake of self containment, the algorithm and some useful results are given below.Refer to [NW88] for detailed discussion.

The Euclidean algorithm is to find the greatest common divisor (gcd) of two integers.

```
Algorithm gcd [NW88]
**function** gcd(**in** $a, b$)
Precondition: integers $a \geq b > 0$
  **begin**
    $(c_{-1}, c_0) = (a, b); (p_{-1}, p_0) = (1, 0); (q-1, q_0) = (0, 1);$
    $t \leftarrow 0;$
    do{
        $t \leftarrow t + 1$
        $d_t = \lfloor \frac{c_{t-2}}{c_{t-1}} \rfloor$
        $c_t = c_{t-2} - d_t c_{t-1}$
        $p_t = p_{t-2} + d_t p_{t-1}$
        $q_t = q_{t-2} + d_t q_{t-1}$
    }while $(c_t != 0)$;
    $T \leftarrow t$ ;
    **return** $c_{T-1}$
**end**
```

Proposition 19 *[NW88] The Euclidean algorithm is correct and*

$$c_t = (-1)^{t+1}(p_t a - q_t b) \quad for \quad t = -1, 0, ..., T.$$

Proposition 20 *[NW88] The Euclidean algorithm runs in polynomial time $O(\log a)$.*

Now consider the solution of the following equation

$$ax + by = c \tag{4.3}$$

where x,y, a, b, c are integers.

Proposition 21 *[NW88] Let r = gcd(a,b) with r=pa-qb, where p and q are relatively prime.Equation 4.3 has a solution if and only if $r|c$. If $r|c$ the general solution can be described by*

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{c}{r} \begin{pmatrix} p \\ -q \end{pmatrix} + \frac{z}{r} \begin{pmatrix} b \\ -a \end{pmatrix}, z \in Z^1 \tag{4.4}$$

r, p and q can be obtained from the Euclidean algorithm.

4.3.2 Bound Consistency on Solved Form

As is pointed in last section, the complexity of bound consistency enforcing on a single equation may be related to the size of the domain of variable. Reconsider the example given in last section

$$3x = 4y, [x] = [y] = [0, 10].$$

To make the system bound consistent we have the following iteration sequences

$$[x]_1 = [0, 10], [y]_1 = [0, 7], [x]_2 = [0, 9], [y]_2 = [0, 6], [x]_3 = [0, 8], [y]_3 = [0, 6].$$

The general solution 4.4 can be employed to improve the efficiency in bound consistency, for both equation and in-equation.

Proposition 22 *Given a constraint $ax + by \diamond c$, it can be made bound consistent in $\mathcal{O}(\log a)$.*

Proof:

The bound consistency can be achieved by two steps.

First we obtain the projection on x and y by projection function

$$\begin{cases} [x] = \Box(\frac{c-b[y]}{a}) \cap [x] \\ [y] = \Box(\frac{c-a[x]}{b}) \cap [y] \end{cases}$$

Next step is to adjust the changed bound of x and y such that no more iteration is needed.

Assume in the first step,the lower bound of x has been changed to $Lb'(x)$. By proposition 21 x should satisfy

$$\min x, x = \frac{c}{r}p + bz.$$

So the final lower bound of x is exactly

$$\min x = \frac{c}{r}p + bz' \tag{4.5}$$

where

$$z' = \lfloor \frac{Lb'(x) - \frac{c}{r}p}{b} \rfloor.$$

Similarly, if upper bound of x has been changed to $Ub'(x)$, it should be adjusted to

$$\max x = \frac{c}{r}p + b\lceil \frac{Ub'(x) - \frac{c}{r}p}{b} \rceil \quad \square$$

Example

$$4x + 3y = 7$$

$$[x] = [y] = [-20, 20]$$

By naive method, the iteration sequences are

$$[x]_1 = [-13, 16], [y]_1 = [-19, 18], [x]_2 = [-11, 16], [y]_2 = [-19, 18].$$

With the help of Euclidean algorithm, we have

$$a = 4, b = 3, c = 7, r = 1, p = 1, q = 1$$

$$[x]_1 = [-13, 16].$$

By 4.5

$$[x] = [-11, 16],$$

and according to x and projection function we have

$$[y] = [-19, 18].$$

Of course, we may have iteration sequence

$$[y]_1 = [-19, 18] \to [x]_1 = [-13, 16]$$

□

Proposition 23 *The binary equation system (V, D, C) with C in solved form can be balanced in $\mathcal{O}(e)$.*

Proof:

In this proof the resulting $[x_i]$ means a real interval rather than a Z-interval.

The single equation $ax+by = c$ can be trivially balanced in constant time (property 3). For a system,a special order have to be employed to revise the domain of variable. We decompose C into a set of subsystems $\{CS_{f_i}\}$ according to free variables in C. CS_{f_i} contains all the equations that contain the same free variable x_{f_i}. Obviously,

$$vars(CS_{f_i}) \cap vars(CS_{f_j}) = \emptyset \text{ and}$$
$$\bigcup_{f_i} CS_{f_i} = C.$$

For each subsystem CS_{f_i}, we obtain $[x_{f_i}]$ by intersecting the projections of each equation in CS_{f_i} on x_{f_i}. The projection on subject variable of CS_{f_i} can be obtained by the corresponding equation containing the subject variable. Under new bounds of variables,

CS_{f_i} is balanced. Obviously the complexity to balance CS_{f_i} is linear of $|CS_{f_i}|$. Hence force, the system is balanced in $a\sum|CS_{f_i}| = ae$,where a is a constant, the time to balance a single equation.□

Corollary 4 *The binary equation system (V, D, C) with C in solved form can be made bound consistency in $\mathcal{O}(e\log a)$ where a is the greatest coefficient in C.*

Proof:

As in the proof above, we only need to prove that each subsystem CS_f can be made bound consistent linearly. In order to evaluate projections of CS_f on x_f, Euclidean algorithm has to be employed. (r_j, p_j, q_j) can be obtained for each equation in CS_f in $\mathcal{O}(\log a')$ where a' is the coefficient of greatest magnitude in CS_f. For each equation

$$a_jx_j + b_jx_f = c_j,$$

we have

$$x_f^j = -\frac{c_j}{r_j}q_j - a_j\frac{z_j}{r_j}.$$

By making

$$x_f^1 = x_f^2 = \cdots = x_f^{|CS_f|}$$

and applying 4.4 $(|CS_f| - 1)$ times, the general solution of x_f can be obtained. We obtain $[x_f]$ as in the proof of last proposition and adjust the bounds of x_f in terms of its general solution to get the ultimate bounds of x_f. At last, bounds of subject variables are obtained according to the bounds of x_f. So, the complexity to enforce bound consistency on CS_f is $\mathcal{O}(|CS_f|\log|a'|)$. The whole system can be made bound consistent in $\mathcal{O}(e\log|a|)$ where a is the coefficient of the greatest magnitude in C. □

We conclude this section by the following bound consistency enforcing algorithm on an arbitrary binary equation system (V, D, C).

- Transforming C into its solved form by Gaussian-Jordan elimination,
- Selecting a special order to revise the bounds of variables as proposed by corollary 4.

It is known that the complexity of Gaussian-Jordan elimination on a binary system is $\mathcal{O}(e)$ ([AS80]) and thus we have

Corollary 5 *The worst case complexity of the above algorithm is $\mathcal{O}(e \log a)$, where a is the greatest coefficient ever produced in Gaussian-Jordan elimination.*

Note that the complexity of the above algorithm does not depend on the size of domain any longer.

4.3.3 Better Fixed Point on Solved Form

By algebraic manipulation, not only an efficient algorithm can be achieved but also a better fixed point can be obtained, which implies that more invalid values will be pruned on the solved form.

The following lemma says that if a constraint is bound consistent, the projection of the constraint on any variable is exactly the bounds of the variable. Note this is not true for n-ary equation $(n > 2)$.

Lemma 1 *The fixed point $([x], [y])$ of a constraint cs*

$$ax + by = c$$

satisfies $[x] = \frac{1}{a}[c - b[y]]$.

Proof:

In order to construct a contradiction we assume $[x] \neq \frac{1}{a}(c - b[y])$.By the definition of bound consistency, we have

$$[x] \subset \frac{1}{a}[c - b[y]] \tag{4.6}$$

It is easy to verify that for any two interval I_1 and I_2

$I_1 \subset \frac{1}{a} I_2$ implies $aI_1 \subset I_2$, and $I_1 \subset c + I_2$ implies $I_1 - c \subset I_2$.

From 4.6 we have

$$\frac{1}{-b}[a[x] - c] \subset [y]$$

which contradicts that the constraint is bound consistent.□

Proposition 24 *The fixed point achieved by the bound consistency algorithm on a binary system (V, D, C) is not smaller than that achieved on system (V, D, C') where C' is a solved form of C.*

Proof:

The solved form C' can be obtained by a series of variable elimination. Assume, by Gaussian-Jordan elimination, we get a sequence of systems

$$C_0 = C, C_1, \cdots, C_k = C',$$

where C_i is obtained by eliminating one variable from C_{i-1}. Now we prove that the maximal fixed point of CS_h is not greater than that of C_{h-1}.

Let x_i be the eliminated variable of C_{h-1} and the substitution for x_i be

$$x_i = \frac{1}{a_{i1}}(b_i - a_{i2}x_k) \tag{4.7}$$

which is shared by both C_{h-1} and C_h. Let the maximal fixed point of C_h be

$$\tilde{x}^h = ([x_1], [x_2], \cdots, [x_n]).$$

Evidently,all the equations in C_{h-1} which are shared by CS_h are bound consistent under $\tilde{x}^h$.

Now consider equation in CS_{h-1}

$$a_{j1}^{h-1} x_i + a_{j2}^{h-1} x_l = b_j^{h-1} \tag{4.8}$$

where x_i will be substituted out. After x_i is substituted out, the resulted equation in C_h must be one of the following forms

$$\begin{aligned} &0 = 0 \\ &0 = c(c \neq 0) \\ &x_k = \text{fixed value} \\ &a_{j1}^h x_k + a_{j2}^{h-1} x_l = b_j^h. \end{aligned}$$

The first case implies $l = k(x_k$ and x_l are the same variable) and

$$\begin{pmatrix} a_{j1}^{h-1} \\ a_{j2}^{h-1} \\ b_j^{h-1} \end{pmatrix} = \frac{1}{\lambda} \begin{pmatrix} a_{i1} \\ a_{i2} \\ b_j \end{pmatrix} \lambda \in R.$$

Therefore,$[x_i]$ and $[x_k]$ being a fixed point of (4.7) implies that $[x_i]$ and $[x_k]$ is also a fixed point of (4.8).

The second case means that $[x_i]$ is empty and thus the maximal fixed point of C_h is not greater than that of CS_{h-1}.

The third case implies $l = k$ and x_k is fixed.By (4.7) x_i will also take a constant value.Naturally, $[x_i]$ and $[x_k]$ satisfies (4.8).

In last case, substituting out x_i in (4.8) gives

$$a_{j1}^{h-1}(\frac{1}{a_{i1}}(b_i - a_{i2}x_k)) + a_{j2}^{h-1}x_l = b_j^{h-1} \tag{4.9}$$

whose simplified form in C_h is

$$a_{j1}^{h}x_k + a_{j2}^{h-1}x_l = b_j^{h} \tag{4.10}$$

It is easy to verify that the fixed point $([x_k], [x_l])$ of (4.10) is also the fixed point of (4.9). By lemma and (4.7), we have

$$\frac{1}{a_{i1}}[b_i - a_{i2}[x_k]] = [x_i]$$

and thus by (4.9),$[x_i]$ and $[x_l]$ is a fixed point of (4.8).

In a word,the fixed point $([x_1], \cdots, [x_n])$ is also a fixed point of C_{h-1}. Therefore it is smaller than or equal to the maximal fixed point of C_{h-1}. □

Definition 26 *A* solution *of (V, D, C) is an assignment of values to all variables in V such that all domain constraints and C are satisfied.*

A system (V, D, C) is consistent *if there exists a solution for it.*

Proposition 25 *A binary equation system with solved form is consistent if it is bound consistent.*

Proof:

A solution can be obtained as following. Assign each free variable the lower bound.

Each subject variable will take either upper bound or lower bound according to the constraint imposed on it.□

Corollary 6 *For a binary equation system ,the bound consistency algorithm achieves the same fixed point on any solved form of C. Furthermore, that fixed point is the smallest one that can be achieved on all possible equivalent systems.*

It is interesting to observe that for a binary equation system with C in solved form, arc consistency (actually n-consistency) can be easily obtained on the base of bound consistency. After bound consistency enforcing,the general representation of a free variable of C is of the following form

$$x_f = Lb(x_f) + i\triangle \tag{4.11}$$

where i is an integer greater than or equal to 0 and $\triangle$ is the step size(see 4.4). Any value satisfying 4.11 can be extended to a solution of the system.

The nature of the bound consistency enforcing algorithm and Gaussian elimination is incremental. So, all algorithms discussed above is applicable to the solver of a CLP over finite domain.

Chapter 5

On Solving of n-ary Linear Constraints

Given an equation system (V, D, C), we will give in this chapter a discussion on the influence of Gaussian-Jordan elimination on solving the system. Precisely, we want to know if consistency algorithm is faster on the solved form than the original form ,if the consistency algorithm can obtain a better fixed point on the solved form than on the original form, and the influence of the solved form on the searching procedure. As is shown in the last chapter, the answer is positive for binary system. However, for n-ary system, there is not a simple answer of yes or no. Here, we only discuss the first and the last issue. Section 1 gives a mathematical view of bound consistency. Section 2 gives some preliminary experimental results on the impact of Gaussian-Jordan elimination on the solving of n-ary system.

5.1 Mathematical view of bound consistency

5.1.1 Fourier elimination

Fourier elimination ([Koh73],[Koh67] and [Duf74]) is a useful tool in linear programming. The principle of Fourier elimination can be illustrated by the following example. Consider the following linear inequalities:

$$\begin{array}{rrcl} x_1 & & \geq & 0 \\ x_1+ & 2x_2 & \leq & 6 \\ x_1+ & x_2 & \geq & 2 \\ x_1- & x_2 & \geq & 3 \\ & x_2 & \geq & 0 \end{array} \tag{5.1}$$

In order to obtain the range of x_2, we need to eliminate x_1 as follows. We divide the 5 inequalities with respect to x_1 into three classes. The first class contains all the inequalities in which x_1 can be expressed as greater than certain linear terms, the second contains those in which x_1 can be expressed as less than other linear terms, and the third class includes those where no x_1 occurs. By eliminating x_1,the new constraint system includes the third class of constraints and those constraints by adding up any two constraints from first class and second class.For the example, after classifying we have

$$\begin{array}{rcl} & x_1 & \leq 6-2x_2 \\ 0 \leq & x_1 & \\ 2-x_2 \leq & x_1 & \\ 3+x_2 \leq & x_1 & \\ & x_2 & \geq 0 \end{array} \tag{5.2}$$

And the new system is

$$\begin{aligned} 0 &\leq 6 - 2x_2 \\ 2 - x_2 &\leq 6 - 2x_2 \\ 3 + x_2 &\leq 6 - 2x_2 \\ x_2 &\geq 0 \end{aligned} \tag{5.3}$$

by which we have $x_2 \leq 1$. The Fourier elimination can be used to obtain a solution of linear program [Dan63] and applied to integer programming[DE73]. It has also be applied to obtain a projection of a constraint system on a set of variables[JMPY92] in the CLP community.

The greatest weakness of the Fourier elimination is that possibly exponential number of constraints will be generated. However, by removing those redundant constraints in each elimination step, the number of new generated constraints can be significantly reduced([Cer63][Koh67]).

Consider the linear constraint system (V, D, C). The bound consistency actually can be interpreted as follows.

```
While $(V, D, C)$ is not bound consistent
  For each constraint $c_i \in C$
    For each variable $x_{i_k} \in vars(c_i)$
      Apply Fourier elimination to $\{c_i\} \cup D$ to obtain the projection of $c_i$ on $x_{i_k}$;
      round the bound of $x_{i_k}$ to integer.
```

An immediate generalization of bound consistency is to obtain the projection of a subset,rather than a single constraint c_i, of C on x_i . One extreme is to apply Fourier elimination on C to obtain the projection on each variable x_i and then round bounds of x_i to integers(This kind of generalization of the bound consistency is somewhat similar to [DB95]). In that case,bound consistency enforcing algorithm can reach the fixed point which is a subset of the cube evenly containing the feasible region of the relaxation of linear system(D, C) to real.

Our concern here is how the Gaussian elimination of equations affect the bound consistency. From the above view, Gaussian elimination itself can not guarantee to achieve higher consistency, or achieve a smaller fixed point if we still adopt the bound consistency defined in chapter 2. However, it is itself interesting to study the influence on bound consistency of algebraic transformation while in general CSP it is not convenient to do so. Another reason is that Gaussian elimination is an efficient algorithm and is widely implemented in CLP system,such as CLP(R), CHIP, Prolog III etc. The third reason is as indicated in chapter 2.

Note.[For97] transfers the control of Fourier elimination to users by providing some built-in predicates which are used to choose a set of constraints and a set of variables to which the Fourier elimination is applied.Some other kinds of generalization of bound consistency appears in [Lho93].But we will not discuss them here because of their irrelevance to the current topic.

5.2 Algebraic Structure and Solving

Experiments in the section are based on the following puzzle which is a frequently used benchmark.

5.2.1 A Puzzle

Consider the crypt-arithmetic problem from the Mathematical Puzzles of Sam Loyd. The owner of a general store, who is something of a puzzlist, has put up this sign to see if any of his mathematical friends can translate it properly. Each different letter stands for a different digit. The words above the horizontal line(see figure 5.1) represent numbers that add to the total of "ALL WOOL". The problem is to change all the letters to the correct digits .

```
      C H E S S
+       C A S H
+   B O W W O W
+     C H O P S
+   A L S O P S
+ P A L E A L E
+       C O O L
+       B A S S
+       H O P S
+       A L E S
+       H O E S
+   A P P L E S
+       C O W S
+   C H E E S E
+   C H S O A P
+     S H E E P
---------------
  A L L W O O L
```

Figure 5.1: ALL WOOL

Obviously, this problem can be described by the constraints among characters. Here, we model the problem as follows. All the characters occurring in the problem can be thought of as variables whose range is over 0..9. The constraints over the variables are

```
S+H+W+S+S+E+L+S+S+S+S+S+S+E+P+P            = L+ 10*C11+ 100*C12

S+S+O+P+P+L+O+S+P+E+E+E+W+S+A+E+ C11       = O+ 10*C21+ 100*C22

E+A+W+O+O+A+O+A+O+L+O+L+O+E+O+E+ C21 + C12 = O+ 10*C31+ 100*C32
```

```
H+C+W+H+S+E+C+B+H+A+H+P+C+E+S+H+ C31 + C22 = W+ 10*C41+ 100*C42

C+  O+C+L+L+           P+  H+H+S+ C41 + C32 = L+ 10*C51+ 100*C52

    B+  A+A+           A+  C+C  + C51 + C42 = L+ 10*C61+ 100*C62

          P                     + C61 + C52 = A
```

where `Ci1` and `Ci2` represent the two possible carries for *ith* column. One advantage of this modeling(which is not a straightforward description of the problem) is that more pruning can be achieved with bound consistency techniques.

5.2.2 Variable Ordering in Linear Constraints

As indicated in chapter 2, a general way to improve the efficiency of searching procedure is to employ a good ordering of variables. This method has been proved successful in solving combinatorial problem by CLP languages like CHIP,ILOG. Generally, a dynamic variable ordering(DVO) is employed by such languages. One heuristics based on first fail principle is to choose the variable with minimum domain first. Constraint solvers of CLP languages over finite domain are equipped with consistency techniques which prune the search space in each computation step. In that case, the size of the domain of variable more or less reflects the constrainedness on the variable of the constraint system. The fewer values left in the variable's domain mean more active constraints on the variable and thus it more possibly fails to instantiate this variable. Here the constrainedness is approximated by the size of domain.

Given a linear constraint system (V, D, C).Assume all variables have the same initial domain. We give a characterization of a constrainedness according to the semantics of the constraints.

Definition 27 *The* constrainedness *on a variable x_{i_k} of a linear constraint c_i is*

$$\theta^i_{i_k} = \frac{1}{|a_{i_k}|} \sum^{i_l \neq i_k} |a_{i_l}|.$$

Intuitively, the variable with the greater θ will be more restricted than the other variables in the constraint.

Definition 28 *The* constrainedness θ_{i_k} *on a variable x_{i_k} of a linear system (V, D, C) is*

$$\theta_{i_k} = \sum_{i_k \in vars(c_j)} \theta^j_{i_k}.$$

An ordering of variable of a linear constraint system can be easily obtained according to the constrainedness of variables. An alternative is to get the constrainedness by simulating the consistency enforcing. The idea is that once the most constrained variables is decided, its coefficient will be set to 0 and the constrainedness of the rest variables are recalculated and this is repeated until no variable is left. For the puzzle, we have following experiment results (the entry in the following table is the number of backtracks). VO_1

	RO	DVO	VO_1	VO_2
C	10709	1110	826	1586

is the variable ordering obtained by the second approach while VO_2 is that obtained by the first approach.The RO is an arbitrarily variable ordering. DVO refers to the variable ordering obtained by the heuristics of choosing the variable with minimum domain first.

5.2.3 Algebraic Structure, Consistency and Search

In this section, the domain of variable in the puzzle is generalized to range from 1 to d.

C can be written as

$$Ax = b, \quad A \in R^{e \times m}, \quad b \in R^e.$$

Let the solved form C' of C by Gaussian-Jordan elimination be

$$Bx_B = Nx_N + b', \quad B \in R^{e' \times e'}, \quad N \in R^{e' \times m - e'}$$

where e' is the number of constraints left after the elimination.

When A is dense, the bound consistency enforcing algorithm on C' is faster than on C. Obviously, after Gaussian-Jordan elimination, each constraint is shortened by e' and when there are redundant equations in C e' will be smaller than e.

In our experiment to compare the maximal fixed point between the original system and the solved form, we substitute out all the carries in the puzzle with correct values and get a system which can be regarded as a dense system. The results are shown in Table 5.1.

d		Fixed Point									
		C	H	E	S	A	B	O	W	P	L
9	C	0..9	0..9	0..8	0..2	2..8	0..9	2..9	0..9	0..6	0..9
	C'	0..6	3..8	0..4	0..3	2..7	0..9	6..9	0..9	0..5	4..9
32	C	0..16	0..9	0..9	0..2	2..12	0..32	0..13	0..20	0..10	0..32
	C'	0..6	1..8	0..4	0..6	2..7	0..30	6..11	0..32	0..5	4..20

Table 5.1: The fixed point achieved on different system

Results show that for a dense matrix A, the maximal fixed point of the solved form is better than that of the original system. The *better* here means most components of a fixed point are smaller. Note that variable S in C has a smaller interval than in C', which shows that even C and C' are algebraicly equivalent, the fixed points of these two system are not comparable.Actually, the bound consistency treats each constraint in C separately and thus change of the syntactical form of individual constraint results in the change of the maximal fixed pointed.

Of course, more experiments need to be carried out to characterize what is *dense* and *better*.

As for searching on the dense system, we have following results (see following table). VO_1 refers to variable ordering (`A,E,L`) while VO_2 refers to variable ordering (`L,E,A`).

	NO. of Backtracks			
	VO_1	VO_2	VO_1	VO_2
C	20	138	62	110
C'	1	3	3	3

Table 5.2: Searching on a dense system.

Actually we believe that Gaussian-Jordan elimination is definitely useful when A is dense.

When A is sparse, N is not necessarily sparse. So, we may exploit the special structure as what has been done in the area of sparse matrix computation. For band and variable-band matrices [DER86] the Gaussian-Jordan elimination is definitely useful. [DER86] also discusses algorithms and heuristics to ordering a sparse matrix to the special form like band or variable-band matrices.

For an arbitrary sparse matrix A, experiment shows that solving the problem on the original constraint system has almost the same performance as on the solved form,which implies that in a tightly coupled solver the solved form can be shared by both real solver and finite domain solver. In the experiment four different forms of the constraint system are used. C represents the original constraint system,SS the solved form of C obtained by Mathematica by default, and SO the solved form obtained by specifying subject variables in terms of a special variable ordering.SM is obtained as follows. We attempt to obtain a solved form by specifying the 7 least constrained variables as subject variables. Actually, the 7 variables can not be subject variables simultaneously.So, we only obtain a mixture which is composed of a solved subsystem and the rest equations. In table 5.3,VO_1 refers

$d = 9$			d=32
	VO_1	DVO	VO_1
C	826	1110	242
SS	963	1132	200
SO	868	725	191
SM	795	768	180

Table 5.3: Number of backtracks on different form.

to the variable ordering (`C52, C62, C12, C22, C42, C32, C61, A, P, C51, C, S, L, B, H, C41, O, E, C31, C21, W, C11`) which is sorted decreasingly according to constrainedness of variables in the original system.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The research work presented in this thesis focuses on two important class of constraints –basic constraints and n-ary linear constraints.

Due to the nice property of functional constraints, we propose an efficient algorithm to enforce strong n-consistency on FCs whose space complexity is the same as that of the optimal arc consistency enforcing algorithm and time complexity is almost the same as that of the optimal arc consistency enforcing algorithm to enforce arc consistency. When there is no conflict of orienting, our algorithm is optimal in both time and space complexity. Specially, the conflict of orienting can be avoided for a CSP all of whose constraints are known beforehand. Another characteristic of our algorithm is that all functional constraints need to be checked only once. We also give a detailed discussion on how to deal with the conflict of orienting. The results show that even in the presence of conflict of orienting,our algorithm has a good performance($\mathcal{O}(n' \log n'd + ed)$). As for worst case complexity, our algorithm has great advantage over the general path consistency[1]

[1]For functional constraints, the path consistency implies strong n-consistency.

algorithms(time complexity $\mathcal{O}(n^3d^3)$ space complexity (n^3d^3))[MH86][HL88].

As for bound consistency, the most related work is [BO97]. However, [BO97] does not give enough attention to linear constraints over finite domains. In this thesis we give analysis of complexity of bound consistency algorithms. We propose an efficient algorithm whose worst case complexity is $\mathcal{O}(n^2ed)$. We also analyze first time the influence of algebraic transformation on bound consistency. Our study gives a nice result that for binary equation system, the Gaussian-Jordan (G-J) algorithm helps to obtain a global consistency and a fast consistency enforcing algorithm. However, the result no longer holds for n-ary linear constraints. Our empirical study shows that for problem with dense coefficient matrix, the G-J elimination greatly improves constraint solving from consistency enforcing to the whole search procedure. For sparse matrix, although the consistency enforcing algorithm can not be sped up, the performance of the search procedure on the solved form is close to that on the original form. The above observation implies that the G-J elimination is not only useful as a relaxed global satisfiability tester, but also can provide a better form of constraints which will benefit the consistency enforcing algorithm and the whole search procedure.

6.2 Future Works

This thesis presents a fast algorithm for achieving strong n-consistency on functional constraints. As for monotonic constraints, we have the following conjecture. There exists a fast algorithm for MCs which can also achieve strong n-consistency on MCs. The key issue is how to deal with loops. Here loops is similar to those of functional constraints while the detection of loops is fast. Once the conjecture is true, the algorithm for enforcing strong n-consistency on both MCs and FCs can be easily obtained. However, the solving of anti-functional constraints over two variables is NP-complete.

It is still not clear about the role played by the syntactical form of constraints in bound consistency enforcing and constraint solving. The intuition is that a more compact form can improve constraint solving. Specifically, does there exist a compact form for an arbitrary linear constraints such that bound consistency enforcing can both be faster and achieving more pruning(smaller maximal fixed point)? We do not even know how to describe the *compact* form accurately. A practical way is to carry out more experiments to explore the influence of typical algebraic transformations on constraint solving. Except for G-J elimination, there are other elimination methods–for example the Gaussian elimination, BDB form [BSTY95] and other forms [Imb95].

Some work can also be done to define higher level of bound consistency and examine the impact of higher level of consistency on solving of linear constraints over finite domains.

Bibliography

[AS80] Aspvall, Bengt and Shiloach, Yossi A fast algorithm for solving systems of linear equations with two variables per equation. Linear Algebra Appl. 34, 117–124, 1980

[BC93] Bessiere,C. and Cordier,M., Arc-consistency and arc-consistency again, *AAAI-93*,108-113,1993

[BDB94] Beringer, H., and De Backer, B. Combinatorial Problem Solving in Constraint Logic Programming with Cooperating Solvers. *Logic Programming:Formal Methods and Practical Applications*, C. Beierle and L. Plumer(eds.). Elsevier Science Publishers B. V. 1994.

[BMH94] Benhamou,F., McAllester,D. and van Hentenryck, P. , CLP(Intervals) Revisited, *Proceedings of 1994 International Symposium on Logic Programming*,124-138,1994

[BO97] Benhamou,F. and Older, W., Applying Interval Arithmetic to Real, Integer and Boolean Constraints,*Journal of Logic Programming 32(1)*,1997

[BP81] Brown, C.A., and Purdom, P.W.Jr., How to search efficiently, *Proceedings of 7th International Joint Conference on AI*,588-594,1981

[BSTY95] Burg,J., Stuckey,P.J., Tai, J. and Yap, R., Linear Equation Solving for Constraint Logic Programming, *Proceedings of the 12th International Conference on Logic Programming*,1995

[CC93] Carlson, Bjorn, Carlsson, Mats, Constraint Solving and Entailment Algorithms for $cc(FD)$, manuscript,1993.

[CD96] Codognet,P. and Diaz,D., Compiling Constraints in $CLP(FD)$, Journal of Logic Programming,27(3), 185-226, 1996.

[Cer63] Cernikov,S.N., Constractin of Finite Systems of Linear Inequalities, *Soviet Mathematics DOKLADY 4* ,1963

[CL94] Chiu,C.K. and Lee, J.H.M., Towards Practical Interval Constraint Solving in Logic Programming,*Principles and practice of constraint programming : Second International Workshop, PPCP '94*,109-123, 1994

[Cle87] Cleary,J.G., Logical Arithmetic, *Future Computing Systems 2(2)*, 125-149,1987

[CLR90] Cormen,T.H., Leiserson,C.E., and Rivest,R.L., *Introduction to Algorithms*,1990

[Col90] Colmerauer, A., Prolog III Reference and Users manual, Version 1.1, PrologIA, Marseilles, 1990.

[Dan63] Dantzig,G.B., *Liner Programming and Extensions*,1963

[DB95] Dechter, Rina and van Beek, Peter, Local and Global Relational Consistency, *Principles and practice of constraint programming -CP '95* , Cassis, France, 1995

[DE73] Dantzig,G.B.,Eaves, B.C., Fourier-Motzkin elimination and its dual, *Journal of Combinatorial Theory (A)14*,288-297, 1973

[Dec90] Dechter, R., From Local to Global Consistency, *Artificial Intelligence 55*, 87-107, 1992

[DER86] Duff,I.S., Erisman,A.M. and Reid,J.K., *Direct Methods fro Sparse Matrices*, Clarendon Press, 1986

[DP92] Dechter, R. and Pearl,J., Structure Identification in Relational Data, *Artificial Intelligence 58* 237-270,1992

[DVHS87] Dincbas, M., van Hentenryck,P., Extending Equation Solving and Constraint Handling in Logic Programming, *Colloquium on Resolution of Equations in Algebraic Structures*, Texas, 1987

[DVHS88] Dincbas, M., van Hentenryck,P., Simonis,H. and Aggoun, A., The Constraint Logic Programming Language CHIP, *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, 249-264, 1988

[Duf74] Duffin,R.J.,On Fourier's Analysis of Linear Inequality Systems, *Mathematical Programming Study 1*,71-95,1974

[For97] Fordan, A., Linear Projection in CLP(FD), manuscript, 1997

[Fre78] Freuder, E.C., Synthesizing Constraint Expressions, *Communications of ACM 21(11)*, 958-966, 1978

[Fre82] Freuder, E.C., A sufficient condition for backtrack-free search, *Journal of ACM, Vol 29(1)*,24-32,1982

[Fre90] Freuder,E.C.,Complexity of K-tree Structured Constraint Satisfaction Problems, *Proceedings of National Conference on Artificial Intelligence*,4-9,1990

[Gol96] Golub, G.H., *Matrix computations*, Johns Hopkins U P, 1996

[Han92] Hansen, E.R., *Global optimization using interval analysis*, Marcel Dekker Inc., 1992

[HE80] Haralick, R.M., and Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence , Vol14*, 263-313, 1980

[HL88] Han, C. and Lee, C., Comments on Mohr and Henderson's Path Consistency Algorithm, *Artificial Intelligence 36*, 125-130, 1988

[HSSJO92] Havens, W., Sidebottom, S., Sidebottom,G., Jones,J. and Ovans,R., Echidna: A Constraint Logic Programming Shell, *Proceedings of Pacific Rim International Conference on Artificial Intelligence*,1992

[ILOG] *ILOG SOLVER Reference Manual Version 3.2*,ILOG,1996

[Imb95] Linear Constraint Solving in CLP-languages, *Constraint programming : basics and trends (LNCS910)*, 108-127,Springer-Verlag, 1995

[JL87] Jaffar, J. and Lassez, J.-L., Constraint Logic Programming, Proceedings 14th ACM Symposium on Principles of Programming Languages, Munich(1987), 111-119, 1987

[JM94] Jaffar, Joxan and Maher, M. J., Constraint Logic Programming,*Journal of Logic Programming* 19/20, 503-581,1994

[JMSY92] Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R.,The CLP(R) language and system, ACM Transactions on Programming Languages,14(3), 339-395, 1992.

[JMSY94] Jaffar, J., Maher,M., S., Stuckey, P. J., and Yap, R., Beyond Finite Domains, *2nd Workshop on the Principles and Practice of Constraint Programming*, 370-395, 1994

[Koh67] Kohler, D.A., Projections of polyhedral sets, Ph.D. Thesis, 1967

[Koh73] Kohler, D.A., Translation of A Report by Fourier on His Work on Linear Inequalities, *Opsearch 10*,38-42,1973

[Kum92] Kumar,V., Algorithms for Constraint Satisfaction Problems : A Survey, AI Magazine 13(1),32-44.1992

[Liu95] Liu,B., Increasing Functional Constraints Need to be checked only once, *International Joint Conference on Artificial Intelligence 95*, 1995

[Liu96] Liu,B., An Improved Generic Arc Consistency Algorithm and Its Specialization, *4th Pacific Rim International Conference on Artificial Intelligence*,1996

[Lau78] Lauriere,J.,Alanguage and a program for stating and solving combinatorial problems, *Artificial Intelligence 10*,29-127,1978

[Lho93] Lhomme, Olivier, Consistency Techniques for Numeric CSPs, *Proceedings of IJCAI-93*,Chambery,France,232-238, 1993

[Llo87] Lloyd,J.W., *Foundations of Logic Programming*, Springer-Verlag, Second Edition,1987

[Moo66] Moore, R.E., *Interval Analysis*,Prentice Hall, 1966

[Mac77] Mackworth,A. K., Consistency in Networks of Relations, *Artificial Intelligence 8(1)*,118-126,1977

[Man93] Mantsivoda,A., Flang and its Implementation, *Proceedings Symposium on Programming Language Implementation and Logic Programming*, LNCS 714,151-166,1993

[MF85] Mackworth,A. K. and Freud,E.C.,The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence 25*, 65-74, 1985

[MH86] Mohr, R. and Henderson, T.C., Arc and Path Consistency Revisited, *Artificial Intelligence 28*, 225-233, 1986

[MM88a] Mohr,R. and Masini, G., *Running efficiently arc consistency,Syntactic and Structural Pattern Recognition*, Springer, Berlin, 217-231, 1988

[MM88b] Mohr,R. and Masini, G., Good Old Discrete Relaxation, *Proceedings of ECAI-88*,Munich,Germany,1988

[Mon74] Montanari, U., Networks of Constraints: Fundamental Properties and Applications in , *Information Science 7(2)*,95-132,1974

[NW88] Nemhauser, G. L. and Wolsey, L. A., *Integer and Combinatorial Optimization*, New York, Wiley ,1988

[Nil80] Nilson, N.J., *Principles of Artificial Intelligence* , Tioga,1980

[OB93] Older, W. and Benhamou,F., Programming in CLP(BNR), *Proceedings of PPCP'93*,Newport, 1993

[OV90] Older, W. and Vellino,A., Extending Prolog with Constraint Arithmetic on Real Intervals, *Proceedings of the Canadian Conference on Electrical and Computer Engineering*,1990

[OV93] Older, W. and Vellino,A., Constraint Arithmetic on Real Intervals, *Constraint Logic Programming:Selected Research*,Benhamou,F. and Colmerauer,A.(eds.), 175-195,1993

[Per91] Perlin,M., Arc Consistency for Factorable Relations, *Proceedings of the 1991 IEEE International Conference on Tools for AI*,340-345, San Jose,CA,1991

[Pug94] Puget,J.F.,A C++ Implementation of CLP, *Proceedings of Singapore International Conference on Intelligent System*,1994

[RPD89] Rossi, F.; Petrie, C.; and Dhar, V. 1989. On the equivalence of Constraint-Satisfaction Problems, Technical Report ACT-AI-222-89, MCC Corp., Austin, Texas.

[RWH96] Rodosek, R., Wallace, M. G. and Hajian, M T, A New Approach to Integrate Mixed Integer Programming with CLP (extended abstract), *Proceedings of the CP96 Workshop on Constraint Programming Applications: An Inventory and Taxonomy*, 45-54, Cambridge, Massachusetts, 1996

[RWH97] Rodosek, R., Wallace, M. G. and Hajian, M T, A New Approach to Integrating Mixed Integer Programming with Constraint Logic Programming: An Extension, *to appear in Annals of Operational Research. Recent Advances in Combinatorial Optimization*, 1997

[Sho67] Shoenfield,J.R., *Mathematical Logic* , Addison-Wesley,1967

[Tsa93] Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press,1993

[vH89] van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, 1989

[vH90] van Hentenryck, P., Incremental Constraint Satisfaction in Logic Programming, *Proceedings of Seventh International Conference on Logic Programming*, 189-202, 1990

[VHD91a] van Hentenryck, P., and Deville, Y., Operational Semantics of Constraint Logic Programming over Finite Domains,*Proceedings of 3rd International Symposium on Programming Language Implementation and Logic Programming*,1991

[VHD91b] van Hentenryck and Deville, Y. The Cardinality Operator: A New Logical Connective and its Application to Constraint Logic Programming, *Proceedings of International Conference on Logic Programming*, 745-759, 1991.

[VHDT92] van Hentenryck, P., Deville, Y. and Teng, C.-M., A Generic Arc-Consistency Algorithm and its Specializations, *Artificial Intelligence 58*, 291-321, 1992

[VHSD91] van Hentenryck, P., Saraswat, V., and Deville, Y., Constraint Processing in cc(FD), manuscript, 1991

[VHSD93] van Hentenryck, P., Saraswat, V., and Deville, Y., Design, Implementations and Evaluation of the Constraint Language *cc(FD)*, Technical Report CS-93-02, Brown University, 1993

[VHMD97] van Hentenryk, P., Michel,L. and Deville,Y., *Numerica: A Modeling Language for Global Optimization*, MIT Press, Cambridge, 1997

[Wal72] Waltz,D., Generating Semantic Descriptions from Drawings of Scenes with Shadows,Tech. Rept. AI271, MIT, Cambridge, 1972

[Zab90] Zabih,R. Some applications of graph bandwidth to constraint satisfaction problems,*Proceedings of National Conference on Artificial Intelligence(AAAI) 1990*,1990

[ZW98] Zhang, Yuanlin and Wu, Hui, Bound Consistency on Linear Constraints in Finite Domain Constraint Programming, *Proceedings of ECAI-98*, Brighton, UK, 1998