CONSISTENCY TECHNIQUES IN CONSTRAINT NETWORKS

ZHANG YUANLIN

NATIONAL UNIVERSITY OF SINGAPORE

2003

CONSISTENCY TECHNIQUES IN CONSTRAINT NETWORKS

ZHANG YUANLIN

张元林

(BEng, EAST CHINA INSTITUTE OF TECHNOLOGY (华东工学院)

MSc, NATIONAL UNIVERSITY OF SINGAPORE)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR IN PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgments

有朋自远方来, 不亦乐乎?
Is it not delightful to have friends coming from distant quarters?
—Confucius

It is really a great pleasure to meet so many people in different places who have helped me to shape and go smoothly through the path leading to the completion of my doctoral thesis, and make the path enjoyable and memorable. I owe them an immense debt of gratitude and appreciation.

Roland YAP and Joxan JAFFAR introduced me to the interesting topic of Constraint Programming. The motivating discussions with them play an essential role in the preparation of this thesis. They also taught me how to sharpen ideas and present ideas precisely and clearly. Roland has patiently read and commented on every draft of this thesis. I am lucky to have them as the supervisors of my research. I also appreciate their help in making my personal life in Singapore comfortable.

I am grateful to Peter van BEEK for invaluable discussions on set intersection and consistency, an important component in this thesis, and other generous help. Thanks also go out to LIU Bing for the helpful meetings at the early stage of my study in NUS. The comments from Krzysztof APT and Martin HENZ improve the quality of this thesis.

I would like to thank my teachers and friends without whom I would not have been able to start my PhD study. Especially, thanks to WANG Jiasong (王嘉松) who always supported and encouraged me, and introduced NUS to me; OUYANG Zixiang (欧阳梓祥) who supervised my research in Nanjing University; PENG Mingjin (彭明金) and WANG Xinghua (王醒华) who cultivated and kept my interest in physical sciences; XU Weihong (徐蔚鸿) who encouraged me to pursue a postgraduate study; and ZHAO Jiatian (赵嘉田) who initiated the first impetus for me to study abroad and showed me what perseverance is.

I would also like to thank my parents for the support they have provided me all the time; and thank my wife, Xiaoyan (小艳), whose love, patience and support are always the source of strength, and my children Yutong (羽童) and Yufan (羽凡) who bring a lot of fun into my life.

Contents

List of Figures

Summary

Many problems across Artificial Intelligence (AI), Computer Science and Operations Research (OR) can be modeled easily by a Constraint Network. More importantly, Constraint Programming (CP) systems have shown that many difficult real life problems can be efficiently solved under this model. A constraint network describes a problem as a set of variables with finite values and a set of constraints among certain variables. Some of its applications include problems in manufacturing, transportation, telecommunication, logistics and bio-informatics.

Since finding a solution for a constraint network is NP-hard, a systematic search procedure is employed. Pruning the search space by making an active use of constraints proves to be an effective way to make the search procedure practical. This can be abstracted into the concept of various levels of local consistency in a constraint network. Arc consistency is one such well known local consistency.

In this thesis, we consider two aspects of consistency. Firstly, as a pruning facility, enforcing arc consistency on a constraint network is at the core of a search procedure. It is desirable to make such an algorithm as fast as possible. I have several contributions on this aspect. AC-3, an algorithm to enforce arc consistency on a network of binary constraints, has been widely employed by the researchers and practitioners since its invention in 1977 by Mackworth. However, its worst-case time complexity was not regarded as optimal although it performs well in practice. We show that AC-3.1, a new implementation of AC-3, is of optimal worst case complexity and better experimental performance than the traditional understand-

ing of AC-3. The implementation techniques can also be applied to other levels of local consistency, for example path consistency. We also study arc consistency on non-binary constraints, each of which may involve more than two variables. It is NP-hard to enforce arc consistency on a general non-binary constraint. We identify a class of constraints—monotonic constraints. Arc consistency can be efficiently enforced on them. The important and ubiquitous linear arithmetic inequalities belong to this class.

Secondly, consistency is also used as a tool to identify properties to help globally solve particular constraint networks. Across many fields of computer science, it is desirable to infer global information through local computation. So is the case for constraint networks where the computation of local consistency is desirable to ensure the existence of a solution of the network. We propose a new framework on the relationship between local consistency and global consistency. It unifies a number of well-known results. More importantly, several new results are derived from the framework. For example, I show that a certain level of local consistency in a network of *tree convex* constraints implies global consistency. This is a generalization of existing work on row convex constraints. Another example is that a network, with properly m-tight constraints on certain variables, can be made globally consistent by making it relationally $m + 1$-consistent—another type of local consistency. This is a significant improvement over existing work on consistency and tightness of constraints.

Along the line of the second aspect, I also study a special class of constraints—functional constraints which are a primitive in CP Languages. Efficient and elegant algorithms are designed to solve a network of binary functional constraints. They can also be employed to solve some other problems efficiently, for example a network of 0/1/All constraints.

Part I

Introduction and Preliminaries

Chapter 1

Introduction

1.1 Context

Our research focuses on the the issue of efficiency in solving a constraint network, from a perspective of consistency techniques.

A constraint network (CN) consists of a set of variables each of which can take a value from a finite domain, and a set of constraints among variables. The constraint satisfaction problem[1] (CSP) is to find an assignment of values to variables such that all constraints in the network are satisfied.

CSPs originated from the study of, and provides a uniform framework [Mac77a] for, various problems in artificial intelligence (AI). Its strong modeling ability lies in that a constraint can be of any form, in contrast to the strict restrictions on constraints in other models. For example, in Linear Programming [Dan63], only linear arithmetic constraints are allowed to describe a problem.

CSP can be employed to model traditional puzzle-solving problems, combinatorial problems, and many artificial intelligence tasks including vision, language comprehension, diagnosis, temporal and spatial reasoning and many others.

[1] We are talking about the classical CSP here. There are many variations of CSP nowadays, e.g. continuous CSP etc.

The ability of CSP to deal with arbitrary constraints was not fully exploited until the emergence of Constraint Programming (CP) over finite domains in the mid 1980s [JL87, VH89]. The CP paradigm makes it possible for a user to express various complex constraints through some programming language. The paradigm employs the CSP model to solve the constraints generated dynamically by a program. It has thus expanded the application of CSP to a wide range of industrial problems, and increased the interest in the study of CSP in the last two decades. Among the most successful CP systems are CHIP [VH89] and ILOG [ILO00].

While CSP is a very expressive model, it is also an NP-hard problem. For example, the graph k-colorability problem can be formulated as a CSP and it is well known as an NP-complete problem [GJ79]. The finiteness of the domains of variables suggests that a search algorithm is sufficient to solve a CSP. The main problem is the *efficiency* of the search procedure. The great success of CSP in real life problems benefits among others[2] mainly from the progress of the study of *search strategies* and the introduction and development of *consistency techniques*. In the last three decades, much theoretical and empirical work has been done on these two directions.

Backtracking [GB65] is a well known search strategy. A lot of improvements over it, for example *backjumping, learning* and *cutset decomposition* [Dec90b], are proposed and explored. Associated with the backtracking based search strategies are heuristics, for example, on which variable and which value of the variable should be tried first. Some of the heuristics significantly decrease the time to find a first solution for a wide range of problems (see [HE80, page 301] and [VH89, page 129]).

The most distinct feature of CSP solving techniques may be the *consistency techniques*, also widely called constraint propagation in the community of AI. The

[2]For example, as pointed out before, the convenience provided by a programming language and the flexibility and efficiency of CSP solving techniques make it practical to solve challenging combinatorial problems, both academic and commercial[VH89, ILO00].

idea of constraint propagation might have been in existence for a long time. However, its usefulness and importance were not realized until Waltz's work on a scene labeling problem [Wal72] and Fikes' on a problem solver REF-ARF [Fik68]. The essential idea behind consistency is that before assigning a value to a new variable in a search procedure, the constraints are actively used to prune the search space. The pruning is to exclude certain values or combination of values, which are not a part of any final solution of the problem, from further consideration.

Abstracting away application oriented details from the empirical work by Waltz and the theoretical work by Montanari [Mon74], Mackworth [Mac77a] introduced node-consistency, arc-consistency and path-consistency as well as algorithms to enforce those consistencies on a constraint network. The identification of the time complexity of the arc-consistency enforcing algorithm by Mackworth and Freuder [MF85] in 1985 raised further interest of the community in consistency techniques. Since then the study of consistency techniques has been greatly widened and deepened.

Arc-consistency (AC) is one of the most useful consistencies. Its variations are widely used in practical systems. The optimal AC enforcing algorithm was discovered by Mohr and Henderson [MH86] in 1986 and subsequently more efficient AC algorithms were developed for both general and special constraints.

Together with the complexity result of AC, the generalization of basic consistency to k-consistency [Fre78], where k can be any number of variables involved in a problem, made it possible to understand the nature of solving a constraint network from the perspective of consistency. In fact, many interesting results have been found. For example, by restricting the topological structure of a constraint network or the semantics of constraints, sufficient conditions to ensure the global consistency of those networks have been identified [Fre82, Fre85, vBD95, vBD97]. Certain constraints [Mon74, CCJ94, DBVH97, JCG97] are also identified such that the network of such constraints can be solved efficiently.

1.2 Results

The research work reported in this thesis considers two roles of consistency techniques in *efficiently* solving a CSP. Firstly, low level consistency is used to prune the search space. Specifically, arc consistency and its variations have been at the core of most CP systems. It is thus very important to devise fast AC algorithms to improve the overall performance of a CP system. Secondly, for certain constraint networks, some level of local consistency implies global consistency. It is desirable to identify the properties of those networks since local consistency can be computed more efficiently than global consistency.

We find that AC-3, a classical arc consistency algorithm [Mac77a] for a network of binary constraints, can be implemented with optimal *worst case* complexity of $\mathcal{O}(ed^2)$ [ZY01] where e is the number of constraints and d the size of domain. This is surprising since AC-3 has long been considered by the community as an algorithm of complexity $\mathcal{O}(ed^3)$ which is the main result in Mackworth and Freuder's important paper [MF85]. This theoretical complexity bound complements the fact that AC-3 is empirically efficient (see the empirical work by Wallace [Wal93]). Our empirical study also shows that the new implementation of AC-3 is much faster than the traditional understanding of AC-3 in [MF85] and comparable to the state-of-the-art algorithm AC-6 [BC93]. We believe the efficiency and simplicity of AC-3 will make it continue to be a choice for empirical study and constraint systems. The implementation idea proposed here also leads to a simple path consistency algorithm with the best known worst case complexity.

To enforce arc consistency on a network of *non-binary* constraints is an NP-complete problem. We report a new class of constraints [ZY00, ZW98]—*monotonic constraints*—on which arc consistency can be enforced in $\mathcal{O}(er^3d)$ where r is the maximum arity of constraints in the network. As an example, the ubiquitous linear inequality belongs to this class. This result generalizes the work in [VHDT92].

Van Beek and Dechter [vBD95, vBD97] have identified several properties of constraint networks such that local consistency guarantees global consistency in those networks. Motivated by their work, we first find that consistency can be studied from a perspective of set intersection. We then establish a framework to relate results on set intersection to results on consistency (including a relationship between local and global consistency) in a constraint network. Under this framework, a number of new properties of constraint networks, where some level of consistency is ensured or global consistency is implied by local consistency, are identified [ZY02a, ZY03b, ZY03a]. For example, by enforcing *relational m-consistency* on a network with certain *properly m-tight* constraints, the global consistency of the network is guaranteed. This is an improvement over existing work [vBD97].

We also study a special network of functional constraints where local consistency guarantees global consistency. We find that a network of *functional constraints* can be made globally consistent in $\mathcal{O}(ed)$ [ZYJ99], the cost of an optimal AC algorithm [VHDT92]. *Variable elimination* is introduced to elegantly and efficiently solve this problem. We also propose an algorithm to solve a network, where functional constraints are incrementally added, with almost the same time as $\mathcal{O}(ed)$ [ZY02b]. An application of the variable elimination method is also exhibited to design an algorithm to make a network of *implicational constraints* globally consistent in $\mathcal{O}(e(n+d))$, where n is the total number of variables in the network. The new algorithm improves existing algorithms [Kir93, CCJ94].

1.3 Organization of the Thesis

This thesis consists of five parts. The first part contains two chapters. Chapter 1 includes a general introduction of constraint networks and our contributions. The necessary concepts and ideas in constraint networks are reviewed in chapter 2.

The second part studies the consistency as a pruning facility in a search pro-

cedure. Specifically, efficient algorithms are designed for arc consistency enforcing. The new implementation of AC-3 is presented in chapter 3. The monotonic constraints and algorithms to enforce AC on them are studied in chapter 4.

Functional constraints, a tractable CSP, are studied in part III. Chapter 5 includes a variable elimination method to solve functional constraints and its application to solve implicational constraints. Algorithms to solve an incremental network with functional constraints are proposed in chapter 6.

In part IV, we present several properties on set intersection, the relationship between consistency and set intersection, and then a framework on the relationship between local and global consistency. It is followed by several applications of the framework where various new and existing consistency results are obtained.

Part V concludes the thesis by summarizing the results reported here.

Chapter 2

Preliminaries on Constraint Networks

It has been recognized for a long time that some complex problems can be solved by generating all possible solution candidates and checking whether there is any candidate which satisfies the requirements imposed by those problems. This *search* technique may be inherent in the reasoning of a human being. However, the systematic study of search probably started after the emergence of computer science.

The breakthrough to bring search to the attention of scientists and mathematicians is the discovery of the *backtracking*, coined by D. H. Lehmer (see [GB65]). Golomb and Baumert [GB65] are among the first who formulated the method of backtracking search and realized its potential application to a wide class of problems, beyond combinatorial problems. As claimed by Golomb and Baumert, backtrack had been independently "discovered" and applied by many people. This justifies again "when the time is ripe for certain things, these things appear in different places in the manner of violets coming to light in early spring" by Wolfgang Bolyai. In order to highlight the generality of backtracking, Golomb and Baumert model a problem as one of determining the value of variables vector $(x_1, x_2, ..., x_n)$ from the space of the Cartesian product $X_1 \times X_2 \times \cdots \times X_n$ such that the value

of the vector maximizes the criterion function $\phi(x_1, x_2, \cdots, x_n)$. On the one hand, under this framework backtrack as a programming principle can be applied to a wide spectrum of problems. On the other hand, the framework is too general to facilitate further effective exploration of search.

The next propelling source of search is from Artificial Intelligence (AI). Many systems are built to solve problems arising in AI. One of the earliest is Waltz's [Wal75] system to recognize objects from line drawings.[1] In this system, a filtering algorithm is employed to avoid combinatorial explosion of *scene labeling*, assigning meaningful labels to line segments and regions in a drawing (scene). It plays a critical role in the efficiency of Waltz's system. At the same time, based on his experience in picture processing, Montanari realized that constraint manipulation is a common part shared by many problems from different fields. He introduced a network of *constraints* to model a general class of problems and defined a *binary* constraint as a relation on two variables. Solutions of a binary constraint network with n variables can be regarded as a non-binary constraint on all the n variables. Given a binary constraint network, his question is to find an equivalent binary network which is *minimal* in the sense that compared with other equivalent networks, it allows the minimal number of pairs in every constraint. However, this question is NP-hard. So, Montanari introduced a closure operation on a constraint network to obtain an approximation of the minimal network. To some extent the closure of a network is closer to the non-binary constraint implied by the original network. For several problems the closure simply results in the non-binary relation. In other words, the non-binary relation can be efficiently computed from the closure.

Motivated by Waltz's filtering algorithm and the systems using similar techniques (e.g. [Fik68]), and Montanari's closure operation, Mackworth [Mac77a] proposed the following unified framework. First, the task from different areas is

[1] This problem is NP-complete [KP88].

modeled as a constraint satisfaction problem. In the following excerpt, a predicate means a constraint.

> The task specification is formulated to consist of a set of variables, each of which must be instantiated in a particular domain, and a set of predicates that the values of the variables must simultaneously satisfy.

Second, the concept of consistency is introduced to overcome the thrashing problem in a backtracking search. The insight behind the consistency is that the search space should be pruned by removing those inconsistent values or combination of values which will never be a part of a solution. Although his arc consistency and path consistency are from Waltz's filtering and Montanari's closure operation respectively, Mackworth's introduction of different levels of consistencies has greatly motivated the research in CSP (e.g.[Fre78]).

Under the framework of CSP, backtracking search has been studied extensively. For example, backjumping [Gas79, SS77], constraint recording [Dec90b] and conflict directed backjumping (CBJ) [Pro93] have been proposed to improve backtracking. The search efficiency can be significantly improved by exploiting various heuristics like the variable ordering [GB65, HE80] and value ordering.

The power of the CSP model and techniques is not fully exploited until a major breakthrough in Logic Programming. In 1987, Jaffar and Lassez proposed the Constraint Logic Programming (CLP) scheme CLP($\mathcal{X}$) which elegantly combines logic programming and a constraint domain $\mathcal{X}$. Under this scheme, a logic program is regarded as a dynamic generator of constraints in $\mathcal{X}$ and the satisfiability (and a solution) of the constraints is tested (and found) by a constraint solver embedded in the CLP system. Naturally, the CLP scheme provides a programming interface for CSP, and techniques developed for CSP can be easily employed by the solver. CHIP [VH89] was one of the most influential CLP [JL87] over finite domain CSPs.

The CLP combined with CSP has been a great success in industrial applications

[VH89, JM94, ILO00]. It benefits from the efficient techniques developed for solving CSP, the modeling ability of CSP, the flexibility of the host languages, and the ability of the seamless and natural embedding of CSP into the host languages.

2.1 A Review on Sets and Graphs

We recall some notations and vocabularies from sets and graphs before we give a detailed description of a constraint network. They are only used to facilitate exposition in this thesis and thus are presented in an intuitive, rather than formal, way.

Given any two sets A and B, we use $A \cup B$, $A \cap B$ and $A - B$ to denote the union, intersection and difference of A and B respectively. $A - B$ is the set of elements which are in A but not in B.

The *Cartesian product* of A and B, denoted by $A \times B$, is

$$A \times B = \{(a, b) \mid a \in A, b \in B\}.$$

A relation c on sets $D_1, \cdots, D_k$ is a subset of the Cartesian products $D_1 \times D_2 \times \cdots D_k$. c is a *universal relation* if $c = D_1 \times D_2 \times \cdots D_k$.

A directed graph is a set of vertices (or nodes), and a set of arrows (or arcs), with each arrow joining one vertex to another. It is denoted by a tuple (V, E) where $V = \{x_1, \cdots, x_n\}$ and E is a subset of $V \times V$. An arrow has a direction and is usually denoted by a tuple (x_i, x_j). If we replace the arrow in a directed graph by an undirected edge, the graph becomes an undirected graph. An edge is denoted by a set $\{x_i, x_j\}$ where the order of the vertices doesn't matter.

An edge is *incident* to a vertex if the vertex is one end of the edge. An arc is *incident* to a vertex if the vertex is the ending vertex of the arc. A *neighbor* of a vertex is a vertex in the graph such that there is an edge (or arrow) between

them. A *walk* in a graph is a sequence of alternative vertices and edges (or arrows) where each edge (or arrow) joins the vertices before and after it. It is denoted by $x_i x_{i+1} \cdots x_j$ where the edges are usually omitted if there is only one edge (or arrow) between two vertices. If there are no repeated edges in a walk, the walk is called a *path*. A *simple path* is a path without repeated vertices. The *reverse* of a path is the reversed sequence of the path. For example, the reverse of $x_i x_{i+1} \cdots x_j$ is $x_j \cdots x_{i+1} x_i$. A *closed walk* is a walk whose starting vertex is the same as its ending vertex. A *circuit* (also called *cycle* or *loop*) is a closed walk without repeated edges in it.

A *complete* graph is one where there is an edge between any pair of vertices.

A graph is *connected* if for any two vertices there is a walk between them.

An *acyclic directed graph* is a graph which contains no cycle.

A *tree* is a connected undirected graph without any circuit. For a tree, we know that *between any two nodes there is a* unique *simple path*. A tree is denoted by a tuple (T, E). (T_1, E_1) is a *subtree* of (T, E) if it is a tree, $T_1 \subseteq T$, and $E_1 \subseteq E$. Usually, after we designate a node in a tree as the *root*, the tree is called a *rooted tree*. The *level* of a node in the tree is defined as the length of the path from the node to the root. Now we can distinguish the nodes in a rooted tree. x_1 is the parent of x_2 if x_1 is a neighbor of x_2 and its level is lower than x_2's. x_2 is also called a child of x_1.

2.2 Constraint Satisfaction Problem

Most definitions of constraint satisfaction problem used in the literature follow the conventions by Mackworth [Mac77a] and Montanari [Mon74]. Constraint network, another terminology equivalent to constraint satisfaction problem, is also frequently used. They are usually interchangeable. However, in this thesis we differentiate them following the definition of a *problem* from the NP-complete literatures [GJ79].

A CSP is a problem whose instance is a constraint network and the question is to find an assignment of values to variables such that all constraints in the network are satisfied simultaneously.

A *constraint network* $\mathcal{R}(N, D, C)$ is defined as a set of variables $N = \{x_1, x_2, \cdots, x_n\}$; a set of finite domains $D = \{D_1, D_2, \cdots, D_n\}$ where domain D_i, for all $i \in 1..n$, is a set of values that variable x_i can take; and a set of constraints $C = \{c_{S_1}, c_{S_2}, \cdots, c_{S_e}\}$ where S_i, for all $i \in 1..e$, is a subset of $\{x_1, x_2, \cdots, x_n\}$ and each constraint c_{S_i} is a relation defined on the domains of all variables in S_i. The *arity* of constraint c_{S_i} is the number of variables in S_i. Throughout this thesis, n denotes the number of variables, r the maximum arity of the constraints in the network, d the size of the largest domain, and e the number of constraints in C in a constraint network. We list in Appendix A the convention of the symbols frequently used in this thesis.

A *solution* of a network is an assignment of values to variables so that all the constraints in the network will be satisfied by the assignment.

A *constraint satisfaction problem* is a problem whose instance is a constraint network and whose question is to find a solution of the constraint network. A CSP is *satisfiable* if its network has a solution. The *solution space* of a CSP, a relation on all variables in N, is the set of all solutions. Two CSPs (and CNs) are *equivalent* if and only if they have the same solution space.

CSPs are abundant in computer science and specially in Artificial Intelligence [Mac92], and Operations Research [NW88].

Example. The graph k-colorability problem is whether k colors are sufficient to color the nodes of a graph such that no two neighbors have the same color. It can be easily cast into a CSP as follows. The variables $\{x_1, x_2, \cdots, x_n\}$ are to represent the nodes of the graph, all the variables share the same domain $\{$ $\text{color}_1, \text{color}_2, \cdots, \text{color}_k$ $\}$ which consists of all the colors available, and the constraints are that for all $i, j \in 1..n$, $x_i \neq x_j$ if there is an edge between x_i and x_j.

The CSP is to find whether there is a solution for this network. □

In the problem above, note that every constraint involves at most two variables. *Binary constraint network or binary CSP* is specially used to denote this class of problems. A binary constraint c_{ij} denotes the constraint between the variables x_i and x_j. For the problem of interest here, we require that $\forall a, b \quad a \in D_i, b \in D_j, (a, b) \in c_{ij}$ if and only if $(b, a) \in c_{ji}$. In other words, c_{ij} and c_{ji} are understood as one constraint. When c_{ij} and $c_{ji}^{-1} = \{(a, b) \mid (b, a) \in c_{ji}\}$ are different, an intersection of these two constraints (relations) will result in *one* relation on i and j. Most early work on CSPs stems from the study of binary constraint networks. A binary constraint network can be naturally represented by an undirected or directed graph. An undirected graph induced by a constraint network $\mathcal{R}(N, D, C)$ is $G = (V, E)$ where $V = N$ and $E = \{\{x_i, x_j\} \mid \exists c_{ij} \in C\}$. The (topological) structure of the graph representation of a constraint network has motivated a lot of interesting work on CSP (see [Dec92a]).

2.2.1 Representation of Constraints

In this subsection, we restrict our attention to only binary CSP. Constraint plays a central role in the model of constraint network. It is necessary to make a constraint as concrete as possible so that they can be manipulated.

A constraint on variables x_i (with finite domain D_i) and x_j (with finite domain D_j) will restrict the values that x_j can take when x_i take some value. Naturally, a constraint is simply defined as a relation on D_i and D_j by Montanari [Mon74]. $(a, b) \in c_{ij}$ implies that when x_i and x_j are assigned values a and b respectively, the constraint between i and j is satisfied.

Now, we can immediately apply set operations like *intersection* and *union* to constraints. Most importantly, we can *compose* two constraints c_{ij} and c_{jk} in the

following way to get a new constraint between i and k:

$$c_{jk} \circ c_{ij} = \{(p, q) \mid \exists r \in D_j, (p, r) \in c_{ij} \wedge (r, q) \in c_{jk}\}.$$

Following the convention used in composing functions, we define composition of sets as a right associative operation.

We can assume there is only one constraint c_{ij} between x_i and x_j because if there is more than one constraint we can simply take their intersection as the final constraint between x_i and x_j under the relation model.

In addition to the set representation, a relation can also be typically represented by a matrix. This representation is very useful in understanding some of the results in CSP and thus we include it here. The rows of the matrix are indexed by the values of one domain and the columns by those of the other domain. The entry is a boolean value to indicate whether the tuple of row index and column index is allowed by the constraint. The composition of two constraints can be computed by the multiplication of their matrices.

Example. Let the domain of variable i be {John, Allan, Peter }, the domain of j { short, tall }, and the constraint $c_{ij} = \{(John, short), (Allan, tall), (Peter, short)\}$.

$$c_{ij} = \begin{array}{c} \\ \text{John} \\ \text{Allan} \\ \text{Peter} \end{array} \begin{array}{c} \begin{array}{cc} \text{short} & \text{tall} \end{array} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array}$$

There are cases where other form of representation of a constraint is used more conveniently and intuitively. As an example, the arithmetic expression of a constraint composes of variables, integers, operations (*addition, minus, multiplication,* and *division*) and relations $=, \leq, \neq$. For example,

$$i^2 + j^2 \neq 100$$

where the domain for i and j are the integer interval [1..20]. The arithmetic constraints are so important that an independent field of mathematical programming, especially (integer) linear programming, is devoted to this topic. [Dan63, NW88] are good references for this topic.

There are also some constraints frequently arising from real life CSP problems. They are taken as primitives in CSP systems. For example, the *all different* constraint states that all the variables of concern should take different values. The *cardinality* constraint is proposed [VHD91] to represent non-primitive constraints met in real life application.

2.2.2 Solving a CSP

Given the finiteness of the domain for each variable, it is always possible in principle to find a solution for a CSP if it exists. In order to get an assignment of variables from respective domains, we simply check all possible assignments exhaustively to see whether there is any assignment satisfying all the constraints simultaneously. This paradigm is called generate and test in logic programming [VH89]. The backtracking paradigm is an improvement over generate and test. In this paradigm, variables are instantiated one by one. After each instantiation of a variable, all the constraints involving instantiated variables will be checked. If some constraint is not satisfied, we stop instantiating the rest variables because the constraint is still violated no matter how to instantiate them. In other words, a portion of the search space is pruned. A new value will be chosen for the current variable. If no more values for the variable satisfy the related constraints, *backtracking* occurs. It goes back to a previous variable and chooses a new value for it. The process will be repeated until a solution is found or there is no choice of value for the first variable. Given a CSP (V, D, C), an illustrative algorithm for backtracking paradigm is shown in the next section.

Example. The N queens problem is to put N queens on an N by N chess board so that no two queens attack each other, that is, no two queens are on the same row, the same column, or the same diagonal. Here we consider the 5 queens problem.

We model the problem in the following way. Since no two queens can be put into the same column, there is only one queen in each column. For all $i \in 1..5$, the variable x_i means the row occupied by the queen at column i. The domain of any variable will range from the first row to the fifth row, that is $\{1, 2, \cdots, 5\}$. The row constraint that no two queens are on the same row is translated into $x_i \neq x_j$ for all $i, j \in \{1, 2, \cdots, 5\}$ such that $i \neq j$; the diagonal constraint that no two queens are on the same diagonal into $|x_i - x_j| \neq |i - j|$ for all $i, j \in \{1, 2, \cdots, 5\}$ such that $i \neq j$.

A generate and test search to this problem is to systematically assign possible values to five variables simultaneously and then check if the assignment satisfies all the constraints imposed. For example, we generate a candidate solution by the following way. First we try an assignment $(1, 1, 1, 1, 1)$ for variables from x_1 to x_5. It violates the row constraint. Next we try $(1, 1, 1, 1, 2)$ and it still fails the row constraint. After about 359 trials, we hit a solution $(1, 3, 5, 2, 4)$.

Although $x_1 = 1$ and $x_2 = 1$ violate the row constraint, the generate and test method still tries to instantiate x_3 to x_5 with all the possible combinations. This drawback is overcome by backtracking search in the following way. Let's instantiate the variables in the order of $x_1, x_2, \cdots, x_5$. First let $x_1 = 1$ and no constraint is violated. Next, let $x_2 = 1$ and the row constraint is violated. So, we choose 2 for x_2 which fails the diagonal constraint. 3 will be a choice for x_2. Continue this process, after 15 trials we hit a solution $(1, 3, 5, 2, 4)$. We are lucky here and there is no backtracking taking place. $\square$

2.3 Consistency Techniques

As shown in the previous section, to find a solution for a CSP is not difficult in principle. Given the NP-completeness of the CSP, to find a polynomial algorithm to solve it may be futile. Most efforts in the CSP community are devoted to improving the backtracking algorithms.

One observation, which is made as early as in 1965 [GB65], is that the domains of uninstantiated variables can be shrunk by *precluding* those values which are not compatible with the instantiated variables according to some constraints. The preclusion of the values is also called an active use of constraints to prune the domain of uninstantiated variables. By employing a similar idea, Waltz [Wal72] successfully managed the combinatorial explosion of the backtracking search.

At roughly the same time, Montanari conducted a theoretical study on the constraint processing in picture processing [Mon74]. He found that certain binary constraint networks can be solved (or otherwise approximated) by introducing a closure operation on constraints.

Mackworth quickly realized the importance of Waltz and Montanari's work and unified the preclusion and closure operation as different levels of consistencies, that is *arc consistency* and *path consistency*. His work greatly promoted research both in finding practically efficient search algorithms through achieving different levels of consistency [HE80], and in understanding constraint solving through the concept of consistency [Fre82].

Example. Consider again the 5 queens problem. The backtracking search can further be improved by preclusion. Let $x_1 = 1$. Immediately we know the no queens can be put at the first row, the first column, or the diagonal as shaded in Fig 2.1. Now, let $x_2 = 3$, the first place available for x_2. Squares in the third row, the second column, and the two diagonals passing x_2, will be shaded (see Fig 2.2). Now, the choices for other variables are almost unique. Here we make at most six

choices of values for variables in total before a solution is found. □

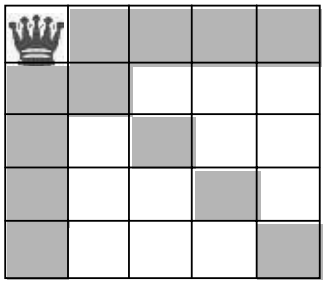


Figure 2.1: The chess board after putting a queen on the first column

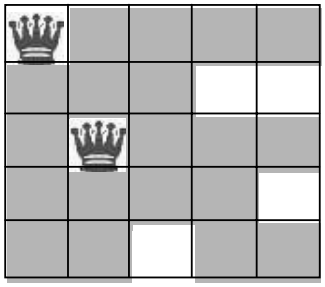


Figure 2.2: The chess board after putting a queen on the second column

The idea of the preclusion can be generalized to the arc consistency in a binary network.

Definition 1 *Given a binary constraint network (N, D, C). A constraint c_{ij} is* consistent *with respect to i if and only if $\forall a \in D_i$, there exists $b \in D_j$ such that $(a, b) \in c_{ij}$. c_{ij} is consistent if it is consistent with respect to both i and j. The network is* arc consistent *if and only if every constraint in the network is consistent.*

Note in the above definition, to check the consistency of a constraint, we need to check two directions: from i to j and from j to i. In this consideration, the undirected edge in the associated graph of a network is better represented by two arrows. The name of arc consistency is from the fact that a constraint should be

consistent along every arrow (also called *arc* in graph texts) in the directed graph. Now the definition above can be simplified as follows:

Definition 2 *Given a binary constraint network (N, D, C) and its associated directed graph (V, E). An arc $(i, j) \in E$ is consistent if and only if $\forall a \in D_i$, there exists $b \in D_j$ such that $(a, b) \in c_{ij}$. The network is arc consistent if all arcs are consistent.*

Later, Freuder [Fre78] generalizes arc consistency and path consistency to k-consistency. To define k-consistency, we need the following notations. An *instantiation* $\bar{a} = (a_1, \cdots, a_j)$ of variables $Y = \{x_1, \cdots, x_j\}$ is to assign a value a_i in D_i to variable x_i for all $i \in 1..j$. An *extension* of $\bar{a}$ to a variable $x(\notin Y)$ is denoted by $(\bar{a}, u)$ where $u \in D_x$. An instantiation of Y is *consistent* if it satisfies all constraints in $\mathcal{R}$ which don't involve any variable outside Y. For example, in the five queens problem, the instantiation $(1, 3)$ of x_1 and x_2 satisfies all constraints on x_1 and x_2 (other constraints involving one of x_1 and x_3 are ignored). So, instantiation $(1, 3)$ of $\{x_1, x_2\}$ is consistent. The following definition of k-consistency is on a general constraint network.

Definition 3 *A constraint network $\mathcal{R}$ is* k-consistent *if and only if for any consistent instantiation $\bar{a}$ of any distinct $k-1$ variables, and for any new variable x, there exists $u \in D_x$ such that $(\bar{a}, u)$ is a consistent instantiation of the k variables. $\mathcal{R}$ is* strongly *k-consistent if and only if it is j-consistent for all $j \leq k$. A strongly n-consistent network is called* globally consistent.

The intuition behind this definition is that a certain level of consistency in a network implies that any consistent instantiation of some variables can be extended to a new variable. Node consistency is 1-consistency, arc consistency is 2-consistency, and path consistency is 3-consistency. Typically a given constraint network is not k-consistent even for small k. *Consistency-enforcing* algorithms are employed to

achieve a certain level of consistency on the constraint network so that those partial instantiations not extensible to a new variable will be removed.

Local consistency is also used in this thesis to denote k-consistency when $k < n$.

Now, a backtracking search with consistency-enforcing algorithm is given below.

```
algorithm Search((N, D, C))
  begin
    i ← 0; backtracking ← false;
    while i < n do //exsits a variable not assigned yet
      if not backtracking then
        Choose a variable x_j from N − {v_0, ···, v_{i−1}};
        v_i ← x_j; //v_i is the current variable
        S_i ← D_j; //S_i is the current domain
      endif
      backtracking ← true;
      while S_i is not empty do // search a value for current var v_i
        choose a value a for v_i;
        S_i ← S_i − {a};
        enforce certain level of consistency on the network;
        if the domain of some variable is empty then
          restore the domains of variables N − {v_0, ···, v_i};
        else //a is a valid value for v_i
          backtracking ← false;
          break;
        endif
      endwhile
      if backtracking then
        i ← i − 1; // backtrack to the previous variable
        if i < 0 then  break;  endif
      else i ← i + 1; // progress to the next variable
    endwhile
    if backtracking then  report unsatisfiability;
    else report the solution;
  end
```

Figure 2.3: A search procdure with consistency enforcing for constraint networks

In practice, it is typical to use the low level consistencies to improve the efficiency of a search algorithm. The higher level consistencies are mainly used to study

the properties of certain constraint networks. For example, it is interesting to identify the situations when the local consistency is sufficient to solve a CSP globally. A wealth of results from the early work of Montanari and Freuder [Mon74, Fre82] to more recent work such as van Beek and Dechter [vBD97], have been obtained. They enhanced our understanding of constraint solving. More discussion on this topic is in Chapter 7 on set intersection and consistency.

2.4 On the Model of CSP

The key techniques to solve CSP have proven to be useful in widely different fields from AI to Operations Research and even to Numerical Analysis. As a result many new variations of CSP are identified and studied. Among others there are numerical CSP [Lho93], continuous CSP [Fal94], and temporal CSP [DMP91].

There are two models in computer science which are very close to the model of CSP. The first one is the backtracking programming by Golomb and Baumert [GB65]. Their intention is to make the backtracking so general that it is applicable to as many applications as possible. So, their model is to find an assignment of n variables $\{x_1, x_2, \cdots, x_n\}$ with finite domains to maximize a criteria function $\phi(x_1, x_2, \cdots, x_n)$. The constraint is absent from their model while there is a criteria function. Because of the generality of this model, there is little research done on this model, although the necessity of pruning search space and choosing a good variable to instantiate was realized in [GB65]. In contrast, by introducing constraints explicitly, the CSP model, together with the progress in other fields of AI and programming languages, has motivated a lot of research on finding efficient search algorithms and heuristics. Thanks also to the introduction of constraints, deeper understanding of constraint solving is obtained [Fre82, vBD97, JCG97].

The other is the relational model of data in database, which includes a set of relations on variables (attributes) with finite domains, and a relational algebra

[Cod70, RG00]. Abstractly, the relational model of data is exactly a constraint network. This leads to interesting interactions [Dec90a, Mon74, Var00] between the studies of database and CSP, despite the apparent difference between the purposes of database and CSP. An example to show the relationship between CSP and relational database is the following. If we take the solutions of a CSP as a relation on all variables, the relation can be simply obtained by a *natural join* [Cod70] of constraints (*relations* in database term) in the network. Benefiting from the relational algebra, the *bucket elimination* proposed by Dechter [Dec99] makes a heavy use of the join operation and has some applications in CSP and belief networks. It is interesting to compare the join operation and the k-consistency proposed by Freuder [Fre78].

Part II

Consistency as Pruning in Search

Arc consistency and its variations have been accepted as an effective way to prune the search space. Since they are frequently called by a search procedure to solve a CSP, it is necessary to explore the most efficient way to enforce arc consistency on a network. In the first chapter of this part, we re-examine the simple and widely used AC-3 algorithm by Mackworth, and propose a new algorithm AC-3.1 which is comparable to the state-of-the-art algorithms both theoretically and experimentally.

Traditionally is studied in binary constraint networks. In the last decade, a lot of effort in CSP community has focused on non-binary networks as the CSP techniques have been finding more and more applications which naturally involve non-binary constraints. The much higher cost of general arc consistency algorithms in the non-binary setting imposes more challenges on the design of efficient algorithms. In the second chapter of this part, we study the arc consistency on a special while applicable class of constraints, *monotonic constraints and linear arithmetic constraints*. Efficient algorithms are also presented.

Chapter 3

A New Arc Consistency Algorithm

In this chapter, we confine our discussion to *binary* constraint networks. Since Waltz's successful application of arc consistency (AC) to solve problems in understanding line drawings, there have been many algorithms developed to improve the efficiency of arc consistency. Among them are AC-3 [Mac77a], AC-4 [MH86], AC-6 [Bes94] and AC-7 [BFR99]. The AC-3 algorithm was proposed by Mackworth in 1977 [Mac77a]. Its worst case complexity was not known until Mackworth and Freuder carried out an analysis in 1985 [MF85] which states that the complexity of AC-3 is of $\mathcal{O}(ed^3)$, where e is the number of constraints and d the size of largest domain. Because of its great impact on the study of CSP, this result itself has been deeply rooted in the CSP community (e.g. [Wal93, BFR99]). Therefore AC-3 is typically considered to be non-optimal. With time complexity $\mathcal{O}(ed^2)$, other algorithms such as AC-4, AC-6, AC-7 are considered worst case optimal. As far as we are aware, there has not been any result showing that AC-3 can be implemented with optimal worst case time complexity.

We show that AC-3 does achieve worst case optimal time complexity of $\mathcal{O}(ed^2)$ under a proper implementation [ZY01]. It is a bit surprising since AC-3 is a coarse

grained *arc revision* algorithm [Mac77a] while the known optimal algorithms are all based on fine grained *value revision*. Experiments are also conducted to examine the practical efficiency of the new AC-3 implementation. On easy CSP instances, it is comparable to the traditional implementation of AC-3 which is known to be substantially better than the optimal fine grained algorithms. On hard instances like those from the phase transition, it is significantly better than the traditional AC-3 and is comparable to the best known algorithms such as AC-6 in terms of running time.

It is also found that the idea behind the new AC-3 implementation immediately leads to a new path consistency algorithm which has the same theoretical time and space complexity as the best known ones [Sin96].

AC-3 is re-examined also for other reasons. It is one of the simplest AC algorithms, and is known to be practically efficient [Wal93]. The simplicity of arc revision in AC-3 makes it convenient for implementation and amenable to various extensions for many constraint systems. Thus while AC-3 is considered as being sub-optimal, it often is the algorithm of choice and can outperform other theoretically optimal algorithms.

Techniques to enforce arc consistency are reviewed in section 3.1 before a formal analysis of traditional AC-3 is presented in section 3.2. The new AC-3 algorithm and its complexity analysis are presented in section 3.3, and an algorithm on path consistency is proposed in section 3.4. Experimental results on AC-3 and other arc consistency algorithms are listed in section 3.5. A comparison of AC-3 with other algorithms is discussed in section 3.6. Section 3.7 concludes this chapter.

3.1 Techniques to Enforce Arc Consistency

This section serves to give an intuition of techniques employed in the evolution of arc consistency algorithms.

3.1.1 Arc Consistency

As discussed in the preliminaries (Section 2.2), we know that it is convenient to associate a binary constraint network with a graph. Knowing that a constraint (relation) can also be visualized by a graph, we also introduce the following notation.

Definition 4 *The* value based constraint graph *of a network (N, D, C) is G=(V, E) where* $V = \{i.a \mid i \in N, a \in D_i\}$ *and* $E = \{(i.a, j.b) \mid (a, b) \in c_{ij}\}$. *A more rigorous name for the traditional constraint graph may be* variable based constraint graph.

Notation. In the following presentation, variable x_i is sometimes replaced, for simplicity and clarity, by its subscript i when there is no confusion. In this case, we use i or j to denote a variable. Small letters a or b will be used to represent a value in the domain of a variable. $i.a$ (or $x.a$) is used to denote a value $a \in D_i$ (or $a \in D_x$).

In this section, the value based graph is used in our drawing to illustrate a constraint network. Note that for simplicity undirected edges are used in the drawing. Each undirected edge (e.g. $\{i.a, i.b\}$) should be understood as an edge with two directions ($(i.a, i.b)$ and $(i.b, i.a)$). □

We begin with some basic concepts underlying an arc consistency algorithm through the following example.

In the network shown in Fig 3.1, there are three variables $\{x, y, z\}$ whose domains are $\{1, 2, 3\}$. The constraint between x and z is a special one while all the others are the *identity* relation where each value of a variable is related to the *same* value in the domain of the other variable. A DOMINO problem is a generalization of this example to n variables $\{x_1, x_2, \cdots, x_n\}$ and d values in the domain $\{1, 2, 3, \cdots, d\}$. The special constraint, called *trigger constraint*, is on x_1 and x_n. It is defined as $c_{1n} = \{(d, d)\} \cup \{(x, x+1) \mid x < d\}$. For all $i < n$, there is an identity constraint on i and $i + 1$

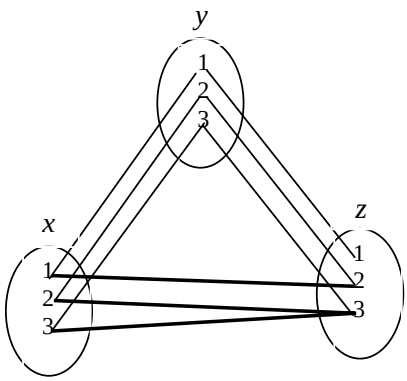


Figure 3.1: Example of DOMINO problem

A close look at the network shows that the value 1 in the domain of z is not related to any value in x. The value $z.1$ is *invalid* or *not supported*. The value $z.2$ and $x.1$ satisfy c_{zx}. $x.1$ is called a *support* of $z.2$. According to the Definition 2 of arc consistency, the arc (z, y) is *not* consistent. It is easy to verify that arcs $(x, y), (y, x), (y, z), (z, y)$, and (x, z) are consistent respectively.

In order to make the network arc consistent, we need to enforce the arc (z, x) consistent. Removing $z.1$ from D_z is sufficient! A careful reader may realize that the removal of $z.1$ will make the arc (y, z) no longer consistent since $y.1$ is not supported now. It is not hard to see that the removal of $z.1$ result in an domino effect on the values of domains of all variables until each domain has only one value 3 (d in the general DOMINO problem) left. A similar effect may occur in enforcing arc consistency on any constraint network. A more general term in the AI community to describe this effect is *constraint propagation*. An arc consistency algorithm needs to iteratively inspect each arc and make it consistent if necessary, until all arcs are consistent. Intuitively such an algorithm terminates finally. It is because some value(s) will be removed each time a constraint is made arc consistent and the total number of values in the network is finite.

3.1.2 AC-3

The question now is how to inspect arcs after some values are removed. There are two main approaches. The first is simply re-inspect the whole constraint network whenever a value is removed. This idea is reflected in the algorithm AC-1 [Mac77a]. It is not very interesting under this context and not discussed in detail although it is an interesting algorithm in the parallel computing. The other one is re-inspect only those constraints involving the variable whose value(s) have been removed. Specifically, if some values of x are removed, only arcs incident to x would be re-inspected. AC-3 embeds this idea and some other minor considerations [Mac77a]. The algorithm of AC-3 will be given in Section 3.2.

Consider the DOMINO network (Fig 3.1) again. $z.1$ is removed when (z, x) is inspected. No matter whether they have been inspected before, (y, z) and (x, z) will be re-inspected since the removed value may be a support of some value in y (and x respectively) with respect to (y, z) (and (x, z) respectively). However, (z, y) need not be inspected at this stage because the removal of values in D_z does not affect its consistency. A further observation made by Mackworth [Mac77a] is that (x, z) needs not be re-inspected either. since all the removed values in z are not supports of any value in x (e.g. z.1). One minor advantage of AC-3 claimed in [Mac77a] is that arcs like (x, z) will not be re-inspected.

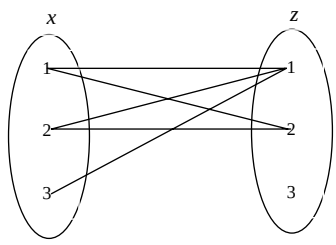


Figure 3.2: Example for algorithm AC-4

3.1.3 AC-4

Immediately after the publication of the complexity results on AC-3, Mohr and Henderson [MH86] designed a worst case optimal algorithm, named AC-4 by them. The key idea behind AC-4 is that when a value of a variable is removed, only some, *not all*, values in the domain of its neighbor variable will be re-inspected. They are exactly those for which the removed value provided support. Consider the constraint in Fig 3.2.

Assume $z.1$ is removed in the inspection of a relevant constraint. We know that the affected values in x will be $1, 2, 3$ since $z.1$ supports each of them. So, a list SUPPORT($z.1$, x) is used to store all values that $z.1$ supports, i.e. SUPPORT($z.1$, x) $= \{1, 2, 3\}$. In a network, for any arc (i, j) and $a \in D_i$, we establish a list SUPPORT($i.a$, j).

Now a question is how to know whether each value in SUPPORT($z.1$, x) (say $x.1$) is still valid after $z.1$ is removed. One way is to remove $z.1$ from SUPPORT($x.1$, z) and then check if SUPPORT($x.1$, z) is empty. Its emptiness means that $x.1$ no longer supports any value in z and is thus not supported, resulting in the removal of $x.1$. Noting that what we need is whether $x.1$ has a support but not what are the supports of $x.1$, Mohr and Henderson just use an extra structure COUNTER($x.1$, z) to denote the number of supports of $x.1$ (or equivalently the number of values in z that $x.1$ supports). Now, if $z.1$ is removed, it is sufficient to decrement COUNTER($x.1$, z), and remove $x.1$ if it is zero. Obviously, we need a COUNTER structure for each arc $(i, j) \in E$ and each $a \in D_i$.

To establish the SUPPORT($x.1$, z) and COUNTER($x.1$, z), a traditional way in the CSP community is to search the supports from $z.1$ to $z.3$ in turn since for a practical AC algorithm, a constraint is not always so explicit as in our drawing. After scanning all elements in D_z, we have SUPPORT($x.1$, z) $= \{1, 2\}$ and COUNTER($x.1$, z) $= 2$ since $x.1$ has two supports $z.1$ and $z.2$. In the same manner,

we have

$$\text{SUPPORT}(x.2,\ z) = \{1, 2\}, \qquad \text{COUNTER}(x.2,\ z) = 2;$$

$$\text{SUPPORT}(x.3,\ z) = \{1\}, \qquad \text{COUNTER}(x.3,\ z) = 1;$$

$$\text{SUPPORT}(z.1,\ x) = \{1, 2, 3\}, \quad \text{COUNTER}(z.1,\ x) = 3;$$

$$\text{SUPPORT}(z.2,\ x) = \{1, 2\}, \qquad \text{COUNTER}(z.2,\ x) = 2;$$

$$\text{SUPPORT}(z.3,\ x) = \{\}, \qquad \text{COUNTER}(z.3,\ x) = 0.$$

After this initialization, we know that $z.3$ will be removed. Now assume $x.3$ is removed as a result of the initialization of some other relevant constraint on x. SUPPORT($x.3$, z) = $\{1\}$ tells us that $x.3$ is a support of $z.1$. So, the removal of $x.1$ will decrease the number of supports of $z.1$ by one, i.e. COUNTER($z.1$, x) = 2. By dint of the SUPPORT structure, we avoid checking the value of $z.2$ since it has nothing to do with $x.3$!

Remark. The minor trick in AC-3 (Section 3.1.2) is also applicable to AC-4. Consider a constraint c_{xz}. Let $z.a$ be removed because of the removal of a value in x. We need not inspect the values in SUPPORT($z.a$, x) because they all have been removed (so that $z.a$ is removed).

The clever way of AC-4 to inspect only the affected values leads to an optimal worst case complexity of $\mathcal{O}(ed^2)$. The initialization of SUPPORT and COUNTER takes $\mathcal{O}(ed^2)$. The SUPPORT structures of all values with respect to all relevant arcs are exactly the value based constraint graph. In the propagation phase to remove invalid values, each directed edge (for example $(x.1, z.1)$) in the value based constraint graph will be examined at most once. So the propagation phase has a time complexity of the number of edges in the value based graph. In summary, the final complexity is $\mathcal{O}(ed^2)$ and thus optimal because it needs $\mathcal{O}(ed^2)$ to check whether a network is arc consistent. Now it seems to be the time to conclude the story of AC algorithms. However, there are more stories to tell [MF93] on the efficiency of AC algorithms for general constraints. They are shown in the following subsections.

3.1.4 AC-6

Through his experimental study of AC algorithms, Wallace shows that AC-4 always has a poor performance [Wal93]. It is observed that in AC-4 the SUPPORT structures are expensive to construct while most of them are not used in later stages because for an ordinary network only a limited number of values will be removed finally. Bessiere remedies this problem through a simple but nice observation [BC93, Bes94]. Recall that the purpose of inspecting the consistency of an arc is to justify each value in one variable by finding a support in the other variable. It is enough to find *one* support for a value. Another support will be looked for only after the current one becomes invalid.

Consider the previous example in Fig 3.2 again. To know whether $x.1$ is valid, we find a support $z.1$ for it. We then proceed to $x.2$, and find a support $z.1$ for $x.2$. Similarly we find a support $z.1$ for $x.3$. Now if $z.2$ or $z.3$ is removed, nothing needs to be done with respect to (x, z) because we know that $z.1$ is the key value to support the values in x as discovered in the previous process. What to do if $z.1$ is removed? We need to know which values in x depend on it. It can be easily done in the previous process by recording them. When justifying $x.1$, we find $z.1$ and record lightSUPPORT($z.1$, x) $= \{1\}$ so that a new support for $x.1$ will be looked for once $z.1$ is removed. After inspecting $x.2$ and $x.3$, lightSUPPORT($z.1$, x) $= \{1, 2, 3\}$ while the lightSUPPORT structures for $z.2$, $z.3$ are empty. When justifying values of domain of z with respect to (z, x), we have lightSUPPORT($x.1$, z) $= \{1, 2\}$ and there is no support for $z.3$. So, $z.3$ will be removed after the inspection of (z, x).

Now let $z.1$ be removed because of some other constraint on z. It means we need to find a new support for all those values in lightSUPPORT($z.1$, x), currently supported by $z.1$ with respect to (x, z). For $x.1$, we find $z.2$ and let lightSUPPORT($z.2$, x) $= \{1\}$. By inspecting $x.2$ and $x.3$, we have lightSUPPORT($z.2$, x) $= \{1, 2\}$,

and remove $x.3$ because there is no support for $x.3$. Note that in searching new support for $x.1$, we just continue from the one after $z.1$, not from the scratch.

AC-6 is used by Bessiere to name this algorithm. It is clear that AC-6 only does the necessary work, saving a lot of effort compared with the SUPPORT structures in AC-4. Some readers may be confused by the gap of numbers in the name of AC-4 and AC-6. AC-5 [VHDT92] is not discussed here because technically it makes use of both arc propagation (like AC-3) and value propagation (like AC-4) and serves other purposes. AC-2 [Mac77a] is also omitted here since it is similar to AC-3.

3.1.5 Bidirectionality

We conclude this section by illustrating bidirectionality. AC-7 [BFR99] is essentially an AC-6 equipped with the ability to make use of bidirectionality. *Bidirectionality* means that given a constraint c_{ij}, a value of x_i and its support in x_j support *each other*. In fact we have made use of it implicitly in previous presentation where we do not distinguish *support* and *supported*. Consider the example in Fig 3.3.

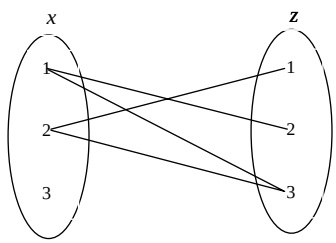


Figure 3.3: Example for bidirectionality

In justifying values in x according to constraint (x, z), AC-6 establishes

$$\text{lightSUPPORT}(z.1,\, x) = \{2\},$$
$$\text{lightSUPPORT}(z.2,\, x) = \{1\},$$

and removes $x.3$. Bidirectionality comes in when justifying values in z with re-

spect to (z, x). When we try to justify $z.1$, lightSUPPORT($z.1$, x) shows that $x.2$ is a support. After checking the validity of $x.2$ (because the inspection of another constraint may remove $x.2$), we set lightSUPPORT($x.2$, z) = $\{1\}$ as in AC-6. In the same way we find a support $x.1$ for $z.2$ and set lightSUPPORT($x.1$, z) = $\{2\}$. Since lightSUPPORT($z.3$, x) is empty, we have to search a support for it from scratch (or where we stop in a previous such search) and find $x.1$. Now lightSUPPORT($x.1$, z) = $\{2, 3\}$. Assume $x.1$ is removed by some other constraint on x. We need to find supports for $z.2$ and $z.3$, and we always try to find supports from lightSUPPORT($z.2$, x) and lightSUPPORT($z.3$, x) first. Since they are empty, we search the domain of x, and find that $z.2$ is not supported and a support of $z.3$ is $x.2$. Next assume $z.1$ is removed. $x.2$ is supported by $z.1$ and thus we need to find a new support for it. We find $z.3$ directly from lightSUPPORT($x.2$, z). Here bidirectionality helps save effort by reusing the result in finding a support for $z.3$ before.

3.2 Algorithm AC-3 and Its Complexity Analysis

As it is clear in the previous section, the central issue of an arc consistency algorithm is to decide what (constraints or values) to be inspected further when some value is removed. Once the strategy is decided, the rest follows in a straightforward way. As an example, we show in this section a specific algorithm, AC-3, and its complexity analysis. There are two considerations to choose AC-3. Firstly, it is a background for our further investigation in the next section. Secondly, the other algorithms follow exactly the same algorithm structure and data structure.

The presentation of AC-3 follows [Mac77a, MF85] with a slight change in notation and node consistency removed.

When the consistency of an arc (i, j) is checked, we also remove those invalid values in x_i to make (i, j) consistent if necessary. This process is called *revising* arc

(i, j), or revising the domain of x_i with respect to (i, j) to emphasize the removal of invalid values. Each removed value of x_i may affect the consistency of all arcs incident into i, $\{(k, i) \mid (k, i) \in E\}$. All removed values in revising (i, j) actually share the same set of arcs to re-inspect. So, for each revision of (i, j), only one set of arcs will be re-inspected as long as some value is removed. The procedure REVISE((i, j)) in Fig 3.4 implements the above revision of (i, j). It uses DELETE to indicate whether there is any value removed from the domain of x_i.

```
procedure REVISE((i,j))
  begin
    DELETE ← false;
    for each a ∈ D_i do
1.    if there is no b ∈ D_j such that (a,b) ∈ c_ij then
          delete a from D_i;
          DELETE ← true
      endif
    return DELETE
  end
```

Figure 3.4: Procedure REVISE for AC-3

We will show in the next section that different implementations of line 1 (in REVISE) cause different worst case complexities. As such, we argue that it is more useful to think of AC-3 as a framework rather than a specific algorithm.

```
algorithm AC-3
  begin
1.  Q ← {(i,j) | c_ij ∈ C or c_ji ∈ C, i ≠ j}
2.  while Q not empty do
3.    select and delete any arc (i,j) from Q;
4.    if REVISE((i,j)) then
5.      Q ← Q ∪ {(k,i) | (k,i) ∈ C, k ≠ i, k ≠ j}
    endwhile
  end
```

Figure 3.5: The AC-3 algorithm

The removal of values in x_i causes a set of arcs to be revised again. Since each time only one arc can be revised, a set[1] Q is used to hold the affected arcs. Let us look at the main algorithm of AC-3 in Fig 3.5. At the beginning, we need to check whether all arcs are consistent. So, they are put into Q in line 1. Obviously, the algorithm shouldn't terminate as long as there is any arc in Q, leading to the while loop in line 2. Line 3 just takes an arc (i, j) and removes it from the Q. It is revised in line 4. The return value of REVISION being true means some value in D_i is removed and thus the arcs incident to i, $\{(k, i)|c_{ki} \in C\}$, go into Q for future revision (line 5). Note the trick that (j, i) is not included in the set. The **while** loop can be intuitively understood as constraint propagation, i.e. propagating the effect of the removed values.

It is time now to have an analysis of the complexity of AC-3. The traditional understanding is given by the following theorem. A more intuitive proof than the one in [MF85] is shown here, which also facilitates the complexity analysis in the next section.

Theorem 1 *[MF85] Given a network (N, D, C), the time complexity of AC-3 is $\mathcal{O}(ed^3)$. The working space complexity, excluding the space for the constraint network, of AC-3 is $\mathcal{O}(ed)$.*

Proof. The key operation of AC-3 is to revise an arc. Consider the times of revision of each arc (i, j). (i, j) is revised if and only if it enters Q. Arc (i, j) enters Q if and only if some value of j is deleted (line 2–3 in Fig 3.5). So, arc (i, j) enters Q at most d times and thus is revised d times. The complexity of REVISE((i, j)) in Fig 3.4 is at most d^2. Hence, given that the number of arcs are $2e$, the time complexity of AC-3 is $\mathcal{O}(ed^3)$.

The only extra space used by the algorithm is the set Q. From the above analysis, each arc enters it at most d times and thus $2e$ arcs imply that the set has

[1] Of course other data structures like queue and stack can also perfectly serve the purpose.

at most $2ed$ arcs in it. The working space complexity of $\mathcal{O}(ed)$ follows. □

Remark. In the implementation of an AC algorithm, a queue, rather than a set, is always used. There are many ways to organize the queue. One of them is to replace the arcs in the queue as the variables (nodes) whose domain have been modified. Whenever we take a variable from the queue, we simply revise all arcs incident to it. One disadvantage of this method is that it fails to play the minor trick[2] of AC-3 mentioned above. Its advantage is that the space occupied by the queue of variables can be easily controlled to be $\mathcal{O}(n)$ by entering a variable into the queue only once (an array with n flags may be used to indicate whether a variable is in the queue or not). Similarly, the space complexity of the original queue of arcs can be decreased to $\mathcal{O}(e)$.

3.3 AC-3.1: A New View of AC-3

The traditional view of AC-3 with the worst case time complexity of $\mathcal{O}(ed^3)$ (described by Theorem 1) is based on a naive implementation of line 1 in Fig 3.4 that b is always searched from scratch. Hereafter, for ease of presentation, we call this implementation AC-3.0. The new approach to AC-3 in this thesis, called AC-3.1, makes use of the observation that b in line 1 of Fig 3.4 does not need to be searched from scratch even though the same arc (i, j) may enter Q many times. The search is simply resumed from the point where it stopped in the previous revision of (i, j). This idea is implemented by procedure EXIST$b((i, a), j)$ in Fig 3.6.

Assume without loss of generality that each domain D_i is associated with a total ordering. For each $(i, j) \in E$ and $a \in D_i$, ResumePoint$((i, a), j)$ records the first support $b \in D_j$ found in the previous revision of (i, j). The $succ(b, D_j^0)$ function, where D_j^0 denotes the domain of j before arc consistency enforcing, returns the

[2]However, in our experiments, the trick to enter only relevant arcs into the queue costs more than it saves.

successor of b in the ordering of D_j^0 or NIL, if no such element exists. NIL is a value not belonging to any domain and precedes all values in any domain. ResumePoint$((i,a),j)$ needs to be initialized with NIL in the main algorithm.

```
procedure EXISTb((i,a),j) // ResumePoint is initialized in main algorithm
  begin
    b ← ResumePoint((i,a),j);
1.  if b ∈ D_j then // b is still in the domain
      return true;
    else
2.        while ((b ← succ(b, D_j^0) and (b ≠ NIL))
            if b ∈ D_j and (a,b) ∈ c_ij then
              ResumePoint((i,a),j) ← b;
              return true
            endif;
          return false
    endif
  end
```

Figure 3.6: Procedure for searching b in REVISE(i,j)

Theorem 2 *The worst case time complexity of AC-3 can be achieved in O(ed^2). The working space complexity is $\mathcal{O}(ed)$.*

Proof. Here it is helpful to regard the execution of AC-3.1 on a network as a sequence of calls to EXIST$b((i,a),j)$.

For each arc (i,j), let us examine the time spent on justifying an $a \in D_i$. As in Theorem 1, an arc (i,j) enters Q at most d times. So, with respect to (i,j), any value $a \in D_i$ will be passed to EXIST$b((i,a),j)$ at most d times. Let the complexity of each execution of EXIST$b((i,a),j)$ be t_l $(1 \leq l \leq d)$. t_l can be considered as 1 if $b \in D_j$ (see line 1 in Fig 3.6) and otherwise it is s_l which is simply the number of elements in D_j skipped before the next support b is found (the while loop in line 2). The total time spent on $a \in D_i$ with respect to (i,j) is $\sum_1^d t_l \leq \sum_1^d 1 + \sum_1^d s_l$ where $s_l = 0$ if $t_l = 1$. Observe that in EXIST$b((i,a),j)$, the while loop (line 2) will

skip an element in D_j at most once with respect to $a \in D_i$. Therefore, $\sum_1^d s_l \leq d$. This gives $\sum_1^d t_l \leq 2d$.

For each arc (i, j), we have to check at most d values in D_i and thus at most $O(d^2)$ time will be spent on checking arc (i, j). Thus, the complexity of the new implementation of AC-3 is $O(ed^2)$ since the number of arcs in the constraint network is $2e$.

For each arc (i, j), and for each value $a \in D_i$ we have to remember its resumption point. The total number of arcs is $2e$ and the maximum number of values in any domain is d. Hence the working space complexity is $\mathcal{O}(ed)$. □

Remark. The space complexity of AC-3.1 is not as good as that of the traditional implementation of AC-3, which can achieve $\mathcal{O}(e)$ easily by using a smart queue as discussed in the remark of the previous section. The additional space to remember the resumption point, needed by AC-3.1, is very hard to compress. However, the extra space required by AC-3.1 is the same as that of AC-6.

3.4 A New Path Consistency Algorithm with the Flavor of AC-3.1

In studying how to solve a network of binary constraints, Montanari introduced path consistency in 1974 [Mon74]. In this section, we assume[3] there is a constraint between *any two* variables in a network. If a network does not satisfy this assumption, the universal constraint is introduced on any pair of variables which is unconstrained.

Definition 5 *Given a network* (N, D, C) *and its undirected graph. A path* $i_0 i_1 \cdots i_m$ *is* consistent *if and only if for any* $(a, b) \in c_{i_0 i_m}$ *there exists a value* a_l *for each node*

[3] [BSH99] does not make this assumption. But the algorithm there can achieve path consistency only on the constraint networks with special topological structure.

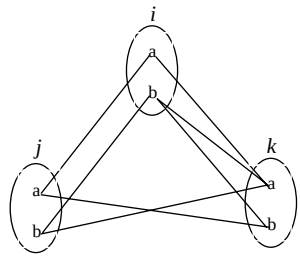


Figure 3.7: Example for path consistency

i_l (that is, variable x_{i_l}) in the path such that constraints

$$c_{i_0 i_1}, c_{i_1 i_2}, \cdots, c_{i_{l-1} i_l}, \cdots, \text{ and } c_{i_{m-1} i_m}.$$

are satisfied. A network is path consistent *iff all its paths are consistent.*

Montanari noticed that in order to make the network path consistent it is sufficient to make any path of length two consistent.

Example. Consider the constraint network in Fig 3.7. It is easy to verify that the network is arc consistent. However, it is not path consistent because the path ikj is not consistent. We have $(a, a) \in c_{ij}$, that is, from $i.a$ we can reach $j.a$ along arc (i, j). In the path from i to j through k, the only tuples allowed are $(i.a, k.a)$ and $(k.a, j.b)$. In other words, from $i.a$ we can not reach $j.a$ along the path ikj. Intuitively, if we take a value a for variable i, constraints along the path ikj tell that a can never be assigned to j although it is allowed by constraint c_{ij}. Hence, the path ikj is not consistent. In order to make it consistent, we simply remove the tuples (a, a) from the arc (i, j). For $(b, b) \in c_{ij}$, along the path ikj we can find $(b, a) \in c_{ik}$ and $(a, b) \in c_{kj}$. In other words, (b, b) satisfies not only c_{ij} but also the constraint *implied* by the path ikj. Finally, $c_{ij} = \{(b, b)\}$. Following the same way, we have $c_{ik} = \{(b, b)\}$ and $c_{kj} = \{(b, b)\}$. Now the modified network is path consistent. Note that in contrast to the revision of domains in arc consistency, here it is the *constraints* that are revised, which of course may lead to the removal of

values. □

If some tuple $(a, b) \in c_{ik}$ is removed, for all $j \in N(j \neq i \ or \ k)$, we need to check whether the paths ikj and kij are consistent. It is done by the procedure REVISE_PATH in Fig 3.10. Specifically, to check ikj, for every $(a, u) \in c_{ij}$, not *all* tuples in c_{ij} but only tuples starting with a, we need to find a *support* $r \in D_k$ such that a, r, u will satisfy the constraints along the path. Similarly we need to find a support $r \in D_i$ for each $(b, u) \in c_{kj}$ with respect to i. See the picture in Fig 3.8 where we need to find a support in D_i or D_k for each edge in c_{kl} or c_{il} for all $l \in N - \{i, k\}$ after the deletion of $(a, b) \in c_{ik}$.

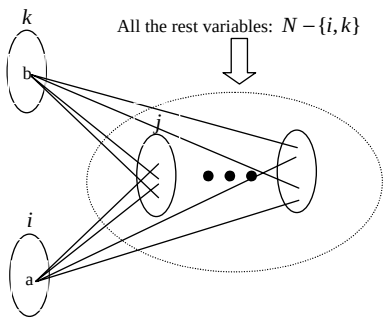


Figure 3.8: The way of propagation in path consistency after the deletion of (a, b) from constraint c_{ik}

The same idea behind AC-3.1 applies here. Specifically, in order to find a new support for each $(a, u) \in c_{ij}$ with respect to a variable, say k, it is not necessary to start from scratch every time. We start from where we stopped before. ResumePoint$((i, a), (j, u), k)$ is used to remember that point. The PC algorithm, which is partially motivated by the algorithm in [CJ96], is shown in Fig 3.9.

Theorem 3 *The time complexity of the algorithm PC is $O(n^3d^3)$ with working space complexity $O(n^3d^2)$.*

Proof. The complexity of PC depends on the procedure REVISE_PATH whose second loop is to find a support for the tuple $(i.a, j.u)$ with respect to k. The while

```
algorithm PC
  begin
    INITIALIZE(Q);
    while Q not empty do
        Select and delete any ((i,a),j) from Q;
        REVISE_PATH((i,a),j,Q))
    endwhile
  end
 procedure INITIALIZE(Q)
  begin
    for any i,j,k ∈ N do
      for any a ∈ D_i, b ∈ D_j such that (a,b) ∈ c_ij do
        if there is no r ∈ D_k such that (a,r) ∈ c_ik ∧ (r,b) ∈ c_kj
        then
            c_ij(a,b) ← false;
            c_ji(b,a) ← false;
            Q ← Q ∪ {(i,a),j} ∪ {(j,b),i}
        else ResumePoint((i,a),(j,b),k) ← r
  end
```

Figure 3.9: Algorithm to enforce path consistency

```
 procedure REVISE_PATH( (i,a),k,Q)
  begin
    for any j ∈ N, j ≠ i, j ≠ k do
      for any u ∈ D_j such that (a,u) ∈ c_ij do
        r ← ResumePoint((i,a),(j,u),k);
1.      while not ((r ≠ NIL) ∧ (a,r) ∈ c_ik ∧ (r,u) ∈ c_kj) do
          r ← succ(r, D_k^0);
        if r = NIL then
          Q ← Q ∪ {((i,a),j} ∪ {((j,u),i)};
        else ResumePoint((i,a),(j,u),k)) ← r
      endfor
  end
```

Figure 3.10: Revision procedure for PC algorithm

loop in line 1 (Fig 3.10) either takes constant time if the condition is not satisfied, or skips several values in D_k otherwise. For the second case, no matter how many times we try to find a support for $(i.a, j.b)$, at most we skip d values since totally we have only d values in D_k.

Now we need to know how many times it is necessary to find a support for $(i.a, j.u)$ with respect to k. It is necessary to find such a support if and only if some tuple (a, u) is removed from c_{ik}. So we need find such a support d times. From the first paragraph, for these d times we have at most d constant checks and d skips in total. As a result, to find a support for $(i.a, j.u)$ with respect to k we need $2d$ steps. Given that i, j, k can be any variables from N and a, u any values from corresponding domains, we have n^3d^2 possible $(i.a, j.b)$'s and k's. Hence, the total time cost is $n^3d^2 \times 2d$, that is $\mathrm{O}(n^3d^3)$.

The main working space is for the structure ResumePoint$((i, a), (j, u), k)$. The size of this structure is the number of combinations of possible choices for i, j, k, a, u, that is $\mathrm{O}(n^3d^2)$. □

The time complexity and space complexity of the PC algorithm here are the same as the best known theoretical results [Sin96]. However, this PC algorithm is simpler than those algorithms reported in [Sin96].

3.5 Preliminary Experimental Results

In this section, we present preliminary experimental results on the efficiency of AC-3. While arc consistency can be applied after each instantiation in the context of search (such as [BR96]), we focus on an experimental evaluation of the standing alone arc consistency algorithms.

The experiments are designed to compare the empirical performance of the new AC-3.1 algorithm with both the classical AC-3.0 algorithm and a state-of-the-art algorithm, AC-6, on a range of CSP instances with different properties. AC-6 is

chosen[4] as a representative of state-of-the-art algorithms because of its good timing performance over the problems of concern (see [BFR99]).

There have been many experimental studies on the performance of general arc consistency algorithms [Wal93, Bes94, BFR99]. Here, we adopt the choice of problems[5] used in [BFR99], namely some random CSPs, Radio Link Frequency Assignment problems (RLFAPs) and the Zebra problem. The Zebra problem is discarded as it is too small for benchmarking. In addition, we propose the DOMINO problem as a new benchmark to study the worst case performance of AC-3.

Randomly generated problems: As in [FBDR96], a class of random CSP instances is characterized by n, d, e and the tightness of each constraint. The *tightness of a constraint* c_{ij} is defined to be $|D_i \times D_j| - |c_{ij}|$, the number of pairs NOT permitted by c_{ij}. A class of randomly generated CSPs is denoted by a tuple (n, d, e, *tightness*). We use the first 50 instances of each of the following classes of problems generated using the initial seed 1964 (as in [BFR99]): (i) $P1$: underconstrained CSPs $(150, 50, 500, 1250)$ where all generated instances are already arc consistent; (ii) $P2$: over constrained CSPs $(150, 50, 500, 2350)$ where all generated instances are *inconsistent* in the sense that some domain becomes empty in the process of arc consistency enforcing; and (iii) problems in the phase transition [GMP+97] $P3$: $(150, 50, 500, 2296)$ and $P4$: $(50, 50, 1225, 2188)$. The $P3$ and $P4$ problems are further separated into the arc consistent instances, labeled as *ac*, which can be made arc consistent at the end of arc consistency enforcing; and inconsistent instances labeled as *inc*. More details on the choices for P1 to P4 can be found in [BFR99].

RLFAP: The RLFAP [CdGL+99] is to assign frequencies to communication links to avoid interference. We use the real-life CELAR[6] instances of RLFAP which

[4]We note that AC-6p has a slightly better performance than AC-6. However, we believe that its heuristic of propagating deletion first may also apply to AC-3 algorithms. Further discussions can be found in the next section.

[5]We thank Christian Bessiere for providing benchmarks and discussions for our experiment on AC algorithms.

[6]We acknowledge the generosity of the French Centre d'Electronique de l'Armement for providing the CELAR benchmarks.

		AC-3.0	AC-3.1	AC-6
P1	#ccks	100,010	100,010	100,010
	time(50)	0.65	0.65	1.13
P2	#ccks	494,079	475,443	473,694
	time(50)	1.11	1.12	1.37
P3(ac)	#ccks	2,272,234	787,151	635,671
	time(25)	2.73	1.14	1.18
P3(inc)	#ccks	3,428,680	999,708	744,929
	time(25)	4.31	1.67	1.69
P4(ac)	#ccks	3,427,438	1,327,849	1,022,399
	time(21)	3.75	1.70	1.86
P4(inc)	#ccks	5,970,391	1,842,210	1,236,585
	time(29)	8.99	3.63	3.54

Table 3.1: Randomly generated problems

are available at `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks`.

RFLAP		AC-3.0	AC-3.1	AC-6
#3	#ccks	615,371	615,371	615,371
	time(20)	1.47	1.70	2.46
#5	#ccks	1,762,565	1,519,017	1,248,801
	time(20)	4.27	3.40	5.61
#8	#ccks	3,575,903	2,920,174	2,685,128
	time(20)	8.11	6.42	8.67
#11	#ccks	971,893	971,893	971,893
	time(20)	2.26	2.55	3.44

Table 3.2: CELAR RLFAPs

DOMINO: A DOMINO problem instance is characterized by two parameters n and d. Recall that the trigger constraint will make only one value invalid and that value will trigger the domino effect on the values of all domains until each domain has only one value d left. So, each revision of an arc in AC-3 algorithms can only remove one value while AC-6 only does the necessary work. This problem is used to illustrate the differences between AC-3 like algorithms and AC-6. The results show that arc revision oriented algorithms may not be so bad in the worst

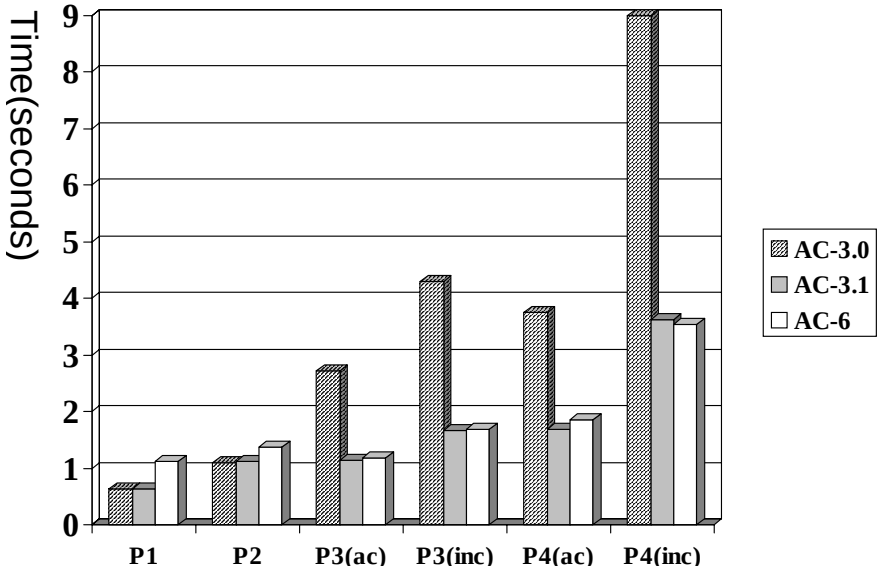


Figure 3.11: Running time for randomly generated problems

case as one might imagine.

d		AC-3.0	AC-3.1	AC-6
100	#ccks	17,412,550	1,242,550	747,551
	time(10)	5.94	0.54	0.37
200	#ccks	136,325,150	4,985,150	2,995,151
	time(10)	43.65	2.21	1.17
300	#ccks	456,737,750	11,227,750	6,742,751
	time(10)	142.38	5.52	2.69

Table 3.3: DOMINO problems

Some details of our implementation of AC-3.1 and AC-3.0 are as follows. We implement domain and related operations by employing a doublely-linked list. The Q in AC-3 is implemented as a queue of variables into which arcs incident will be revised [CJ96]. A new variable will be put at the end of the queue. Variables in the queue are treated in a FIFO order. The code is written in C++ compiled by *g++*. The programs are run on a Pentium III 600 processor with Linux.

For AC-6, we note that in our experiments, using a single currently supported

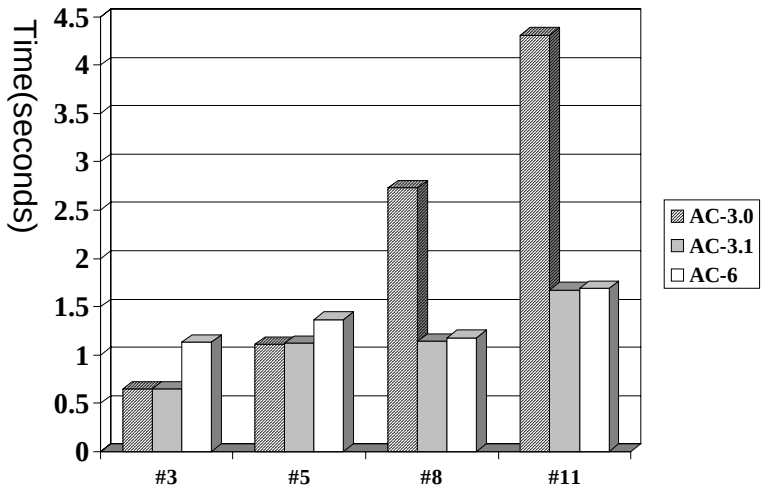


Figure 3.12: Running time for CELAR RLFAPs

list of values (see [Bes94]) is faster than using multiple lists with respect to related constraints proposed in [BFR99]. This may be one reason why AC-7 is slower than AC-6 in [BFR99]. The experimental data reported below is produced by an AC-6 with a single list.

The performance of arc consistency algorithms here is measured along two dimensions: running time and number of constraint checks (#ccks). A *raw constraint check* tests if a pair $(i.x, j.y)$ satisfies constraint c_{ij}. In this experiment we assume constraint check is cheap and thus the raw constraint and additional checks (e.g. line 1 in Fig 3.6) in both AC-3.1 and AC-6 are counted. In the tabulated experiment results, *#ccks* represents the average number of checks on tested instances, and *time(x)* the time in seconds on x instances.

The results for randomly generated problems are listed in Table 3.1 and Fig 3.13. For the underconstrained problems $P1$, AC-3.1 and AC-3.0 have similar running time. No particular slowdown for AC-3.1 is observed. In the over constrained problems $P2$, the performance of AC-3.1 is close to AC-3.0 but some constraint checks are saved. In the hard phase transition problems $P3$ and $P4$, AC-3.1 shows

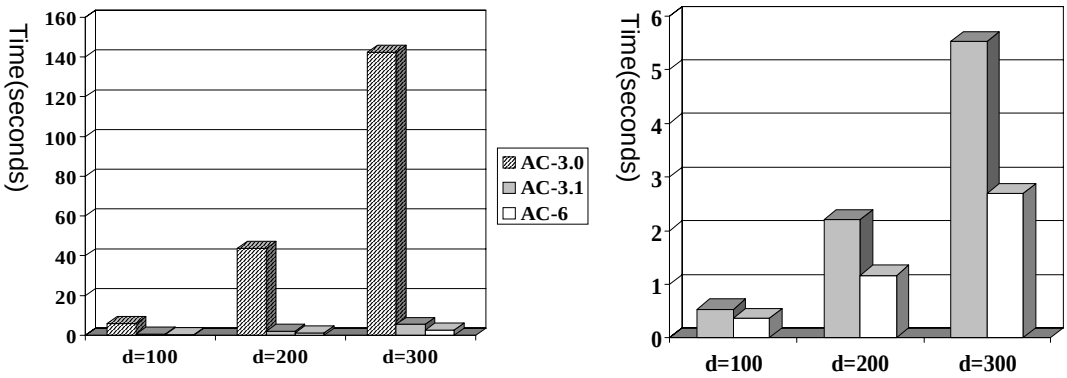


Figure 3.13: Running time for DOMINO problems

significant improvement over AC-3.0 in terms of both the number of constraint checks and the running time. It is better than or close to AC-6 in timing although it has more checks.

The results for CELAR RLFAP are given in Table 3.2 and Fig 3.12. In simple problems, RLFAP#3 and RLFAP#11, which are already arc consistent before the execution of any AC algorithm, no significant slowdown of AC-3.1 over AC-3.0 is observed. For RLFAP#5 and RLFAP#8, AC-3.1 is faster than both AC-3.0 and AC-6 in terms of timing.

The reason why AC-6 takes more time while making less checks can be explained as follows. The main contribution to the slowdown of AC-6 is the maintenance of the currently supported list associated with each value of all domains. In order to achieve space complexity of $\mathcal{O}(ed)$, when a value in the currently supported list is removed, the space occupied in the list by that value has to be released. Our experiment shows that the overhead of maintaining the list doesn't compensate for the savings from less checks under the assumption that constraint checking is cheap.

The DOMINO problem is designed to show the gap between AC-3 implementations and AC-6. Results in Table 3.3 and Fig 3.13 show that AC-3.1 is about

half the speed of AC-6. This can be explained by a variation of the proof in section 3. In AC-3.1 the time spent on justifying the validity of a value with respect to a constraint is at most $2d$ while in AC-6 it is at most d.

The DOMINO problem also shows that AC-3.0 is at least an order of magnitude slower in time and more in number of constraint checks than AC-3.1 and AC-6. This can be justified by the fact that AC-3.0 achieves its worst case complexity on DOMINO problem.

In summary, our experiments on randomly generated problems and RLFAPs show the new approach to AC-3 is stable and efficient on both simple problems and hard problems compared with the traditional view of AC-3 and state-of-the-art algorithms.

3.6 Related Work and Discussion

Our work reported here is related to the development of general purpose arc consistency algorithms, for example AC-3, AC-4, AC-6, AC-7 and the work of [Wal93]. We summarize previous algorithms before discussing how our algorithm gives an insight into AC-3.

Conventionally, the arc consistency algorithms are named chronologically. However, this may not exhibit the intrinsic relationship among the algorithms and is misleading in some sense. We prefer to classify the well known algorithms according to their methods of propagation (see Section 3.1).

As far as we know, there are two approaches employed in efficient AC algorithms: arc oriented and value oriented propagation. The former originates from AC-1 and its underlying computation model is the variable based constraint graph. The latter originates from AC-4 and its underlying computation model is the value based constraint graph.[7] The key idea of value oriented propagation is that once

[7] As far as we know, Perlin [Per92] is the first to make value based constraint graph explicit in

a value is removed only those values related to it will be checked. Thus it is more fine grained than arc oriented propagation. We sometimes also call algorithms working with variable based graph *coarse grained algorithms*, and those working with value based graph *fine grained algorithms*. An immediate observation is that compared with variable based constraint graph, time complexity analysis in value based constraint graph is straightforward (as discussed in section 3.1).

Given a computation model of propagation, the algorithms differ in the implementation details. For variable based constraint graph, AC-3 [Mac77a] can be thought of as an "open implementation". The approach in [MF85] can be regarded as a realized implementation. The new view of AC-3 presented in this chapter can be thought of as another "implementation" with optimal worst case complexity. It simply remembers the result obtained in previous revision of an arc while in the old one, the choice is to be lazy, forgetting previous effort. There are still other algorithms falling into the scope of this model. For example [CJ96] is devoted to improving the space complexity. For value based constraint graph, AC-4 is the first implementation and AC-6 is a lazy version of AC-4 (see Section 3.1). AC-7 is based on AC-6 and it exploits the bidirectionality. [Per92] and [KD94] use this model explicitly.

AC-4 does not perform well in practice [Wal93, BFR99] because it always *reaches* the worst case complexity both theoretically and in actual problem instance when constructing the SUPPORT structure. Other algorithms like AC-3 and AC-6 can take advantage of many instances where the worst case doesn't occur. In practice, both artificial and real life problems rarely make algorithms behave in the worst case except for AC-4. However, the value based constraint graph induced from AC-4 provides a convenient and accurate tool for studying arc consistency.

---

arc consistency enforcing algorithm.

Given that both variable and value based constraint graphs can lead to worst case optimal algorithms, we consider their strength on some special constraints: functional, anti-functional (Chapter 5), and monotonic (Chapter 4) constraints.

For coarse grained algorithms, it can be shown that arc consistency on *monotonic* and *anti-monotonic* constraints can be enforced with time complexity of $\mathcal{O}(ed)$ (e.g. using our new view of AC-3). Fine grained algorithms like AC-4 and AC-6 can deal with *functional* constraints efficiently. We remark that the particular distance constraints in RLFAP can be enforced to be arc consistent in $\mathcal{O}(ed)$ by using a coarse grained algorithm. It is difficult for coarse grained algorithms to deal with functional constraints and tricky for fine grained algorithms to deal with monotonic constraints.

We also notice that general properties or knowledge of a CSP can be isolated from a specific arc consistency enforcing algorithm. Examples are AC-7 and AC-inference. AC-7 is the result of applying bidirectionality to AC-6. However, bidirectionality is also applicable to course grained algorithms, and in fact originates from the study of the latter [BFR99, Gas78]. We are aware that its potential may not be *fully* exploited under the variable based graph model. The idea of *metaknowledge* on a single constraint or a network [BFR99] may be applied to algorithms of both computation models.

Other propagation heuristics [WF92] such as propagating deletion first [BFR99] are also applicable to algorithms of both models.

Now let us compare the new approach to various arc consistency algorithms. AC-3.1 and AC-6 use different computation model of propagation. From a technical perspective, the time complexity analysis of the new AC-3 is different from that of AC-6 whose worst case time complexity analysis is straightforward. One common point shared by AC-3.1 and AC-6 is that they have to face the same problem: the recorded value may be removed from its domain before we use it and thus we need to check whether it is still in the domain each time we need it. This makes some

portions of the new implementation of the AC-3.1 similar to AC-6. We remark that the proof technique in the traditional view of AC-3 may not directly lead to the new AC-3 and its complexity results.

[Wal93] gives detailed experiments comparing the efficiency of AC-3 and AC-4. Our work complements this in the sense that with the new implementation, AC-3 now has optimal worst case time complexity.

The number of raw constraint checks[8] is frequently used to evaluate practical efficiency of CSP algorithms. It can be shown that if the same ordering of variables and values are processed, AC-3.1 and the classical AC-6 have the same number of raw constraint checks. AC-3.0 and AC-4 will make no less *raw* constraint checks than AC-3.1 and AC-6 respectively.

At last, we remark that the coarse grained algorithms are simpler and can be easily integrated into specific applications.

In summary, there are two computation models underlying known algorithms. As is shown in this chapter, it is possible to develop competitive algorithms in both models in terms of worst case complexity and empirical performance. In order to further improve the efficiency of arc consistency enforcing, more properties (both general like bidirectionality and special like monotonicity) of constraints and heuristics are desirable.

3.7 Summary

In this chapter we present a natural implementation of AC-3 whose complexity is better than the traditional understanding. AC-3 was not previously known to have worst case optimal time complexity[9] even though it is known to be practically

[8] In theory, applying bidirectionality to all algorithms will result in a decrease of raw constraint checks. However, if the cost of a raw constraint check is cheap, the overhead of using bidirectionality may not be compensated as demonstrated by [BFR99]

[9] We notice that Bessiere and Regin have independently developed an algorithm AC-2001 [BR01], similar to AC-3.1, with optimal worst case complexity.

efficient. Our new implementation brings AC-3 to $\mathcal{O}(ed^2)$ on par with the other optimal worst case time complexity algorithms. Techniques in the new implementation can also be used with path consistency algorithms.

While worst case time complexity gives us the upper bound on the time complexity, in practice, the running time and number of constraint checks for various CSP instances are the prime consideration. Preliminary experiments show that the new implementation significantly reduces the number of constraint checks and the running time of the traditional one on hard arc consistency problems. Furthermore, the running time of AC-3.1 is competitive when compared with the best algorithms, based on the benchmarks from the experiment results in [BFR99]. The raw constraint checks required by both AC-3.1 and AC-6 are the same. We conjecture that based on the CELAR instances, the new approach to AC-3 may lead to a more robust AC algorithm for real world problems than other algorithms.

We also show how the new AC-3 leads to a new algorithm for path consistency. We conjecture from the results of [CJ96] that this algorithm can be a practical implementation for path consistency.

Chapter 4

Arc Consistency on Non-binary Monotonic and Linear Constraints

In the previous chapter, we focused on binary constraints. Indeed they were the main concern in the study of arc consistency before the 1990s. In the 1990s, more and more applications are brought to the realm of Constraint Satisfaction Problem through Constraint Programming (CP) systems. They can be naturally and conveniently modeled by non-binary constraints, which involve more than two variables. Some typical examples of non-binary constraints include the *all different* constraint, the *cardinality* constraint [Reg96] and linear arithmetic constraints. To deal with new applications efficiently, it is necessary to study non-binary constraint networks.

There are two main approaches to deal with non-binary networks. The first is to avoid altogether the question of a non-binary network. This is achievable since it is always possible to translate a non-binary network into a binary one [DP89, RPD90]. The standard techniques for binary networks can then be used to solve the transformed CSP, thus solving the original non-binary CSP. A recent paper [BvB98] examined this approach in detail.

The second approach is to develop consistency techniques directly for non-

binary constraints. One obvious way is to extend techniques developed in the binary network for non-binary network. AC-3 was generalized by Mackworth [Mac77b] to the algorithm NC to deal with non-binary constraints. Later GAC-4 [MM88] was proposed to make use of the (support) technique developed in AC-4 (see Section 3.1). It improves the complexity of NC, at the cost of a complex data structure and thus higher space complexity. To initialize that data structure, it always reaches its worst case complexity as AC-4 does for any instance. The time complexity of GAC-4 is $\mathcal{O}(ed^r)$ where e is the number of constraints, d is the size of the domain and r the maximum arity of constraints in a network. Unlike their counterparts for binary network, NC and GAC-4 may not be practical due to their high time complexity. A more feasible approach is the GAC-schema [BR97] based on *single support*, and *multidirectionality* that is a generalization of bidirectionality to non binary constraint, but it has the same worst case time complexity as GAC-4.

Another possibility in the second approach is to design specialized techniques to exploit the semantics of particular non-binary constraints. Efficient consistency algorithms have been developed for particular classes of constraints. Examples are the algorithms for the global *all different* constraint and *cardinality* constraint [Reg96].

In this chapter, we address the issue of efficiency of arc consistency enforcing algorithms. We found that even with a restriction of non-binary constraints to linear constraints, to enforce arc consistency on a network remains intractable. We identify a general class of monotonic non-binary constraints, which includes linear inequalities as a special case, with tractable algorithms. A network of monotonic constraints can be made arc consistent in time $\mathcal{O}(er^3d)$. A network of linear inequalities can be made arc consistent in time $\mathcal{O}(er^2d)$ by using bounds consistency which exploits the special properties of a *projection function*.

We first present some background material for arc consistency on non-binary network. In Section 4.2, bounds based propagation is formalized as bounds consis-

tency on linear constraints. An efficient bounds consistency algorithm is proposed for linear constraints. In Section 4.3 we consider arc consistency on linear inequalities. A new class of monotonic constraints is identified and a polynomial algorithm is developed to enforce AC on those constraints. In Section 4.4, we examine arc consistency for linear equations. Related work is discussed in the last section.

4.1 Arc Consistency on Non-binary Constraints

A constraint in a non-binary network may be defined and represented in a number of ways. It can be represented explicitly as a set of allowed (or disallowed) tuples, implicitly as an arithmetic expression, or by any predicate whose semantics is defined by a particular definition/program code. c_S is used to denote both the (representation) form of a constraint among variables in S and the set of tuples that satisfy the constraint.

Notation. Again in the presentation of this chapter a variable x_i and its index i are used interchangeably when there is no confusion. For a constraint c without subscript S, *vars(c)* and $|vars(c)|$ is used to denote the set and the number of variables that occur in c respectively.

Definition 6 *Given a network (N, D, C) and a constraint $c_S \in C$ where $S = \{i_1, \cdots, i_r\}$, we define a* solution *of constraint c_S to be any tuple $(v_{i_1}, \cdots, v_{i_r}) \in c_S$. If c_S is empty, we say that there is no solution for c_S.*

We also use $c_S(v_{i_1}, \cdots, v_{i_r})$ to denote that $(v_{i_1}, \cdots, v_{i_r})$ is a solution of c_S.

The following definition of arc consistency for non-binary constraints [Mac77b] is a natural generalization of the one for binary networks.

Definition 7 *Given a network (N, D, C), a constraint $c_S \in C$ is* arc consistent *with respect to D iff $\forall i \in S$ and $\forall v \in D_i$, v is* valid *with respect to c_S, that is v is a*

component of a solution of c_S*. A network* (N, D, C) *is arc consistent iff all* $c_S \in C$ *are arc consistent.*

The arc consistency for non-binary network is sometimes also called *hyper-arc consistency*. We remark that the definition of arc consistency is similar to relational arc consistency [vBD95]. Enforcing higher level of consistency such as relational path consistency on non-binary networks is NP-complete in general (see Section 4.4).

The task of an arc consistency algorithm is to remove those invalid values from the domains of variables with respect to each constraint. In a binary network, the representation of a constraint may not be so important for this process. In the non-binary case, the representation form of a constraint may fundamentally affect the complexity of an arc consistency algorithm. For example, the *all different* constraint can be represented in a number of ways. Suppose that we represent the *all different* constraint using an explicit tuples as in GAC-4, the arc consistency algorithm has a polynomial complexity with respect to the size of input. However, the set of allowed tuples could be too huge to make the algorithm practical in terms of space and time. The GAC-schema of [BR97] is proposed to partly address this problem. However, it is a general framework and does not address how to deal with special constraints such as linear arithmetic constraints efficiently.

4.2 Bounds Consistency on Linear Constraints

In the first subsection, we introduce a special class of non-binary networks—linear arithmetic constraints and define bounds consistency on them. In the second subsection, we present an algorithms to enforce bounds consistency and its complexity analysis.

4.2.1 Linear Constraint and Bounds consistency

We denote the set of integers by Z.

Definition 8 A linear arithmetic constraint $c_{\{x_1,\cdots,x_r\}}$ *is of the form*

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

$$a_i \in Z \textit{ for } i \in 1..r, b \in Z, \textit{ and } \diamond \in \{=, \leq\}.$$

A *linear constraint network* is one where every constraint is a linear arithmetic constraint and each domain contains only a finite number of integers. Other linear arithmetic constraints with $(<, >, \geq)$ can be easily transformed into the above form.

Essentially, the problem of enforcing arc consistency on a single constraint is related to that of finding all solutions of the given constraint. This may be quite expensive. One well known way to reduce this cost is to relax domains of the variables so that they form a continuous real interval bounded by the maximum and minimum values of the corresponding domains. Since variables can now take real values and are no longer discrete, it is easy to make the constraint arc consistent with respect to the real intervals[1]. Some basic interval arithmetic operations [Moo66] are introduced to simplify our presentation.

Assume that each variable x is associated with an interval $[l, u]$. $[x]$ denotes the interval $[l, u]$ associated with x, and $\langle x \rangle$ denotes a vector with two components l and u:

$$\langle x \rangle = \begin{pmatrix} l \\ u \end{pmatrix}$$

Let us now define two types of operations on x: interval operation and literal operation.

[1]Note that, here, the arc consistency of a constraint network with infinite domains is a straightforward extension of the arc consistency of a network with finite domains.

Given $[x] = [l_1, u_1]$, $[y] = [l_2, u_2]$, and a a real number, the *interval operations* are defined in the usual fashion:

$$[x] + [y] = [l_1 + l_2, u_1 + u_2],$$

$$[x] - [y] = [l_1 - u_2, u_1 - l_2],$$

$$[x] - a = [l_1 - a, u_1 - a],$$

$$a[x] = \begin{cases} [al_1, au_1], a > 0 \\ [au_1, al_1], a < 0, \end{cases}$$

$$[x] \cap [y] = [max(l_1, l_2), min(u_1, u_2)].$$

The *literal operations* are defined as a pairwise vector operation, which differs in subtraction from the interval counterpart:

$$\langle x \rangle \pm \langle y \rangle = \begin{pmatrix} l_1 \pm l_2 \\ u_1 \pm u_2 \end{pmatrix}.$$

We also need the transformation between [ ] and $\langle\ \rangle$.

$$\langle [x] \rangle = \begin{pmatrix} l \\ u \end{pmatrix} \text{ where } [x] = [l, u],$$

and

$$[\langle x \rangle] = [l, u] \text{ where } \langle x \rangle = \begin{pmatrix} l \\ u \end{pmatrix}.$$

To relate the consistency and interval operations, consider the example

$$3x - 4y = 0, [x] = [y] = [1, 10].$$

Clearly, y cannot take the value 10 no matter what value x takes. More precisely, given any value of x in $[1, 10]$, y can only take a value in $[3/4, 30/4]$. So, the set of valid values of y with respect to the above constraint is $[3/4, 30/4] \cap [1, 10]$=$[3/4, 30/4]$. The above process to remove invalid values can be formalized as follows.

Definition 9 *The* projection function π_i *of a constraint* c *on* x_i *is*

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

Given an interval for each variable, we can define the interval version of the projection of c *on* x_i *as:*

$$\Pi_i(c) = \frac{-1}{a_i}(a_1[x_1] + \cdots + a_{i-1}[x_{i-1}] + a_{i+1}[x_{i+1}] + \cdots + a_r[x_r] - b).$$

We call $\Pi_i(c)$ *the* natural interval extension *of* $\pi_i(c)$.

In the above example, $\pi_y(c) = \frac{3}{4}x$. Its natural interval extension $\Pi_y = \frac{3}{4}[x]$.

We now define the function $Proj_i(c)$ as follows:

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

where

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

$$Ub([l, u]) = u,$$

$$Lb([l, u]) = l.$$

We have the following property on arc consistency on a single constraint.

Proposition 1 *Given a constraint c with initial domains $([x_1], \cdots, [x_r])$, c is arc consistent with respect to new domains $([x_1] \cap Proj_1(c), \cdots, [x_r] \cap Proj_r(c))$.*

Proof. This is an immediate consequence of the *intermediate value theorem* from calculus. □

The relaxation of the domain of a variable from discrete to a continuous real interval allows efficient arc consistency enforcement on a single linear constraint. It can be done by computing once $Proj_i(c)$ for each $i \in 1..r$. Unlike the discrete case, iteration is not necessary here. However, for a network of constraints, this process should be iterated and may not terminate [JMSY94].

We now define bounds consistency. Instead of using the real interval relaxation, we restrict the interval to the *Z-interval* whose upper bound and lower bound have to be integers. The Z-interval representation of a set A of reals is $\Box A = [\lceil u \rceil, \lfloor v \rfloor]$ where $\lceil u \rceil$ is the ceiling of the minimum real values in A and $\lfloor v \rfloor$ is the floor of the maximum real values in A.

Definition 10 *A constraint c is* bounds consistent *with respect to $(\Box D_{x_1}, \cdots, \Box D_{x_r})$ iff $\forall x_i \in \text{vars}(c)$, $\Box D_{x_i} \subseteq \Box Proj_i(c_i)$. A linear constraint network (N, D, C) is* bounds consistent *with respect to $(\Box D_1, \cdots, \Box D_m)$ iff every $c_i \in C$ is bounds consistent.*

4.2.2 A bounds Consistency Algorithm and Its Complexity

We now describe an AC-3 like algorithm to achieve bounds consistency (BC) on a network of linear constraints. Recall that in AC-3, one constraint is actually treated as two arcs. However, here we take a non-binary constraint as a whole, rather than r (arity) directions (arcs). Another difference is that the REVISE procedure here is specialized for BC and linear constraints. A queue is employed to hold those constraints that need to be revised when the domain of some of its variables is changed. The algorithm is listed in Fig 4.1.

```
Algorithm BC
begin
  Q ← {c|c ∈ C};
  while (Q not empty) do
  begin
    select and delete c from Q;
    REVISE(c, Q);
  end
end
procedure REVISE(c, Q)
begin
  for each x ∈ vars (c) do
  begin
    if [x] ⊈ □Proj_x(c) then
      begin
1.      [x] ← [x] ∩ □Proj_x(c);
2.      Q ← {c ∈ C | x ∈ vars (c)}
      end
  end
end
```

Figure 4.1: Algorithm BC

We point out that the operation in line 1 of BC differs from the *narrowing operation* [BO97] in that the Z-interval representation performs inward rounding while the [BO97] performs an outward rounding operation. The operation on c defined by REVISE is not *idempotent* [BO97].

Lemma 1 *Given a linear constraint network* (N, D, C)*, the worst case time complexity of algorithm BC is* $\mathcal{O}(er^3d)$

Proof. The worst case complexity of BC depends on the number of constraints ever entering the queue Q. A constraint c enters Q iff some value in some domain involved in c is deleted. For each variable $x_i \in N$, assume it appears in k_i constraints. The number of constraints ever entering Q is at most $\sum_{i=1}^{n} d \cdot k_i$. Let α be $\sum_{i=1}^{n} k_i$. A loose estimate of k_i can be simply e which means the variable can appear in any constraint in the network. However, a relatively tighter estimation for α is as follows. Consider the bipartite graph $G_{m,e}$ with vertices sets N and C. There is an edge between $x_i \in N$ and $c_j \in C$ iff x_i appears in c_j. α is exactly the number of edges of $G_{m,e}$. Since the degree of c_j is not more than r we have that the number of edges in $G_{m,e}$ is less than re, that is $\alpha \leq re$. The complexity of procedure REVISE is at most r^2. Therefore the complexity of BC is $\mathcal{O}(er^3d)$. □

This proof has a flavor of the proof in [MF85]. Using the proof technique in the previous chapter, we can have a simple proof. We know that a constraint c enters Q iff some value in some domain involved in c is deleted. Since c has at most r variables, it enters Q at most rd times. Hence time spent on c is $rd \cdot r^2$. Finally, we know that there are e constraints in the network.

The naive algorithm can be improved by making REVISE more efficient.

Proposition 2 *Given a non-binary linear arithmetic constraint network* (N, D, C)*, bounds consistency can be achieved in time* $\mathcal{O}(er^2d)$

Proof. To improve the efficiency of BC, one way is to make REVISE faster. Specifically, when we revise c_j, we need to compute r projections.

Let constraint c_j be

$$a_{j_1}x_1 + a_{j_2}x_2 + \cdots + a_{j_r}x_r \diamond b_j.$$

Let

$$f_j = a_{j_1}x_1 + a_{j_2}x_2 + \cdots + a_{j_r}x_r - b_j$$

Let F_j be the natural interval extension of f_j. Now, for any $x_i \in c_j$

$$\Pi_i(c_j) = -\frac{1}{a_{j_i}}[\langle F_j \rangle - \langle a_{j_i}[x_i] \rangle]. \tag{4.1}$$

since we have that

$$\begin{aligned}[\langle F_j \rangle - \langle a_{j_i}[x_i] \rangle] &= [\langle [a_{j_1}[x_1] + \cdots a_{j_i}[x_i] + \cdots a_{j_r}[x_r] - b_j \rangle - \langle a_{j_i}[x_i] \rangle] \\ &= a_{j_1}[x_1] + \cdots + a_{j_{i-1}}[x_{i-1}] + a_{j_{i+1}}[x_{i+1}] + \cdots + a_{j_r}[x_r] - b_j\end{aligned}$$

Note f_j is not a projection function and the use of the literal $\langle\rangle$ operations in $\Pi_i(c_j)$.

F_j in Equation 4.1 can be computed in time of r. For all $i \in 1..r$, according to its definition, $Proj_i(c_j)$ can now be computed in constant time by using Equation 4.1. Hence, REVISE can be implemented in linear time of r. So, the BC algorithm is of time complexity $\mathcal{O}(er^2d)$. □

Remark. When c_j is revised, F_j is computed only once in terms of the intervals of all variables involved. In the procedure of the revision, the intervals of some variables in c_j may be updated, resulting in the later revision of c_j (line 2 in Fig 4.1). So, the $\Pi_i(c_j)$ calculated by using current F_j may not be up to date. However, the accurate value for $\Pi_i(c_j)$ will be obtained in the next round of revision of c_j.

4.3 Linear Inequalities and Monotonic Constraints

We now consider arc consistency on a network of linear inequalities.

Proposition 3 *Given a non-binary network (N, D, C) with integer domains and only linear inequalities, it is arc consistent if it is bounds-consistent.*

Proof. Assume the network is bounds-consistent. Now we show that any constraint c_j is arc consistent with respect to D. Consider any variable x_i, $x_i \in vars(c_j)$, and any value v, $v \in D_i$. Let l and g be the least and greatest integers in D_i. Let us first assume that the coefficient of x_i in c_j $a_i > 0$, we have $x_i \leq \pi_i(c)$. Since the network is bounds-consistent, we have $[l, g] \subseteq \Box Proj_i(c_j)$, which means that $v \leq g \leq Ub(Proj_i(c_j))$ where $Ub(Proj_i(c_j))$ is obtained by letting

$$x_k = v_k, \forall k \in 1..r, k \neq i.$$

where v_k is either the lower bounds or the upper bounds of D_k depending on the interval operation. So, $(v_1, \cdots, v_{i-1}, v, v_{i+1}, \cdots, v_r)$ satisfies c_j. Similarly, when $a_i < 0$, we can prove v is part of a solution of c_j. $\Box$

The following theorem follows from the above proposition.

Theorem 4 *A network of linear inequalities can be made arc consistent in worst case time complexity of $\mathcal{O}(er^2d)$.*

In fact, this result can be generalized to a bigger class of constraints, the non-binary monotonic constraints. We begin by recalling the definition of binary monotonic constraint in [VHDT92].

Definition 11 [VHDT92] *Given a binary network (N, D, C), a constraint $c \in C$ is* monotonic *with respect to domain $TD = \cup_{i=1}^{n} D_i$ iff there exists a total ordering on TD such that for all values $v, w \in TD$, $c(v, w)$ implies $c(v', w')$ for all $v' \leq v$ and $w' \geq w$.*

An example of an arithmetic constraint which is monotonic under this definition is $x \leq y, [x] = [y] = [1, 10]$. However, with this definition, the linear inequality

$$x + y \leq 10, [x] = [y] = [1, 10]$$

is *not* a monotonic constraint. Consider $x = 5, y = 5$ satisfying the inequality. $x' = 5$ and $y' = 6$, where $x' \leq x$ and $y' > y$ under the natural ordering, are not a solution of the inequality. In fact, there is no total ordering on TD which makes this constraint monotonic under the above definition.

However, a binary network of both kinds of constraints can be made arc consistent in time $\mathcal{O}(\text{ed})$ by algorithm BC. Thus we see that this definition of monotonicity is stronger than necessary and does not fully exploit the special properties of inequalities which may lead to more efficient arc consistency algorithms. We introduce the following generalization of binary monotonic constraint which remedies this problem. Here, the total ordering requirement on the union of all the domains in [VHDT92] is relaxed.

Definition 12 *Given a binary network (N, D, C), a constraint $c_{ij} \in C$ is* monotonic *iff there exists total orderings $\leq_1$ and $\leq_2$ on D_i and D_j respectively such that $\forall v \in D_i, \forall w \in D_j$,*

$$c(v, w)$$

implies

$$c(v', w') \text{ for all } v' \leq_1 v, w \leq_2 w'.$$

The constraint

$$x + y \leq 10, [x] = [y] = [1, 10]$$

is now monotonic if we assume the natural ordering on x, and the reverse of natural ordering on y. A generalization of monotonicity to non-binary constraints is as

follows.

Definition 13 *Given a non-binary network (N, D, C), a constraint $c_S \in C$ is* monotonic *with respect to variable $i \in S$ iff there exists total orderings $\leq_1, \cdots, \leq_r$ on D_1 to D_r respectively such that $\forall v \in D_i, \forall v_j \in D_j$,*

$$c_S(v_1, \cdots, v_{i-1}, v, v_{i+1}, \cdots, v_r)$$

implies

$$c_S(v'_1, \cdots, v'_{i-1}, v', v'_{i+1}, \cdots, v'_r)$$

for all $v' \leq_i v, v_1 \leq_1 v'_1, \cdots, v_{i-1} \leq_{i-1} v'_{i-1}, v_{i+1} \leq_{i+1} v'_{i+1}, \cdots, v_r \leq_r v'_r$.

A constraint $c_S \in C$ is monotonic *iff c_S is monotonic with respect to all variables of S.*

Immediately we have the following result.

Lemma 2 *A non-binary linear arithmetic inequality is a monotonic constraint.*

Another example of a monotonic constraint is

$$x * y \leq z, D_x = D_y = D_z = \{1, \ldots, 100\}.$$

For finite domain constraints, our definition of monotonic constraints is more general than the monotonic functions defined in [Hyv92].

In order to achieve arc consistency on monotonic constraints, the REVISE in algorithm BC should be modified as in Fig 4.2. It is important to note that the new algorithm doesn't require an explicit projection function. At the initialization phase of BC, for any constraint c and $i \in vars(c)$, we explicitly store the particular ordering of each domain involved which makes c monotonic with respect to i.

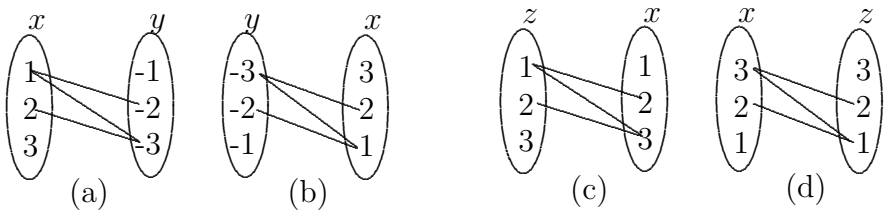
Example. Consider a constraint network with $x + y < 0, y + z > 0, z - x < 0$ and $D_x = D_y = \{1, 2, 3\}$ and $D_z = -3, -2, -1$. We draw the constraints c_{xy}

```
procedure REVISE(c_j, Q)
begin
  for each x_i ∈ vars (c_j) do
  begin
    ∀k ∈ 1..r, v_k ← the greatest value in D_k wrt x_k;
    DELETE = 0;
1.  while ( (v_1, ···, v_r) ∉ c_j) do
      begin
        remove v_i from D_i;
        if D_i is empty then
          Exit and report inconsistency;
        DELETE = 1;
        v_i ← the greatest value in D_i
      end
    if DELETE then
      Q ← {c_k ∈ C | x_i ∈ vars (c_k)}
  end
end
```

Figure 4.2: Procedure REVISE for monotonic constraints

and c_{xz} in Fig 4.3 where the values in a domain is in increasing order from top to bottom. To revise c_{xy}, we first revise x. According to Fig 4.3(a), the greatest values in x and y are 3 and -3 respectively. For $3 + (-3) = 0$, 3 is removed from D_x. The next greatest value is 2. Now we can stop revising x since $2 + (-3) < 0$. To revise y, the greatest values in y and x are -1 and 1 respectively. -1 is removed since $(-1) + 1 = 0$. The next greatest value -2 is supported by 1 in x. We finish the revision of y and thus the constraint c_{xy}. Similarly, when c_{yz} is revised, -3 in D_y and 1 in z are removed. After revising c_{xz} and c_{xy} in sequence, we have an empty domain for x.

Theorem 5 *Given a network (N, D, C) which contains only monotonic constraints, it can be made arc consistent in time complexity of $O(er^3d)$ if the complexity of evaluating $c(v_1, \cdots, v_r)$ is $O(r)$.*



c_{xy} monotonic wrt x (a) and y (b) c_{xz} monotonic wrt z (c) and x (d)

Figure 4.3: An example for enforcing AC on monotonic constraints

The sketch of the proof is as follows. In a similar fashion to Proposition 3, we can show that arc consistency can be achieved on monotonic constraints.

The complexity of the algorithm depends on the number of executions of line 1 in the REVISE of Fig 4.2. Consider expanding one execution of the algorithm BC according to line 1. Executions of line 1 fall into two groups. One group contains executions without any value removed. The other group contains those with at least one value removed. Because REVISE can be executed at most r^2ed times, the complexity of executions of the first group is r^3ed under the assumption of the linear time evaluation for c. As for the second group, we cluster the computation around variables. Now the total computation is

$$\sum_{i=1}^{n} r \cdot (d_{i,1} + d_{i,2} + \cdots + d_{i,l}) \leq \sum_{i=1}^{n} r \cdot d \leq er^2d$$

where $d_{i,j}(j : 1..l)$ denotes the number of elements removed from D_i in some execution of the while loop in line 1 on i, and l is the total number of such executions. Since $n \leq r \cdot e$, the complexity of the second group will be smaller than the first group and thus the complexity of the algorithm is $\mathcal{O}(er^3d)$. $\square$

We remark that, as in Proposition 2, by using the special semantics of monotonic constraint, it may be possible to decrease the complexity of the arc consistency algorithm by a factor r.

We now would like to briefly discuss how to embed the monotonic arc consistency algorithm into a fine grained algorithm. AC-6 is used as an example. To simplify the discussion, we will illustrate the idea using a binary monotonic constraint c_{xy} given in Fig 4.4. In the initialization phase of AC-6 for c_{xy}, we only need

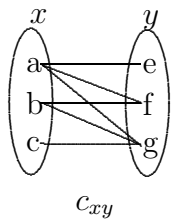


Figure 4.4: A monotonic constraint

the least value in x and the greatest value in y (no need to find support for any other values because of the monotonic property). The ordering used here makes a the least value in x and g the greatest value in y. In the implementation, we can easily associate the values a and g with the revision process for c_{xy}. Now, any deletion of values of b, c, e, or f by other constraints will not invoke the revision of constraint c_{xy}. Only when a (or g) is removed will monotonic constraint revision be invoked. After its execution, the monotonic revision process, will be associated to the current least (or greatest) values. This approach conforms to the *lazy* principle behind AC-6.

4.4 Linear Equations

We now consider non-binary networks where the constraints are linear equations. Bounds-consistency on a network of equations no longer implies arc consistency when the domains are discrete although it does if we relax the discrete domains to Z-intervals.

Unfortunately, the problem of enforcing arc consistency on a single linear equation is a very hard problem. Recall from the definition (see Section 4.1) that the

arc consistency of a single constraint implies its satisfiability.

Consider the one-line integer programming problem: Is there a solution for

$$a_1x_1 + a_2x_2 + \cdots + a_rx_r = b$$

where $a_1, \cdots, a_r, b$ are constant positive integers, and $x_i \in \{0, 1\}$ for all $i \in 1..r$? It is NP-complete [Pap81]. Therefore enforcing arc consistency on a single equation is NP-complete. Of course, to enforce arc consistency on a network of linear equations is also NP-complete.

This observation highlights the computational difficulty with arc consistency on non-binary constraints. Arc consistency is tractable on linear inequalities (monotonic constraints), but intractable on arbitrary linear constraints such as linear equations. Let us look at the arc consistency on different representations of a constraint. One can choose to represent a linear equation

$$a_1x_1 + a_2x_2 + \cdots + a_rx_r = b$$

as two inequalities

$$a_1x_1 + a_2x_2 + \cdots + a_rx_r \leq b$$

$$a_1x_1 + a_2x_2 + \cdots + a_rx_r \geq b.$$

If the discrete domains are approximated by Z-interval, arc consistency enforcing gives the same resulting domains on both representations, in the same time complexity. Without any approximation, the two inequalities can still be made arc consistent as shown in Section 4.3. However, the resulting domains don't make the original equation arc consistent since arc consistency on inequalities only ensures the satisfiability of each inequality separately and not both. This also shows that on a network of linear inequalities, enforcing relational path consistency is NP-

complete while relational arc consistency can be achieved in polynomial time. For the relational consistency, see Chapter 7.5.

4.5 Related Work

A substantial body of work on non-binary constraints comes from the continuous domain rather than the discrete domain. The early work [Hyv92, OV93] focused mainly on issues of correctness, convergence, and searching strategy etc. In more recent work the emphasis is on using numerical methods such as Newton methods [BMVH94] and Aitken acceleration [LL98] to speed up convergence. Our definition of bounds-consistency is similar to *arc B consistency* [Lho93] and *interval consistency* [BMVH94, DMP91] but differs in that bounds-consistency uses an inward rounding operation. The time complexity of filtering algorithms in the continuous domain, on the other hand, is usually not treated for the following reasons. Firstly for real/rational intervals, the *interval Waltz filtering* algorithm may not terminate given arbitrary linear constraints [Dav87]. Secondly for floating point intervals, the domain is huge and thus the worst case time complexity may not be of practical relevance. The efficiency is gained not so much by reducing the time complexity, but by faster convergence using numerical methods. In [Lho93], existing complexity results from general discrete arc consistency algorithm are used to bound their filtering algorithms. Thus, the work in the continuous case does not directly help in obtaining more efficient algorithms and the consequent time complexity analysis in the discrete case.

Non-binary discrete constraints, including integer linear constraints [NW88], are widely used for modeling and solving real life problems in finite domain CP systems [CD96, ILO00, VH89] underlying which is essentially a CSP model. Such systems employ various techniques based on the propagation of bounds for arithmetic constraints [Lau78]. The use of bounds based propagation techniques is not

new and originated as early as in 1978 [Lau78]. However, the efficiency and level of consistency of such techniques are not studied and described in detail.

In this chapter, we have addressed the question of what level of consistency can be achieved efficiently on non-binary linear constraints. The observation from Section 4.4 shows that arc consistency on non-binary linear equations is not tractable. We carefully introduce and formalize the notions of bounds-consistency in the context of discrete networks. It is shown that arc consistency for the networks of linear inequalities can be achieved with a simple AC-3 like algorithm in time complexity of $\mathcal{O}(er^3d)$. Where an efficient implementation of REVISE is possible as is the case with the projection of linear inequalities, the time complexity is improved to $\mathcal{O}(er^2d)$.

Given that arc consistency on a single non-binary constraint can be NP-complete, we identify a general class of monotonic constraints (which need not be linear) for which arc consistency can be efficiently enforced.

The work reported in this chapter extends the results on binary network in [VHDT92] to non-binary network. It also complements the GAC-schema [BR97] by showing the difficulty of arc consistency enforcing and identifying some tractable class of constraints.

Some open questions are suggested by the results here. What are other general classes of non-binary constraints for which enforcing arc consistency is efficient? What is the optimal worst case time complexity for arc consistency on linear inequalities and monotonic constraints?

Part III

Solving Functional Constraints

Binary *functional constraints* are an important class of constraints in a constraint programming system [VH89, VHDT92]. In this part, a *variable elimination* method is developed to find solutions for a network of functional constraints both efficiently and elegantly. Two types of networks are considered: *static* networks and *incremental* networks. A network is static if all constraints in the network are known a priori. A network is incremental if constraints are added into the network incrementally and the satisfiability of the network is tested each time a constraint is added. For a static network of functional constraints, an algorithm with optimal worst case time complexity of $\mathcal{O}(ed)$ is designed, where e is the number of constraints and d is the size of the domain. For an incremental network of functional constraints, an incremental algorithm is designed with "almost" the same time complexity as that of the static one. The elimination method may also lead to efficient algorithms for networks containing both functional constraints and other kinds of constraints. For example, it is shown that a network of *0/1/All constraints* [Kir93] can be made *minimal* with a time complexity of $\mathcal{O}(e(d+n))$ that significantly improves the time complexity and level of consistency over existing work.

Chapter 5

Variable Elimination and Its Application

We know that CSP is NP-complete in general. However, in real life constraint satisfaction problems, there are many important classes of constraints which can be solved in polynomial time and thus are *tractable*. Binary functional constraints are one such class of constraints. For any value taken by one variable, a functional constraint allows at most one value for the other variable.

Functional constraints occur naturally in scene labeling [Kir93, KP88, PT93] and other problems [SS77]. More importantly, functional constraints are implemented in Constraint Programming (CP) systems as a primitive. For example, in Constraint Logic Programming (CLP) languages [DVHS$^+$88, JM94, CD96], "equality", a functional constraint, is essential. Even when a constraint store of a CP system does not initially contain any functional constraints, during search or execution of a constraint program, some constraints may become functional as a result of variable instantiation or as a result of domain reduction.

The functional constraint is first studied by van Hentenryck et al. [VHDT92]. They propose an efficient algorithm AC-5 to enforce arc consistency (AC) on a network of functional constraints. This algorithm has a time complexity of $\mathcal{O}(ed)$

while $\mathcal{O}(ed^2)$ is the optimal worst case time complexity of an algorithm, for example AC-6 [Bes94] and AC-3.1 (Chapter 3), to enforce AC on a network of general constraints. Later, networks of functional constraints were found to be tractable by Kirousis [Kir93] and Cooper et al. [CCJ94].

In this chapter, we propose a variable elimination method to solve a network of functional constraints. Based on the elimination method, an algorithm is designed to globally solve a *static* network of functional constraints, where constraints are given a priori. It has the optimal time complexity of $\mathcal{O}(ed)$, the same cost as the fastest arc consistency enforcing algorithm AC-5.

We also demonstrate the application of the variable elimination by investigating a class of 0/1/All constraints, also called *implicational* constraints in [Kir93]. 0/1/All constraints represent a significant class of scene labeling problems. Cooper et al. and Kirousis studied them and proposed polynomial algorithms to find a solution for a network of 0/1/All constraints independently [CCJ94, Kir93]. It is interesting to find that the variable elimination method can be employed to efficiently solve 0/1/All constraints. Specifically, we show that the 0/1/All network can be made *minimal* in $\mathcal{O}(e(d+n))$, improving the early results in terms of both time complexity and the level of consistency.

Preliminaries and definitions are given in Section 5.1. The variable elimination method and an algorithm to solve a static network of functional constraints are presented and analyzed in Section 5.2. The algorithm for solving 0/1/All network is presented and analyzed in Section 5.3. This chapter is concluded with a discussion on related work.

5.1 Functional Constraints

Functional constraints and minimal network are reviewed in this section.

Definition 14 *A binary constraint c_{ij} is* functional *iff for every $a \in D_i$ (respec-*

tively $b \in D_j$*) there exists at most one* $b \in D_j$ *(respectively* $a \in D_i$*) such that* $c_{ij}(a, b)$*holds.*

The most common varieties of functional constraints are equality constraints. A typical functional constraint in linear arithmetic is a binary linear equation like $2x + 3y = 5$. Functional constraints also include nonlinear equations like $x = y^2$ where $x, y \in 1..10$. Other applications of functional constraints include problems in scene labeling [Kir93].

Definition 15 Functional network *denotes a network with* only *functional constraints.*

The minimal network was defined first by Montanari [Mon74]. In its original definition, the constraint graph is forced to be a complete graph. However, in the following definition, we consider only those constraints explicitly in the network.

Definition 16 *A binary constraint network* (N, D, C) *is* minimal *if each pair of values allowed by each constraint* $c \in C$ *is a part of a solution of the network.*

In general, a constraint network may not be minimal. As usual, we can remove those pairs which can not be extended to any solution of the network. If we regard each constraint as a set, this process reduces the constraint to its minimal size. That is why the word *minimal* is used. Note that the minimality in the definition is with respect to the topological structure (associated graph) of the network. So a network is minimal doesn't mean it is minimal among *all* equivalent networks. There may be many networks, with different topological structures, *equivalent* to a given network.

Since CSP is NP-complete, making a network minimal is consequently an NP-hard task in general. However, in polynomial time, a functional network can be transformed into another network, with a possibly different constraint graph and

constraints, which is minimal with respect to the new topological structure. A network of 0/1/All constraints can also be made minimal in polynomial time.

Remark. In the study of arc consistency, given a value $a \in D_i$ and a constraint c_{ij}, the support(s) of a is not assumed to be known beforehand. In other words, we have to try each value in D_j to find a support. However, for a functional constraint c_{ij} and any value $a \in D_i$, we assume that in constant time, we can determine whether a has a support in D_j and its value if there is one.

5.2 An Elimination Algorithm

In this section, we introduce the variable elimination method and an algorithm to globally solve a *static* network of functional constraints.

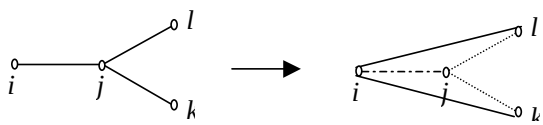


Figure 5.1: Elimination of variable j

Consider a functional constraint c_{ij} in a static network (N, D, C). Variable j (or i) can be eliminated in the following way to get a new network (N, D, C'). Let $C_j = \{c_{jk} \mid c_{jk} \in C\} - \{c_{ji}\}$ denote the set of all constraints, except c_{ij}, involving variable j. $C' = (C - C_j) \cup \{(c_{jk} \circ c_{ij}) \cap c_{ik} \mid c_{jk} \in C_j\}$. Recall that $\circ$ denotes a composition of two relations.

In the new network, there is only one constraint (c_{ij}) on j and thus j can be regarded as being eliminated.

For example, in Fig 5.1 after the elimination of j, the original network of $\{c_{ij}, c_{jl}, c_{jk}\}$ becomes $\{c_{ij}, c_{il}, c_{ik}\}$ where c_{il} is the composition of c_{ij} and c_{jl}, and c_{ik} the composition of c_{ij} and c_{jk}.

The variable elimination process has the following property.

Property 1 *Given a functional network (N, D, C) and a constraint $c_{ij} \in C$. The new network (N, D, C') after the elimination of variable j or i is equivalent to (N, D, C).*

Proof. Without loss of generality, assume variable j is eliminated. a_i denotes a value in the domain of variable i.

Assume $(a_1, a_2, \cdots, a_n)$ is a solution of (N, D, C). We need to show it satisfies C'. The difference between C' and C is that it has new constraints $C'_j = \{c'_{ik} \in C' \mid \exists c_{jk} \in C_j\}$. Consider any $c'_{ik} \in C'_j$. Since $c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}$, where $c_{jk}, c_{ik} \in C$, $(a_i, a_j) \in c_{ij}$ and $(a_j, a_k) \in c_{jk}$ imply $(a_i, a_k) \in c'_{ik} (\in C)$. Hence a_i and a_k satisfies c'_{ik}.

Conversely, we need to show that any solution $(a_1, a_2, \cdots, a_n)$ of (N, D, C') is a solution of (N, D, C). Given the difference between C' and C, it is only necessary to show the solution satisfies $C_j = \{c_{jk} \in C \mid c_{jk} \notin C'\}$. Consider any $c_{jk} \in C_j$. According to variable elimination, we have $c'_{ik} \in C'$ such that $c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}$. We know $(a_i, a_j) \in c_{ij}$ and $(a_i, a_k) \in c'_{ik}$. Since $c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}$, there exists $u \in D_j$ such that $(a_i, u) \in c_{ij}$ and $(u, a_k) \in c_{jk}$. However, as $(a_i, a_j) \in c_{ij}$ and c_{ij} is functional, u has to be a_j. So, a_j and a_k satisfy c_{jk}. $\square$

We assume the constraint graph of a functional network is connected in the rest of this section. If it is not connected, the following presentation still perfectly applies to each connected component.

Definition 17 *A network (N, D, C) is* canonical *if and only if*

$$C = \{c_{i1}, c_{i2}, \cdots, c_{i(i-1)}, c_{i(i+1)}, \cdots, c_{in}\}$$

for some i.

In other words, all constraints in a canonical network share one and only one common variable and form a tree with height of 1.

In terms of the property of variable elimination we can immediately reduce a network by eliminating one variable after another. Let N_E be the set of already eliminated variables and N_U be $N - N_E$. Take a variable j from N_U such that there exists a constraint c_{jk} where $k \in N_U$, and eliminate j, resulting in $N_E = N_E \cup \{j\}$ and $N_U = N_U - \{j\}$. The elimination process is repeated until there is one variable left in N_U. Now we show that the reduced network is canonical.

Assume $n \geq 2$ and the constraint graph of the network is connected. First, we show that the elimination process terminates with $|N_U| = 1$. When j is chosen to be eliminated, there always exists c_{jk} with $k \in N_U$ because the variable elimination preserves the *connectedness* of the graph.

Next, we show the property that there are no constraints between variables in N_E, and for any variable $j \in N_E$, there is a unique constraint in the network involving j (of course, the other variable in the constraint is from N_U). It is easy to be verified when the first variable is eliminated. Assume the property holds after m rounds of elimination. Now, let us choose $j \in N_U$ and a constraint c_{jk} where $k \in N_U$. If there is no constraint between any variable in N_E and j, the property still holds after the elimination. Otherwise, let constraint $c_{k'j}$ be the unique one on j and $k' \in N_E$. After the elimination of j, $c_{k'j}$ is discarded while a unique new constraint $c_{k'k}$ is introduced. In this case, the property holds again after the elimination. In summary, no matter what the original network is, the final reduced network has one variable not eliminated, called *free variable*, and only one constraint between the free variable and every other variable. Hence, it is canonical. The reduced network is also called a *canonical form* of the original one.

Given the special structure of a canonical form, an arc consistency enforcing algorithm is sufficient to make it minimal. An instantiation of the free variable will lead to a valid instantiation of all other variables. This instantiation of all variables makes up a solution of the network.

A close look at the above reduction reveals that a proper ordering of variables

```
procedure Static-Eliminate (inout (N, D, C), out consistent)
{ consistent ← true;
1. Take any i ∈ N, L ← {j | ∃c_ij ∈ C};
  while (L ≠ ∅) {
    Select and delete j ∈ L;
2.  D_i ← {x ∈ D_i | ∃y ∈ D_j such that (x, y) ∈ c_ij};
    if (D_i = ∅) then {
      consistency ← false;
      return;
    }
3.  for each c_jk ∈ C − {c_ij} {
4.    c'_ik ← c_jk ∘ c_ij;
5.    C ← C − {c_jk};
      if ∃c_ik ∈ C then c'_ik ← c'_ik ∩ c_ik;
      L ← L ∪ {k};
      C ← C ∪ {c'_ik};
    }
  }
}
```

Figure 5.2: Elimination algorithm for static functional constraints

to eliminate is necessary to avoid redundant composition of constraints. In a static network, we simply choose any variable as a free variable and eliminate all its neighbors until there is no new neighbors generated (note that elimination will produce new neighbors for the free variable). A detailed algorithm is listed in Fig 5.2. The algorithm uses a set L to hold the neighbors of the free variable to be eliminated. It also revises (line 2) the domain of the free variable in each elimination step, in order to check the satisfiability of the network. Note in the algorithm, it is not necessary to revise the domain of D_i in the **for** loop (at line 3) because the revision will ultimately be done later by line 2.

Example. Fig 5.3 shows how the algorithm works step by step. After k is eliminated, we introduce a new neighbor k_1 for i. In the final step we eliminate k_1 and revise the domain of i by removing a. Now we obtain a canonical network. Since the domain of the free variable i is not empty, the original constraint network

is satisfiable.

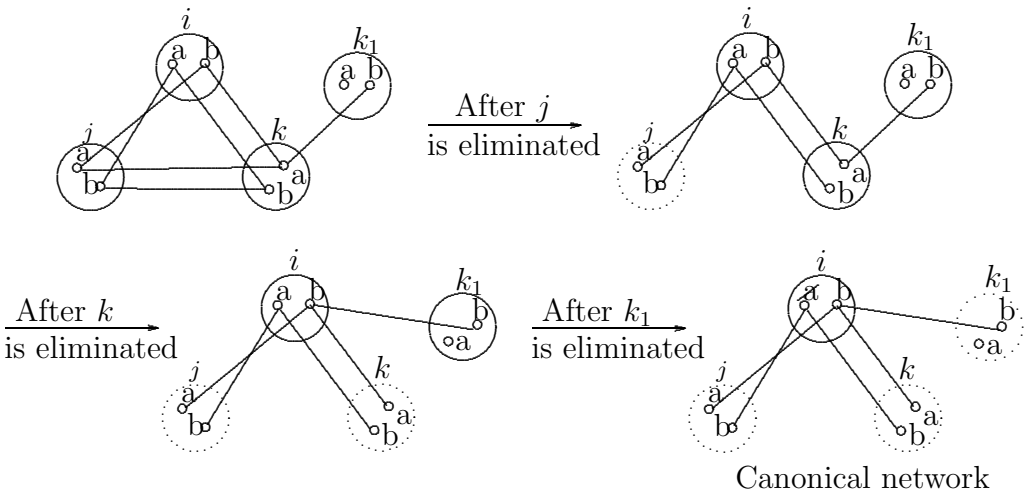


Figure 5.3: An example for eliminating variables in a constraint network

Theorem 6 *Given a functional network, its satisfiability can be determined by the algorithm in Fig 5.2 in $\mathcal{O}(ed)$ time.*

Proof. Algorithm `Static-Eliminate` finally changes the original constraint network to a canonical one. The revision of the domain of the free variable in line 2 in Fig 5.2 will tell the satisfiability of the network.

The complexity of the algorithm depends on the number of executions of line 3 in Fig 5.2. Let C^0 be the initial set of constraints in the network. Any constraint $c_{jk} \in C^0$ will be checked only once because it is excluded from further consideration in line 5. Any new constraint produced in line 4 will never satisfy the loop condition in line 3 because it is directly incident on the free variable i. Therefore it is only considered once at line 2. So, the **for** loop is executed at most $|C^0| = e$ times. Each operation in the algorithm can be done in time of at most d. Hence the algorithm has a complexity of $\mathcal{O}(ed)$ $\square$

Corollary 1 *A functional network can be transformed in $\mathcal{O}(ed)$ time to an equivalent network which is minimal.*

Proof. A canonical form can be obtained by the elimination algorithm. It can be verified that every value of the free variable in the canonical network is part of a solution of the network thanks to the revision of the domain of the free variable. To make the network minimal, it is only necessary to revise the domains of all the other variables. □

Without the revision of the domain of the free variable in the elimination algorithm, the canonical form may not be minimal. For example a network with $D_x = D_y = \{a, b\}$ and $C = \{c_{yx}, c_{xz}\}$ where $c_{yx} = \{(a, a)\}$ and $c_{xz} = \{(a, a), (b, b)\}$ is canonical but not minimal. It is minimal if we remove the value b from the domain of x (consequently (b, b) will be implicitly removed from c_{xz}).

5.3 Solving 0/1/All Constraints

As pointed out in the previous section, a network with mixed types of constraints may benefit from the variable elimination. In this section we investigate efficient algorithms to solve a network of 0/1/All constraints. More information and motivation can be found in [Kir93, CCJ94].

Definition 18 ([CCJ94]) *A constraint c_{ij}, is a* directed 0/1/All constraint *if for each value $a \in D_i$ (c_{ij} satisfies the following):*

1. for any value $b \in D_j$, $(a, b) \notin c_{ij}$; or

2. for any value $b \in D_j$, if $\exists u \in D_i$ such that $(u, b) \in c_{ij}$, then $(a, b) \in c_{ij}$; or

3. there is a unique value $b \in D_j$, $(a, b) \in c_{ij}$.

A constraint c_{ij} is functional *if either condition 1 or condition 3 is satisfied for both c_{ij} and c_{ji}. A constraint c_{ij} is* 0/1/All constraint *iff both c_{ij} and c_{ji} are directed 0/1/All constraint. A* two-fan *constraint c_{ij}, also called an "All" constraint, is a constraint where there exist $a \in D_i$ and $b \in D_j$ such that $c_{ij} = (\{a\} \times D_j) \cup (D_i \times$*

$\{b\}$). *A* fan-out *constraint is a constraint* c_{ij} *such that* $c_{ij} = \{a\} \times S$ *for some* a *in* D_i *and* $S \subset D_j$*, or* $c_{ij} = S \times \{b\}$ *for some* b *in* D_j *and* $S \subset D_i$*.*

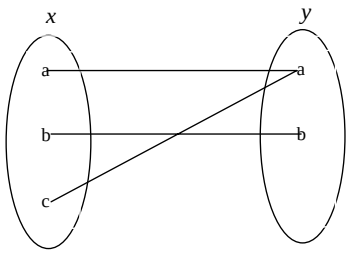


Figure 5.4: c_{xy} is a directed 0/1/All constraint but c_{yx} is not

Condition 1 in the definition means that there is no support in D_j for a value a. Condition 3 means that there is a unique support for the value a. If neither condition 1 nor condition 3 is true for a, condition 2 implies that if one value in D_i is related to some value $b \in D_j$ and it is not the only one then all values in D_i are related to b. That c_{xy} is directed 0/1/All does not mean that c_{yx} is also directed 0/1/All. Consider the example in Fig 5.4. For each value in x, there is a unique support in y. So c_{xy} is directed 0/1/All. However, for c_{yx} and value $y.a$, neither condition 1 nor 2 is satisfied. To satisfy condition 2, we need $(a, b) \in c_{yx}$ because of $(b, b) \in c_{yx}$. Contradiction. Hence c_{yx} is not directed 0/1/All.

Examples of two-fan constraint and fan-out constraint are illustrated in Fig 5.5.

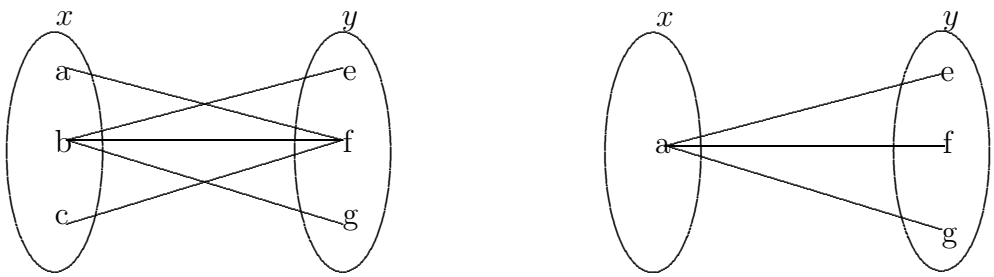


Figure 5.5: Two-fan (left) and fan-out (right) constraints

0/1/All constraints have two nice properties which can be verified in accordance with the definition.

Property 2 ([CCJ94]) *After enforcing arc-consistency on 0/1/All constraints, any 0/1/All constraint is trivial, bijective or two-fan.*

A *trivial* constraint is either empty or universal.

Property 3 ([CCJ94]) *The set of 0/1/All constraints is closed under the operations involved in path consistency: 1) Intersection of constraints; 2) Composition of constraints.*

Definition 19 ([vBD95]) *A binary relation c_{ij} represented as a $(0,1)$-matrix is* row convex *if and only if in each row all of the ones are consecutive; that is, no two ones within a single row are separated by a zero in that same row.*

Both functional and 0/1/All constraints are row convex. For row convex constraints there is the result:

Theorem 7 ([vBD95]) *For a path-consistent complete constraint network, if there exists an ordering of the domains $D_1, \cdots, D_n$ such that all constraints are row convex, the network is minimal and strongly n-consistent.*

It is obvious that a path consistency enforcing algorithm will make the 0/1/All constraint network minimal by Theorem 7 and Property 2 and 3, and thus the problem is solved. However, the complexity of a typical path algorithm is high, such as $\mathcal{O}(n^3d^3)$ in [MH86] and $\mathcal{O}(n^3d^2)$ in [DBVH97]. The rest of this section presents more efficient algorithms.

Remark. For any 0/1/All constraint c_{ij} and any value $a \in D_i$, we assume in constant time we know whether $i.a$ has no support, one support, or all elements in D_j as supports. For the case of one support, the support can be accessed in constant time.

5.3.1 Arc Consistency on 0/1/All Constraints

First we enforce arc consistency on 0/1/All constraints. As shown by the Property 2, it simplifies the presentation and analysis of our algorithms although it may not be necessary for solving 0/1/All constraint network.

The *categorisation* lemma in [CCJ94] shows that in an arc consistency algorithm a 0/1/All constraint can be dealt with in the same way as functional and monotonic constraints in [VHDT92]. An adaptation of AC-5 (see [VHDT92]) to 0/1/All constraints will result in an algorithm of complexity of $\mathcal{O}(ed)$.

5.3.2 The Elimination Phase

After making the 0/1/All constraints arc consistent, we remove the universal constraints. The new network, called an *1/All network*, contains only functional and two-fan constraints.

Definition 20 *Given a 1/All network, a* functional block *is a maximum connected sub graph of the network, which has a spanning tree containing only functional constraints.*

Definition 21 *A functional constraint c_{ij} is* bivalued *if and only if $|D_i| \leq 2$ and $|D_j| \leq 2$.*

The algorithm to solve 0/1/All constraints is presented in Fig 5.6. First, elimination is applied only to the functional blocks (line 1), which is implemented by line 1 in the algorithm `Eliminate` shown in Fig 5.7. Now, there is a free variable for each functional block. Let us hide the eliminated variables together with all constraints between the free variable and eliminated variable from the network (see line 2 to line 3 in Fig 5.6). The solution of the new network can be easily extended to a solution of the original one by instantiating the eliminated variables according

to the value of the free variables. The algorithm `A` in line 4 is to find a solution for the new network, and will be discussed in next subsection.

Algorithm OA-algorithm
{ Enforce arc consistency on (N, D, C) and remove universal constraints;
1. Eliminate $((N, D, C)$, *consistent*);
  if *consistent* **then** {
2. $EV \leftarrow$ { all eliminated variables };
    $FC \leftarrow$ { constraints between free variables and eliminated variables};
    $N \leftarrow N - EV$;
3. $C \leftarrow C - FC$;
    Enforce arc consistency on (N, D, C) and remove universal constraints;
4. $A((N, D, C))$;
    instantiate variables in EV wrt FC;
  }
  else report inconsistency
}

Figure 5.6: Algorithm for 0/1/All constraints

The algorithm `Eliminate` (in Fig 5.7) differs from the algorithm `Static-Eliminate` only in that the former deals with a network with several functional blocks.

An immediate question is, after elimination, what kinds of constraints there are in the new network. We know that constraints are generated by composition and then intersection of constraints during the elimination process. An exhaustive examination shows that there are only three types of constraints: bivalued functional; fan-out; and two-fan constraints (see the `switch` statement in line 2 of Fig 5.7) if no empty constraint (inconsistency) has occurred during elimination. The current functional constraints result from the intersection of two-fan constraints and thus allow only two valid values for each variable involved in the constraints. By enforcing arc consistency on the network and removing fan-out constraint (universal constraints), we have only bivalued functional constraints and two-fan constraints. If we continue variable elimination with respect to bivalued functional constraints, we will inevitably fall into a situation similar to an incremental network where the

time complexity of the algorithm becomes $\mathcal{O}(ed\alpha(2e, n))$ (see Chapter 5). Here we want an algorithm with complexity of $\mathcal{O}(ed)$.

```
procedure Eliminate (inout (N, D, C), out consistent)
{ FC = {c_ij | c_ij ∈ C is functional};
  V = {i, j | ∃c_ij ∈ FC};
  consistent ← true;
  while (V ≠ ∅) {
    select and delete i ∈ V;
    L ← {j | ∃c_ij ∈ FC};
    while (L ≠ ∅) {
      Select and delete j ∈ L;
      D_i ← {x ∈ D_i | ∃y ∈ D_j such that (x, y) ∈ c_ij}; // revise domain D_i
      if (D_i = ∅) then {
        consistency ← false;
        return;
      }
      for each c_jk ∈ C − {c_ji} {
1.      if c_jk ∈ FC then L ← L ∪ {k};
        c'_ik ← c_jk ∘ c_ij;
        C ← C − {c_jk};
        if ∃c_ik ∈ C then c'_ik ← c'_ik ∩ c_ik;
2.      switch (c'_ik) {
          case ∅ : consistency ← false;
                 return;
          case functional:
          case fan-out:
          case two-fan: c_ik ← c'_ik;
        }
    } // until L = ∅; process one connected component
  } // until V = ∅; process all components
}
```

Figure 5.7: Elimination algorithm for functional constraints

5.3.3 The A ("All") Algorithm

In this subsection, we study two-fan constraints (also called "All" constraints) and present an algorithm for a network with both two-fan constraints and bivalued functional constraints which is simply called *mixed network* in this subsection.

For ease of presentation, we introduce the following notations:

Definition 22 *Given a two-fan constraint c_{ij}. The* pivot *of c_{ij} in D_i is defined to be the value $a \in D_i$ such that $\forall b \in D_j$ $(a, b) \in c_{ij}$, and denoted by p_i^j. The* coordinate *of any value in D_j with respect to c_{ij} is defined to be p_i^j.*

A variable i is *fully two-fan constrained* if and only if all constraints incident to i are two-fan constraints. The values of the domain of a fully two-fan constrained variable fall into two classes. One, called the *pivot class*, consists of the pivots of all incident constraints, while the other, called the *nonpivot class*, includes all the other values.

A two-fan constraint c_{ij} can be simply represented by the two pivots $(p_i^j,\ p_j^i)$. p_i^j is the only value in D_i which is supported by any value in D_j, and every other value has a unique support, p_j^i, in D_j.

A search procedure is employed to solve the mixed network. Before presenting the algorithm, we highlight some properties of two-fan constraints. First recall an observation made in [CCJ94]. To emphasize its importance, we formalize it as follows.

Definition 23 *Given a network (N, D, C), an instantiation of a set of variables $S \subseteq N$ is* separable*, if it satisfies all constraints among S, and for any constraint $c_{ij} \in C$ between a variable $i \in S$ and a variable $j \in N - S$, c_{ij} allows j to take any value under the current instantiation of i.*

For a network with only one *two-fan* constraint c_{ij}, the instantiation of i by the pivot p_i^j is separable.

Proposition 4 *Given a CSP with network (N, D, C) and a separable instantiation of a set of variables. If the CSP is satisfiable, then the instantiation is part of some solution of the CSP.*

The correctness of the above proposition is immediate. This proposition implies that after a separable instantiation is found, we can exclude further consideration of those instantiated variables and all constraints involved in at least one of those variables. Thus, we get a smaller problem to work on. It can be shown that repeating this process will at last decompose the mixed network into a set of separable instantiations and the combination of them is a solution to the original problem.

The identification of a separable instantiation is achieved by the `A-propagate` procedure(Fig 5.8). It works as follows. First, select a starting variable i and instantiate it to a value a. The next step is to try to instantiate its neighbor variables that have not been done so yet. For any uninstantiated neighbor k such that there exists $c_{ik} \in C$, we have two cases. In the case that a is p_i^k, the identification procedure is stopped along the direction of c_{ik}. Otherwise, no matter whether c_{ik} is two-fan or functional, we have a unique choice in D_k and thus we need apply the identification process to the neighbors of k since in the direction of c_{ik} the instantiation has not yet been found to be separable. Finally we get a set of variables whose instantiation is separable. A trivial case is that the set of variables is N itself.

One problem in the procedure above is that the instantiation step for a variable may fail. This failure occurs when the instantiation step tries to instantiate a variable to two different values, which is a *contradiction*. It is also possible that there is no value to assign to a variable. This case is easier and can be addressed in the same spirit as discussed below. The algorithm in [CCJ94] simply returns to the starting variable and select the next value available. However, there is a better and faster way to resolve the failure.

Proposition 5 *In the procedure of identifying a set of variables with separable instantiations, if there is a contradiction, then for the starting variable there are at most two possible values leading to a solution of the problem.*

```
procedure A-Propagate(in a, i, N, out M, con, p1, p2)
{ L ← ∅;
  for each j ∈ N    a_j ← null;
  a_i ← a;
  con ← true;
  for each j such that c_ij ∈ C {
    if c_ij is bivalued functional {
      let (a_i, b) ∈ c_ij;
      a_j ← b; a_j.coordinate ← NIL;
      L ← L ∪ {(a_j, j)};
    } else { // two-fan constraints
      a_j ← p^i_j; a_j.coordinate ← p^j_i;
      L ← L ∪ {(a_j, j)};
    }
  }
  while (L ≠ ∅ and con) {
    Delete first element (b, j) from L
    for each c_jk
      if there is only one u such that (b, u) ∈ c_jk then
        if a_k = null then {
          L ← L ∪ {(u, k)};
          a_k ← u;
          u.coordinate ← b.coordinate;
        } else if x_k ≠ u then {
            con ← false;
            p1 = b.coordinate;
            p2 = u.coordinate;
            M ← all the other uninstantiated variables
        }
  } // until L = ∅ or not con;
}
```

Figure 5.8: A-Propagate for a network with two-fan constraints and bivalued functional constraints

Proof. Let the starting variable be i. If there is any bivalued functional constraint incident to i, the domain of i has at most two values and thus the proposition is true. Otherwise, let j be the variable where the contradiction occurs. Now let us trace the cause of the assignment of two different values v_1 and v_2 to the same variable j. If v_1 (respectively v_2) is caused by the instantiation of only one variable k, we keep tracing the cause of the instantiation of k. Otherwise, if the instantiations of more than one variables enforce j to take v_1 (respectively v_2), we just choose any variable and continue tracing the cause of its instantiation. The trace stops at the starting variable i. Let j_1 (respectively j_2) be the second last variable in tracing v_1 (respectively v_2). The instantiations of j_1 and j_2 must be $p^i_{j_1}$ and $p^i_{j_2}$. In the chain of variables from j_1 to j (respectively j_2 to j), $p^i_{j_1}$ (respectively $p^i_{j_2}$) of j_1 (respectively j_2) leads to the unique instantiation of any variable after it. Hence, as long as j_1 and j_2 take the same instantiations, we will have at least one contradiction at j. The only way to exclude the instantiation $p^i_{j_1}$ of j_1, or $p^i_{j_2}$ of j_2, is to instantiate i with the coordinate of $p^i_{j_1}$ and $p^i_{j_2}$. For all other values, the contradiction still remains at j. □

Definition 24 *Given a CSP, a value $a \in D_i$ is* almost globally valid *if and only if a is part of a separable instantiation of the CSP.*

Property 4 *Given a fully two-fan constrained variable i, we have*

- *for the nonpivot class of i: if one of the values is almost globally valid, then any value will also be almost globally valid;*
- *for the pivot class of i: if three of the values are almost globally valid, then any value will also be almost globally valid.*

Proof. Consider the nonpivot class of i. Since none of the values in the class is a pivot, we have to choose the pivot of any neighbors of i no matter what value we take for i from the nonpivot class. This proves the claim on nonpivot class. For

the pivot class, assume there exists some pivot which is not almost globally valid. We have at most two valid pivots in terms of Proposition 5, which contradicts the assumption that there are three valid values in the pivot class. □

Remark. Obviously, in a mixed network, a variable which is not fully two-fan constrained has at most two values in its domain since there is a bivalued functional constraint incident to it.

To find a solution of a mixed network, we only need to identify separable instantiations recursively until all the variables are instantiated. The algorithm `A` is given in Fig 5.9.

Algorithm A (**in** (N, D, C))
{ Select any value $a \in D_i$ for any variable $i \in N$
  A-Propagate$(a, i, N, M, consistent, i_1, i_2)$;
  if not *consistent* **then** { // contradiction occurs
    if $\exists c_{ij} \in C$ such that c_{ij} is bivalued functional **then** {
    let b be the other value in D_i;
    A-Propagate$(b, i, N, M, consistent, -, -)$;
    } **else** {
      A-Propagate$(p_i^{i_1}, i, N, M, consistent, -, -)$; // try the first
      if not *consistent* **then**
        A-Propagate$(p_i^{i_2}, i, N, M, consistent, -, -)$; // try the second
    } //end of the process of fully two-fan constrained variable
  } // end of the process of contradiction
  if *consistent* **then** {
1. $N \leftarrow N - M$;
      if $N \neq \emptyset$ **then** A$((N, D, C))$;
  } **else** report no solution for (N, D, C);
}

Figure 5.9: Algorithm for a network with two-fan constraints and bivalued functional constraints

Theorem 8 *A two-fan network can be made minimal in time complexity of* $\mathcal{O}(en)$.

Proof. The `A` algorithm is correct according to Proposition 4 and 5. The complexity of `A-Propagate` is at most e and it is called at most n times. The `A`

algorithm finds (at least) one solution to the network. To achieve the minimality, it can be slightly modified using Property 4 to check (a constant number of values of) each variable by `A-Propagate` rather than skipping a set of variables (M) in the main loop (see line 1). The time complexity is still $\mathcal{O}(en)$. □

Theorem 9 *A 0/1/All network can be enforced to be minimal in a time complexity of* $\mathcal{O}(ed + en)$.

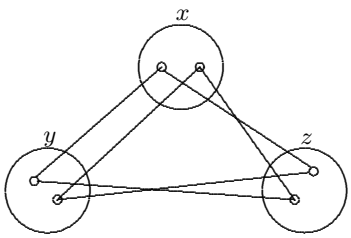
Proof. The transformation of an 0/1/All CSP to a 1/All CSP takes $\mathcal{O}(ed)$ (see Section 5.3.1). Consider the OA-algorithm (see Section 5.3.2). In the `elimination` procedure, it can be verified that the type of c'_{ik} at line 2 will be of only four types as shown in the algorithm according to Property 3. After the elimination procedure, there are only two-fan functions and bivalued functions. So, they can be made minimal in $\mathcal{O}(en)$ in accordance with the previous theorem. Now, a revision of the domains of all eliminated variables with respect to their free variables will make the whole network minimal. The complexity of the elimination procedure is still $\mathcal{O}(ed)$ since all operations involved in 0/1/All constraints can be done with a complexity of at most d. □

5.4 Related Work

For functional constraints, there are two classes of related work. The first class includes work [VHDT92, Liu95, AB96, Zha98] which is done mainly in the context of arc consistency. [VHDT92] focuses on efficient arc consistency algorithms on functional constraints and does not consider finding a global solution. [Liu95] proposes a more efficient arc consistency algorithm for *increasing* functional constraints. [AB96] introduces a new kind of consistency, *label-arc consistency*, and shows that functional constraints with limited extensions to other constraints can be solved globally. However, analytical results are not given there. [Zha98] embeds

the techniques dealing with functional constraint in arc-consistency algorithms in a similar way to [Liu95] and observes the problem of *conflict of orienting* from which all the above mentioned algorithms (except [VHDT92]) suffer. The other class of work (e.g. [Dav93]) aims at the tractability (in the context of NP-completeness) rather than more efficient algorithms.

Motivated by the above work, we propose an elimination algorithm to globally solve functional constraints both efficiently and elegantly. Its complexity of $\mathcal{O}(ed)$ to achieve minimality on static constraints is the same as that of the best algorithm achieving arc-consistency [VHDT92]. Note that AC is not sufficient to decide the satisfiability of a functional network. For example, the network in the following picture is arc consistent but has no solution.



Conceptually, variable elimination can be regarded as a generalization of Gaussian elimination. One of its application is that a system of linear equations with at most two variables per equation can be solved linearly ([AS80]).

Another relevant work in CSP is bucket elimination [Dec99]. It is designed mainly for a general CSP (NP-complete) and has been useful as an abstract tool. As such the time and space complexity of the algorithm in [Dec99] is high. Our work here may motivate more efficient bucket elimination algorithms for special classes of constraints.

The directly related work on 0/1/All constraints are [CCJ94] and [Kir93], both of which give a sequential algorithm with time complexity of $\mathcal{O}(ed(n+d))$ to find one solution (note that the non-binary 0/1/All constraints system defined in [Kir93] can be treated as a binary constraint system). Here, we obtain better results

with a time complexity of $\mathcal{O}(en)$ for a network with only "All" constraints and $\mathcal{O}(e(d+n))$ for network with 0/1/All constraints. Furthermore, minimality of the network is also achievable with the same time complexities. Thus, compared with [CCJ94, Kir93], we obtain a higher degree of consistency on 0/1/All constraints, with more efficient algorithms.

Chapter 6

Solving Functional Constraints Incrementally

The constraint store of a Constraint Programming (CP) system can be modeled as a CSP and processed by the help of techniques developed to solve a CSP. Constraints are added to this constraint store as a CP program is executed, while in the study of CSP all constraints are assumed to be known a priori.

It is interesting to study the incrementality of the store to design more efficient CSP techniques and thus improve the efficiency of the hosted CP system. It is also necessary to do so since CP systems play a key role in the successful application of CSP to real life problems and wider areas across AI and OR.

Given that functional constraints are primitive in CP systems [VHDT92], it is worthwhile to study functional constraints in an incremental context. Due to their incremental nature, arc consistency enforcing algorithms can still be directly used, without compromising efficiency, in an incremental CSP. However, a direct employment of some other algorithms for static CSP may be inefficient. For example, to invoke the variable elimination process, developed in the previous chapter, each time a constraint is added is not efficient.

An *incremental algorithm* is proposed in this chapter. It solves an incremen-

tal system of only functional constraints in $\mathcal{O}(ed\alpha(2e, n))$ where α is the inverse Ackermann function.[1] It is significant to observe that incremental solving can be achieved with almost the same cost as the static algorithm in previous chapter.

In a general CSP consisting of both functional and non-functional constraints, the algorithm above while efficient does not establish global consistency on as many constraints as possible. We present another algorithm for a general CSP which establishes global consistency, in $\mathcal{O}(ed^2 \log e)$ time, on all constraints as long as they are between variables connected through a path of only functional constraints. This time complexity is still close to that of general arc consistency algorithms typically used in CP systems.

6.1 Incremental Network

In a finite domain CP system, a *constraint store*, essentially a constraint network, is maintained incrementally. A constraint may be added to or removed from the constraint store during the execution of a program. The constraint solver is required to determine whether the constraint store is consistent (to a certain degree) each time a new constraint is added. To capture the incremental property of a constraint store, we introduce the notion of an *incremental system* or *incremental network*. Initially at time 0, the system is empty. At any later time t, some new variables with their associated domains and constraints may be added to the system. In this chapter, when we refer to a functional constraint, we mean that it is functional when it is added into the system unless it is explained otherwise.

Usually in CP systems, the removal of constraints only happens during backtracking which is implemented by restoring the state of the constraint store together with its associated data structures.[2] So we consider only the addition of constraints

[1] The inverse Ackermann function grows extremely slowly and for all practical purposes, we have $\alpha(2e, n) \leq 4$.

[2] Arbitrary removal of constraints is not provided in most CP systems. It remains a research

into the store. More details on constraint solving in the context of CLP languages and systems can be found in [JM94].

Recall that a *functional network* denotes a network containing only functional constraints. A *mixed network* here denotes one with functional constraints and other general constraints. A *functional block* of a constraint network denotes a maximum connected subnetwork which has a spanning tree containing only *explicit* functional constraints. An *explicit* functional constraint is one which is functional when it is added into the system.

For example, the network in Fig 6.1 (a) is functional. The part of a network shown in Fig 6.1 (b) is a functional block where c_{13} drawn with dark lines is not functional.

6.2 Solving Incremental Functional Networks

To solve an incremental functional network, a naive approach is to apply the variable elimination method developed in Chapter 5 to the network each time a new constraint is added. It leads to an algorithm with worst case complexity of $\mathcal{O}(e^2d)$.

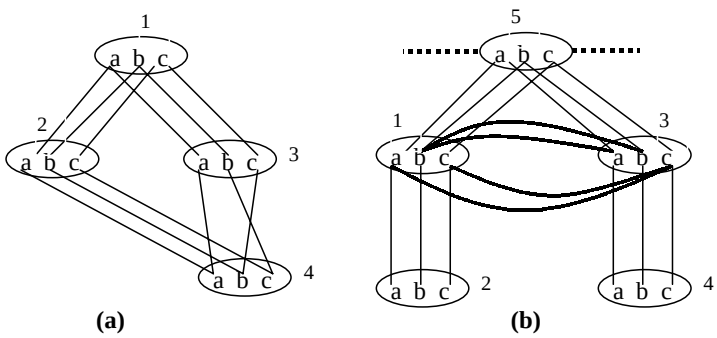


Figure 6.1: (a) A functional network; and (b) A functional block

To decide the satisfiability of the network, we observe that it is not necessary

challenge since semantically it leads to non-monotonic behavior. There is little work on the algorithmic aspect either.

to apply the variable elimination every time a new constraint is added. It suffices to do so when the newly added constraint forms a circuit with those already in the network.

Example. Consider the network in Fig 6.1(a). There are four variables $\{1, 2, 3, 4\}$ with the domain $\{a, b, c\}$ in the network. Constraints are added into the network in the order of c_{12}, c_{34}, c_{13} and c_{24}. The idea to deal with the constraints is illustrated as follows.

1. $c_{12} = \{(a, a), (b, b), (c, c)\}$. We first mark a variable, say 2, as *eliminated* with respect to c_{12}. Then mark 1 as *free*, and *revise* the domain of 1 with respect to c_{21}, i.e. remove values in D_1 which are not allowed by c_{21}.

2. $c_{34} = \{(a, a), (b, c), (c, b)\}$. Mark 4 as eliminated and 3 as free, revising D_3 wrt c_{34}.

3. $c_{13} = \{(a, a), (b, b), (c, c)\}$. Both 1 and 3 are free variables. The property we will maintain is that in any connected component of the constraint graph, there is only one free variable. Thus, we keep, say 1, as free and eliminate 3. Then revise D_1 with respect to c_{31}. So far, no real elimination has occurred but we can verify that there is a solution for the current network since D_1 (the domain of the free variable 1) is not empty.

4. $c_{24} = \{(a, a), (b, c), (c, b)\}$. Now both variables 2 and 4 have been eliminated. We require that a new constraint is allowed only on free variables rather than eliminated ones. Since an eliminated variable is marked with respect to a particular constraint, we can follow the chain of such constraints until a free variable is found. From variable 4 we get 3 and from 3 we get 1 which is free. Elimination also occurs during this tracing. A new constraint $c'_{14} = \{(a, a), (b, c), (c, b)\}$ is obtained by composing c_{13} and c_{34}, and 4 can now be marked as eliminated with respect to c_{14}. Now discard c_{34} from the

network. Similarly we trace 2 to free variable 1. The fact that 2 and 4 share the same free variable 1, implies a circuit is formed. Further eliminating 2 (wrt c_{12}) leads to a new constraint $c''_{14} = \{(a,a),(b,b),(c,c)\}$. The intersection of c'_{14} and c''_{14} gives $c_{14} = \{(a,a)\}$. Revising D_1 with respect to c_{14} causes $D_1 = \{a\}$. Discard constraint c_{24}, c'_{14}, and c''_{14} from the network. Now the network contains $\{c_{12}, c_{14}, c_{13}\}$ and is satisfiable.

Intuitively, we try to maintain any connected component in the network as close to its canonical form (see Chapter 5) as possible. □

The circuit detection in the above process can be achieved by (a) maintaining all connected components and (b) checking whether two given variables are in the same component. Operation (a) can be implemented efficiently by *union*; and operation (b) by *find* in the disjoint set union algorithm [Tar75]. Each *connected component* is maintained as *a set of variables* which is represented by a tree structure. Each variable has a field to point to its parent. Fig 6.2 shows a connected component containing variables form 1 to 6. The *representative variable* (6 in the example) of a connected component is at the root of the tree. It is distinguished from other variables by its field pointing to itself.

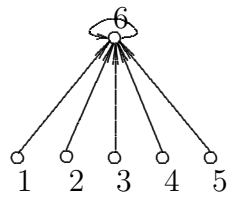


Figure 6.2: Data structure for variables in the same connected component

We define the following two operations:

- *Find(i)* returns the representative variable of the component to which i belongs.

- *Union(i,j)* merges two disjoint components represented by i and j, and returns one representative variable for the new component.

There are many ways to implement *union* and *find*. To obtain a fast algorithm, we use both strategies of union by rank and path compression. *Union by rank* makes the representative variable of the component with more variables the new representative for the merged component. For example, in Fig 6.3 the representative 4 points to 1 in the new component. After finding the path from a variable i to its representative variable, *path compression* is applied to relocate the pointers of all variables along the path to the root. For the example in Fig 6.4, after the representative of variable 5 is found, the parent of 5 is changed to be the root.

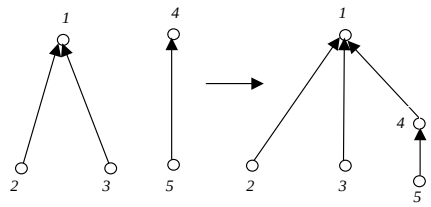


Figure 6.3: Example of $union(1, 4)$

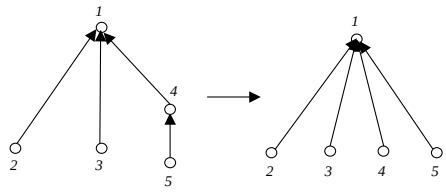


Figure 6.4: Example of $find(5)$

A new constraint c_{ij} merges two connected components into one if i and j are in different components; otherwise it results in a circuit. Checking whether a circuit arises after the addition of c_{ij} amounts to checking whether $find(i) = find(j)$.

Variable elimination is triggered by each occurrence of path compression in the *find* function. Assume $find(i)$ returns j as the representative variable and the

path from i to j is $i_1(=i), i_2, \cdots, i_k(=j)$. Elimination is applied to $i_{k-1}, \cdots, i_2$ in sequence. The eliminations are not necessary for enforcing global consistency but is simply an efficient way to prepare the network for the case of a circuit. Algorithms for *find*, *union*, and an auxiliary operation *init* are shown in Fig 6.5. *Init* is invoked whenever a new variable is added into the network. In the algorithms, $p[i]$ represents the parent of variable i, and $h[i]$ the rank, the number of variables in the tree rooted at i.

```
procedure Init(in i){ p[i] ← i; h[i] ← 0; }

procedure Union(in i, j) // i,j are roots
{ if (h[i] > h[j]) then {
      p[j] ← i;
      return i;
  } else {
      p[i] ← j;
      if (h[i] = h[j]) then h[j] ← h[j] + 1;
      return j;
  }
}

procedure Find(in i, inout(N, D, C))
{ if (i ≠ p[i]) then {
      k ← Find(p[i], (N, D, C));
      c_ki ← c_(p[i])i ∘ c_k(p[i]); // eliminate p[i]
      C ← (C − c_(p[i])i) ∪ {c_ki};
      p[i] ← k;
  }
  return p[i];
}
```

Figure 6.5: Disjoint set union algorithms for functional constraints

The algorithm for solving functional constraints incrementally is shown as procedure Pure-Eliminate in Fig 6.6. Its input includes the constraint c_{ij} to be added and a network (N, D, C) which is a previous output of the algorithm itself or $C = \emptyset$. It outputs a new network (N, D', C') and whether $(N, D, C \cup \{c_{ij}\})$ is sat-

isfiable. Pure-Eliminate will first test whether i and j are in the same connected component. Let k and l be the representative variables of i and j respectively. Immediately after the *find*(i) and *find*(j), there is a constraint directly between i and k (and a constraint between j and l) because of the elimination accompanying path compression.

If the test result is negative, the two connected components represented by k and l are unioned together, and the new constraint c_{kl} (line 2) added to the network. Let k be the representative element for the new connected component. Domain D_k is revised wrt c_{kl} (line 3) in order to determine the satisfiability of the network.

Otherwise, a circuit results from adding c_{ij}. The elimination is now triggered to establish global consistency. By eliminating variables i and j in sequence (line 1) with respect to c_{ki} and constraint c_{kj} respectively, the constraint c_{kk} is obtained (line 1). Note the two eliminations here are equivalent to one elimination and intersection discussed in the example above. D_k is then revised with respect to c_{kk} (line 3). Id_k in the algorithm denotes the identity relation on D_k. It means that an element in D_k is only related to (or supported by) itself.

---

procedure Pure-Eliminate (**inout** (N, D, C), **in** c_{ij}, **out** *consistent*)
{ *consistent* $\leftarrow$ **true**;
  $k \leftarrow$ *find*$(i, (N, D, C))$; $l \leftarrow$*find*$(j, (N, D, C))$;
1. $c_{kl} \leftarrow c_{jl} \circ (c_{ij} \circ c_{ki})$;
  if $(k \neq l)$ **then** {
2. $C \leftarrow C \cup \{c_{kl}\}$;
    if $(union(k, l) \neq k)$ **then** swap k and l;
  } **else** $c_{kk} \leftarrow c_{kk} \cap Id_k$;
3. $D_k \leftarrow \{x \in D_k \mid \exists y \in D_l \text{ such that } (x, y) \in C_{kl}\}$
  if $(D_k = \emptyset)$ **then** *consistent* $\leftarrow$ **false**;
}

Figure 6.6: Incremental elimination for functional constraints

Theorem 10 *Given that at time t, a total of e constraints are added into an incremental functional network which has n variables. The algorithm Pure-Eliminate*

determines the satisfiability of the network incrementally in worst case time complexity of $\mathcal{O}(ed\alpha(2e, n))$.

Proof. We first prove that the algorithm can decide the satisfiability of the network correctly.

Let C be the set of all constraints added incrementally into the network until time t, N the set of variables, and D the set of domains. Let the final output of Pure-Eliminate be (N, D', C'). Given the fact that all operations in the algorithm are variable eliminations and domain revisions, according to Property 1 (see Section 5.2), (N, D', C') is equivalent to the original network (N, D, C) because domain revisions obviously preserve the equivalence. If Pure-Eliminate returns inconsistency, there is no solution for (N, D, C). Otherwise, we only need to prove (N, D', C') is satisfiable. Specifically, for the root variable r of any connected component of the network (N, D', C'), every value in D_r can be extended to a solution of the network. The proof is given inductively on e.

When $e = 1$, the claim is true because there is now only one constraint and the revision of D_r is sufficient to make the network consistent.

The following proposition will be useful later.

Proposition 6 *Given any connected component* $G = (V, E)$ *in the graph of* (N, D', C') *where* V *and* E *are the set of vertices and edges respectively.* $\forall u \in V, \forall v \in N - V$, *there is no constraint between* u, v *in* (N, D', C').

This is a consequence of the use of *find* and *union* in the algorithm, and C is empty at time 0.

Assume for the first m constraints, the algorithm outputs network $(N^{m'}, D^{m'}, C^{m'})$ (graph $G^{m'}$) with the property claimed above. Now, consider the addition of a constraint c_{ij} to $(N^{m'}, D^{m'}, C^{m'})$. Let $(N^{(m+1)'}, D^{(m+1)'}, C^{(m+1)'})$ (graph $G^{m+1'}$) be the output of Pure-Eliminate. First assume i and j are in different connected components and with representative variables k and l respectively, and k is the

representative variable of the new component merging k and l in $G^{(m+1)'}$. Clearly all connected components except k and l in $G^{(m+1)'}$ are the same as those in $G^{m'}$. So according to Proposition 6, we only need to show that all values in D_k can be extended to a solution of the newly formed connected component. According to line 3 in Fig 6.6, for any value $a \in D_k$, there exists $b \in D_l$ satisfying constraint c_{kl}. The induction hypothesis implies that a can be extended to a solution of the component k, and b to a solution of the component l in $G^{m'}$. Proposition 6 shows that there is no constraint between variables in components k and l except the newly added c_{kl}. Hence, a combination of the two solutions of components k and l in $G^{m'}$ gives a solution to the new connected component in $G^{(m+1)'}$. In a similar manner, the claim can be proved when i and j are in the same connected component.

The complexity of the algorithm depends on the total number of executions of *find* and *union* because all other operations in the algorithm accumulate to a complexity of $\mathcal{O}(ed)$ while *find* has a complexity scaled up to d times of its usual complexity. For e constraints and n variables, we have $2e$ *find* operations and at most *n-1 union* operations. Hence, the complexity of the algorithm is $\mathcal{O}(ed\alpha(2e, n))$ [Tar75] where α is the inverse Ackermann function. □

6.3 On Incremental Mixed Networks

In practice, a CP system deals with mixed rather than functional networks. Obviously, the Pure-Eliminate algorithm can be directly applied to a mixed functional network by simply ignoring the non-functional constraints. While this is efficient in time, it does not fully exploit the properties of functional constraints. Functional constraints can interact with the non-functional ones through composition and intersection.

Example. Consider the functional block in Fig 6.1(b). There are variables $\{1, 2, 3, 4, 5, \ldots\}$ with domain $\{a, b, c\}$ in the network. Constraints are added into

the network in the order of c_{12}, c_{34}, c_{13}, some constraints on 5, c_{15} and c_{53}. They will be processed by Pure-Eliminate as follows:

1. $c_{12} = \{(a,a),(b,b),(c,c)\}$. Revise D_1 with respect to c_{21}.
2. $c_{34} = \{(a,a),(b,c),(c,b)\}$. Revise D_3 with respect to c_{43}.
3. $c_{13} = \{(a,c),(b,b),(b,a),(c,c)\}$, a non-functional constraint. So ignore it.
4. Some other constraints on 5 and other variables are added.
5. $c_{15} = \{(a,a),(b,b),(c,c)\}$. Because of the other functional constraints on 5, we mark 5 as free and 1 as eliminated.
6. $c_{53} = \{(a,a),(b,b),(c,c)\}$. Mark 5 as free and 3 as eliminated.

Nothing is pruned here although c_{13} could have been actively used to prune D_5. To get a better pruning, we eliminate a variable as soon as possible as follows.

1. $c_{12} = \{(a,a),(b,b),(c,c)\}$. Revise D_1 with respect to c_{21}.
2. $c_{34} = \{(a,a),(b,c),(c,b)\}$. Revise D_3 with respect to c_{43}.
3. $c_{13} = \{(a,c),(b,b),(b,a),(c,c)\}$. Ignore it.
4. Some other constraints on 5.
5. $c_{15} = \{(a,a),(b,b),(c,c)\}$. Eliminate 1 immediately. As a consequence two new constraints are added. The first is $c'_{52} = \{(a,a),(b,b),(c,c)\}$, the composition of c_{51} and c_{12}. The second is $c'_{53} = \{(a,c),(b,b),(b,a),(c,c)\}$ (the composition of c_{51} and c_{13}). Revise D_5 with respect to the two new constraints. Discard c_{12} and c_{13}.
6. $c_{53} = \{(a,a),(b,b),(c,c)\}$. Eliminate 3. Add $c'_{54} = \{(a,a),(b,b),(c,c)\}$ (the composition of c'_{53} and c_{34}) and $c'_{55} = \{(a,c),(b,b),(b,a),(c,c)\}$ (the composition of c'_{53} and c_{35}). D_5 is revised to be $\{b,c\}$ (wrt c'_{55}). Discard c'_{53} (non-functional) and c'_{55}. Now the final network has constraints $\{c_{51}, c'_{52}, c_{53}, c'_{54}\}$,

revised domains and is satisfiable. Note. Here we use only compositions of constraints. An alternative way is to intersect c'_{53} and c_{53} first and then to eliminate 3. □

In order to make an active use of a non-functional constraint added earlier into the network, one of its variables is eliminated as soon as a functional constraint on it is added. This contrasts sharply with the strategy used in Pure-Eliminate. When a functional constraint is added, the variable with fewer constraints incident to it will be eliminated to decrease the cost of elimination. Let $nc[i]$ denote the number of constraints incident to i; $p[i] = j$ and $p[j] = j$ if i is eliminated with respect to c_{ij}, otherwise $p[i] = i$ and $p[j] = i$. As before, when i is eliminated wrt c_{ij}, we can safely say j is the *free* variable of i, or j has one eliminated variable i. At time 0, for all $i \in N$, $nc[i] = 0$ and $p[i] = 0$ where $0 \notin N$.

The algorithm Mixed-Eliminate given in Fig 6.7 works as follows. Assume a constraint c_{ij} is added. If any one or both i and j are eliminated, c_{ij} needs to be expressed with respect to correct free variables k and/or l (line 1-3), by eliminating i (wrt c_{ki}) and/or l (wrt c_{lj}) in line 4. Furthermore, if c_{ij} is functional, one of k and l has to be eliminated wrt the newly formed c_{kl}. The for loop in line 7 is to eliminate the variable l. In line 8, the new c_{km} is not checked whether it is a functional constraint. However, one may check it and initiate more elimination operations if necessary to reach a minimal (sub)network of more constraints. The parameter (N, D, C) of the algorithm is either initially empty or a previous output of the algorithm.

Theorem 11 *Given that at time t, a total of e constraints have been added into a mixed network with n variables. Algorithm Mixed-Eliminate makes any functional block in the network minimal in a worst case time complexity of $\mathcal{O}(ed^2 \log e)$.*

Proof. Let C be the set of all constraints added into the network until time t, N the set of variables and D the set of domains. It can be shown that each

```
procedure Mixed-Eliminate (inout (N, D, C), in c_ij, out consistent)
{  consistent ← true;
1. k ← i; l ← j;
2. if (p[i] ≠ 0) then  k ← p[i];
3. if (p[j] ≠ 0) then  l ← p[j];
4. c_kl ← c_kl ∩ (c_jl ∘ (c_ij ∘ c_ki));
5. if c_ij is functional then  Eliminate((N, D, C), c_kl, consistent);
   else { nc[k] ← nc[k] + 1; nc[l] ← nc[l] + 1; }
}

procedure Eliminate (inout (N, D, C), in c_ij, out consistent)
{ if (nc[i] > nc[j])  then  { k ← i; l ← j }
   else { k ← j; l ← i }
   C ← C ∪ {c_kl};
   Revise D_k wrt c_kl;
   if (D_k = ∅)  then  {
      consistent ← false;
      return;
   }
   if (k = l)  then  return;
7. for all c_lm ∈ C do {
8.    c_km ← (c_lm ∘ c_kl) ∩ c_km;
      C ← (C − {c_lm}) ∪ {c_km};
      p[m] ← k;
   }
   p[l] ← k;
   nc[k] ← nc[k] + nc[l];
   nc[l] ← 1;
}
```

Figure 6.7: Incremental elimination algorithm for mixed constraints

functional block (N^r, D^r, C^r) in (N, D, C) is eventually transformed into a canonical form $(N^r, D^{r'}, C^{r'})$ by Mixed-Eliminate. Specifically, in the canonical form, there is only one free variable r from N^r and there is one and only one functional constraint between the free variable and any other variable in N^r. Any non-explicit functional constraint $c_{ij} \in C$ between variables in the same functional block will be transformed to a constraint c_{rr} on r or a functional constraint between r and i (or j) $\in N^r$ because at last one of i and j is eliminated. Clearly, because of the continuous revision of D_r, any value in D_r can be extended to a solution of the functional block. So, each functional block is minimal.

The cost of all operations in the algorithm is bounded by $\mathcal{O}(d^2)$. For example, the composition of a functional constraint and a non-functional constraint has a cost of $\mathcal{O}(d^2)$. The complexity of the algorithm is determined by the number of times, from time 0 to time t, a constraint is processed (say c_{lm}) in the **for** loop (line 8) in Eliminate. For any constraint $c_{lm} \in C$, each time it is processed, the number of constraints under k (line 8) is at least twice the nc counter of l (Note the next appearance of c_{lm} in line 8 will be under the name of c_{km} when k is eliminated). Given e constraints in total, c_{lm} will be processed at most $\log e$ times. Hence the complexity of the algorithm is $\mathcal{O}(ed^2 \log e)$. □

In practice, arc consistency enforcing is widely used as a pruning facility in a constraint solver in CP systems. The above theorem shows that Mixed-Eliminate may be used in a practical solver since its cost is comparable to that of the optimal arc consistency enforcing algorithms (see Chapter 3).

6.4 Discussion

As discussed in the previous chapter, there is a lot of work related to functional constraints. However, little work takes the incrementality into consideration.

Two algorithms, Pure-Eliminate and Mixed-Eliminate, have been proposed to

solve functional constraints in an incremental system. They are especially useful for CP systems [JM94]. When applied to a CP system with mixed constraints, the first algorithm is more efficient while the second may achieve more pruning than the first. The choice between the two algorithms in a CP system will depend on the trade-off between efficiency of the consistency algorithm and its pruning ability.

In summary, our results are both significant and promising because:

- such functional constraints are an important class which occurs in many problems, and is a primitive for most CP systems.
- minimality is achieved on the functional block.
- the time complexities are low and almost the same as corresponding optimal AC algorithms.
- they are applicable to other constraint domains like Gaussian elimination and unification.

Part IV

Set Intersection and Consistency

As shown in Chapter 5, there is a canonical form for functional constraints. From the canonical form, it is straightforward to test the satisfiability and find a solution of the original network if necessary. A similar effort in CSP community is to identify, under certain conditions, "forms" of a network which imply the satisfiability of the network. One of the "forms" discovered by researchers is local consistency (k-consistency).

A major progress in the community of CSP is that local consistency in a network with particular properties is sufficient to guarantee global consistency. In this part, we present a framework to study the relationship between the local consistency and global consistency of a network. Under this framework, results on *set intersection problem* can be lifted to results on the consistency of a network. The set intersection problem is: under what condition the intersection of a class of sets is not empty.

The framework unifies several well-known consistency results including van Beek and Dechter's work on row convex, m-tight, and m-loose constraints. Thanks to the framework, some new results on local consistency and global consistency are also discovered. They improve our understanding of *convex* and *tight* constraints.

Chapter 7

Set Intersection and Consistency

A pure backtracking search procedure is not efficient for most CSP problems. From the work by Waltz [Wal72, Mon74, Mac77a], it is observed that arc consistency enforcing can significantly improve the efficiency of a search procedure by pruning the search space. A natural question arises: to what extent can we prune the search space so that no backtracking is needed? The introduction of the concept of *k-consistency* [Fre78](see Section 2.3) is an attempt to answer this question. A problem can then be solved without backtracking by enforcing strong n-consistency, where n is the number of all variables. Although this operation is too expensive to be of any practical use, it plays a role in understanding the solving of some classes of CSP problems with particular properties.

An interesting progress is that certain properties of constraint networks are identified such that, a certain level of local consistency is sufficient to guarantee global consistency, that is strongly n-consistency, on these networks.

This progress is significant in terms of the following considerations. As observed by Dechter [Dec92b], in resource-bounded reasoning, at any inference step it is desirable to examine only a few data items and to avoid decision where to store intermediate results. This kind of locality is involved in all realistic models of human reasoning. In constraint based reasoning, this principle of locality has well

been distilled into local consistencies. There are natural relationships between local consistency and global consistency in real life reasoning tasks. For example, the scene labeling scheme [Wal75] (essentially an arc consistency enforcing algorithm) of Waltz often leads to globally consistent objects. Secondly, local consistency is obviously more efficient to compute.

The existing work has mainly focused on two classes of properties of a constraint network: topological properties of the associated graph of the network and semantic properties of constraints. One example of the first class is that if a constraint network forms a tree, arc consistency is sufficient to make the network minimal in the sense defined in Section 5.1. Freuder identifies a parameter *width* of a graph [Fre82, Fre85]. Given the width of the graph of a network, a certain level of consistency in the network is sufficient to ensure global consistency. Dechter and Pearl generalizes the results on trees to hyper-trees [DP89].

An example of the second class is that path consistency in a network, where the domain of each variable has two or less values, ensures global consistency [vBD95, page 550]. In fact, there are many results in this class. Montanari shows that for *monotone* constraints, path consistency implies global consistency [Mon74]. Dechter discovers that a certain level of consistency in a network whose domains are of limited size ensures global consistency [Dec92b]. Van Beek and Dechter generalizes the monotone constraints to a much larger class of *row convex constraints* [vBD95]. Later, they make use of the looseness and tightness of a constraint to study the consistency inside a network [vBD97]. Another line of work starting from Schaefer's [Sch78] work on boolean satisfiability to the work of Jeavons et. al. [JCG97] on closure properties on constraints also falls into this class.

We observe that k-consistency is closely related to set intersection results. Here, we propose a framework[1] relating consistency in a network to set intersection re-

[1]Our work can be related to the first class in those cases where the network topology leads to some set intersection property.

sults. It unifies a number of well known results on semantic properties of constraints, for example those in [vBD95, vBD97]. Our framework allows the study of properties of consistency on a particular network from the perspective of properties of set intersection. For example, we have the following property of set intersection. For a collection of *convex* sets, if every two (2) of them intersect, then the intersection of all the sets is not empty. The interesting point is that local information on intersection of every pair of sets gives global information on intersection of all sets. Intuitively, this can be related to getting global consistency from local consistency. In fact, by dint of convex sets, a special class of constraints—*row convex* constraints—is identified. Our framework enables us to lift the result on convex sets to this result. If a binary network of row convex constraints is (2+1)-consistent (path consistency), it is globally consistent. This example will be elaborated in Section 7.3.

In this chapter, we first present the properties on intersection of *tree convex* sets, *small* sets and *large* sets in Section 7.1. The framework is developed in Section 7.2. It consists of a lifting lemma and a proof schema. The schema provides a generic way of using the lifting lemma to obtain consistency results from properties of set intersection.

We then demonstrate several applications of the framework. Specifically, a class of tree convex constraints is identified and the property of consistency on a network of such constraints is presented in Section 7.3. It generalizes the existing work on row convex constraints [vBD95]. A new result on the tightness of constraints is presented in Section 7.4. It advances the existing work [vBD97] in the aspect that a certain level of local consistency still ensures global consistency on a network where only some constraints are tight. Finally, tree convexity and tightness are studied under relational consistency and directional consistency in Section 7.5. An important result is that networks with certain constraints satisfying tightness restrictions can be made globally consistent by enforcing some level of relational

consistency.

This chapter is summarized in the last section.

7.1 Properties of Set Intersection

The set intersection property which we are concerned with is:

> Given a collection of l finite sets, under what conditions is the intersection of all l sets not empty?

We use $\mathcal{S}$ to denote a collection of l sets: $\{E_1, E_2, \cdots, E_l\}$, and U the union of all the sets in $\mathcal{S}$, that is $U = \bigcup_{i \in 1..l} E_i$.

A well known example is for mutually intersecting real intervals there is a real number common to all intervals.

For a collection of arbitrary sets, we may not have any answer for the intersection problem. In this section, we study two special types of sets. The first type is restricted to be "convex" in some sense, and the second type is restricted by the size of sets involved.

7.1.1 Sets with Convexity Restrictions

Definition 25 *Given a set U and a total ordering "$\preceq$" on it, a set $A \subset U$ is* convex *if the elements in it are consecutive under the ordering, that is*

$$A = \{v \in U \mid \min A \preceq v \preceq \max A\}.$$

Given $\mathcal{S}$ and U, the sets in $\mathcal{S}$ are convex *if there is a total ordering on U such that every set in $\mathcal{S}$ is convex under the ordering.*

A convex set E contains every element in U between its least ($\min E$) and greatest ($\max E$) elements under the ordering $\preceq$.

Note that when we say a set E is convex, we need a reference set U and an ordering on it. So, any finite set is convex with respect to itself, under any total ordering. For any two sets E_i and E_j, there exists an ordering such that they are convex under the same ordering with respect to $E_i \cup E_j$. The ordering may be defined as follows. We adopt any ordering for the elements in $E_i - E_j$. We do so for those in $E_i \cap E_j$ and in $E_j - E_i$ respectively. Finally, we let all elements in $E_i - E_j$ be smaller than those in $E_i \cap E_j$ which in turn are smaller than those in $E_j - E_i$. For example, $E_1 = \{1, 2\}$ and $E_2 = \{0, 2, 3\}$ are convex with respect to $U = \{0, 1, 2, 3\}$ under ordering $1 \preceq 2 \preceq 0 \preceq 3$. E_1 consists of the first two elements and E_2 the last three elements.

As an example of a collection of more than two sets, $\{1, 9\}, \{3, 9\}$, and $\{5, 9\}$ are not convex. It can be verified by exhausting all possible orderings on the set $\{1, 3, 5, 9\}$.

So, an interesting question is whether a collection $\mathcal{S}$ of more than two sets is convex with respect to U.

Proposition 7 *The convexity of a collection of sets $\mathcal{S}$=$\{E_1, E_2, \cdots, E_l\}$ can be tested in*

$$\mathcal{O}(l + |U| + \sum_{i \in 1..l} |E_i|).$$

Proof. Construct a matrix A in the following way. $A[i, j] = 1$ where $i \in 1..l$ and $j \in U$ if and only if the set E_i contains the element j. $\mathcal{S}$ is convex if and only if A is *row convex* (whose definition is in Section 5.3). In terms of the theorem by Booth and Lueker [vBD95, page 551], the proposition follows. □

The following result on the intersection of convex sets is a variation of van Beek and Dechter's lemma 3.1 in [vBD95].

Lemma 3 (Convex Sets Intersection) *Assume the sets in $\mathcal{S}$ are convex under*

a common total ordering on U.

$$\bigcap_{i \in 1..l} E_i \neq \emptyset$$

if and only if for every $E_i, E_j \in \mathcal{S}$,

$$E_i \bigcap E_j \neq \emptyset.$$

Proof. The necessary condition for the intersection of all the sets to be non-empty is immediate. Next we prove the sufficient condition. Now we try to find an $a \in U$ such that $a \in E_1, a \in E_2, \cdots, a \in E_l$. Since sets in $\mathcal{S}$ are convex, a must satisfy

$$\begin{cases} \min E_1 \preceq a \preceq \max E_1, \\ \min E_2 \preceq a \preceq \max E_2, \\ \qquad\qquad \vdots \\ \min E_l \preceq a \preceq \max E_l. \end{cases}$$

In other words, a is no smaller than the greatest of the elements in the left column to a, and no greater than the least of the elements in the right column to a:

$$\max\{\min E_1, \min E_2, \cdots, \min E_l\} \preceq a \preceq \min\{\max E_1, \max E_2, \cdots, \max E_l\}. \quad (7.1)$$

For any set $E_i \in \mathcal{S}$, since it intersects every other set in $\mathcal{S}$, the least element in E_i is no greater than the greatest element of any set in $\mathcal{S}$ (including itself) :

$$\min E_i \preceq \min\{\max E_1, \max E_2, \cdots, \max E_l\}.$$

Hence,

$$\max\{\min E_1, \min E_2, \cdots, \min E_l\} \preceq \min\{\max E_1, \max E_2, \cdots, \max E_l\}.$$

It implies that there exists an a satisfying constraint 7.1 and thus

$$\bigcap_{i \in 1..l} E_i \neq \emptyset$$

□

The convexity of a collection of sets imposes a strong restriction on the relationship among the sets of concern in the sense that all sets are dense under a common total ordering. The Hasse diagram of a total ordering is a chain. Recall that Hasse diagram is a graph for a total ordering (relation) by removing edges which can be obtained by transitivity and reflexivity. We now generalize the chain to a tree.

Definition 26 *Given a set U and a tree $\mathcal{T}$ with vertices U. A set $A \subseteq U$ is* tree convex *if and only if there exists a subtree of $\mathcal{T}$ whose set of vertices is exactly A. A set of $\mathcal{S}$ is* tree convex *if there is a tree on U under which every set in $\mathcal{S}$ is tree convex.*

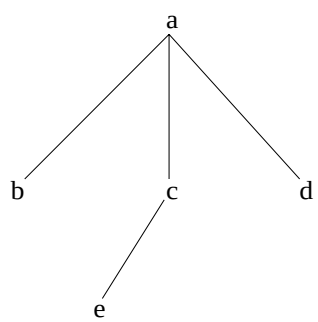


Figure 7.1: A tree with nodes $\{a, b, c, d, e\}$

Example. Consider a set $U = \{a, b, c, d, e\}$ and a tree in Fig 7.1. The subset $\{a, b, c, d\}$ is tree convex. So is the set $\{b, a, c, e\}$ since the elements in the set consists of a subtree. However, $\{b, c, e\}$ is not tree convex for it does not form a subtree of the given tree.

Consider again the example of $\{1, 9\}, \{3, 9\}$, and $\{5, 9\}$ again. The three sets are not convex but tree convex. A tree on $\{1, 3, 5, 9\}$ can be constructed with 9

being the root and $1, 3, 5$ being its children. Each set covers the nodes of one branch of the tree. Therefore they are tree convex. $\square$

The tree convexity describes certain relationship among the sets of concern. For the tree convex sets, we have the following property on intersection.

Lemma 4 (Tree convex Sets Intersection) *Assume the sets in $\mathcal{S}$ are tree convex.*

$$\bigcap_{i \in 1..l} E_i \neq \emptyset$$

if and only if for every $E_i, E_j \in \mathcal{S}$, $E_i \bigcap E_j \neq \emptyset$.

Proof. Let $\mathcal{T}$ be a tree such that there exists a subtree T_i for each $E_i \in \mathcal{S}$. We take $\mathcal{T}$ as a rooted tree and thus every T_i ($i \in 1..l$) can be regarded as a rooted tree whose root is exactly the node nearest to the root of $\mathcal{T}$. Let r_i denote the root of T_i for $i \in 1..l$.

To prove

$$\bigcap_{i \in 1..l} E_i \neq \emptyset,$$

we want to show the intersection of the trees $\{T_i \mid i \in 1..l\}$ is not empty. The following propositions on subtrees are necessary in our main proof.

Proposition 8 *Let T_1, T_2 be two subtrees of a tree $\mathcal{T}$, and $T = T_1 \cap T_2$. T is a tree.*

If $T = \emptyset$, it is a trivial tree. Now let $T \neq \emptyset$. Since T is a portion of T_1, there is no circuit in it. It is only necessary to prove T is connected. That is to show, for any two nodes $u, v \in T$, there is a path between them. $u, v \in T_1$ and $u, v \in T_2$ respectively imply that there exist paths $P_1 : u, \cdots, v$ in T_1 and $P_2 : u, \cdots, v$ in T_2 respectively. Recall that *there is a unique path from u to v in $\mathcal{T}$* and that T_1 and T_2 are subtrees of $\mathcal{T}$. Therefore, P_1 and P_2 cover the same nodes and edges, and they are in T, the intersection of T_1 and T_2. P_1 is the path we want.

Proposition 9 *Let T_1, T_2 be two subtrees of a tree $\mathcal{T}$, and $T = T_1 \cap T_2$. T is not empty if and only if at least one of the roots of T_1 and T_2 is in T.*

Let r_1 and r_2 be the roots of T_1 and T_2 respectively. If $r_1 \in T$, the proposition is correct. Otherwise, we show $r_2 \in T$. Assume the contrary $r_2 \notin T$. Let r be the root of $\mathcal{T}$ and v the root of T (T is a tree in terms of Proposition 8). We have paths $P_1 : r_1, \cdots, v$ in T_1; $P_2 : r_2, \cdots, v$ in T_2; and $P_3 : r, \cdots, r_1$, and $P_4 : r, \cdots, r_2$ in $\mathcal{T}$. The assumption tells that $r_1 \neq r_2$. From the closed walk $P_3 P_1 P_2' P_4'$ where P_2' and P_4' are the reverse of P_2 and P_4 respectively, we can construct a circuit containing at least r_1 and r_2. It contradicts that there is no circuit in $\mathcal{T}$.

Further we have the following observation.

Proposition 10 *Let the root of $\mathcal{T}$ be r. Given two subtrees T_1 and T_2 of $\mathcal{T}$ with roots r_1 and r_2 respectively. Let r_1 be not closer to r than r_2, and T the intersection of T_1 and T_2. r_1 is the root of T if T is not empty.*

Let r_1 be farther to the r than r_2. Assume r_2 is the root of T. Since r_1 is farther to r than r_2, r_2 is not possible to be a node of T_1. It contradicts that $r_2 \in T$.

Let $T = \bigcap_{i \in 1..l} T_i$. We are ready now to prove our main result $T \neq \emptyset$. We select a tree $T_{\max}$ from $T_1, T_2, \cdots, T_l$ such that its root $r_{\max}$ is the farthest away from r of $\mathcal{T}$ among the roots of the concerned trees. In terms of Proposition 10, that $T_{\max}$ intersect with every other trees implies that $r_{\max}$ is a node of every T_i ($i \in 1..l$). Therefore, $r_{\max} \in T$. $\square$

Recall that a partial order can be represented by an acyclic directed graph, or Hasse diagram. It is tempting to further generalize the tree convexity to partial convexity in the following way.

Definition 27 *Given a set U and a partial order on it. A set $A \subset U$ is* partially convex *if and only if A is the set of nodes of a connected subgraph of the partial order. Given $\mathcal{S}$ and U, the sets in $\mathcal{S}$ are* partially convex *if there is a partial ordering on U such that every set in $\mathcal{S}$ is convex under the ordering.*

However, for this generalization, we do not have a result similar to Lemma 4. See the following example. Each of the three sets $\{c, b, d\}$, $\{d, f, a\}$ and $\{a, e, c\}$

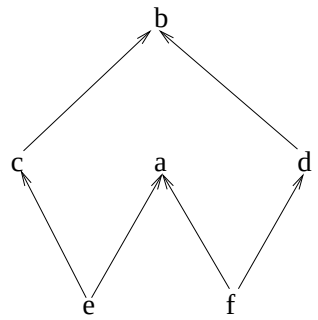


Figure 7.2: A partial order with nodes $\{a, b, c, d, e\}$

consists of the nodes of some subgraph in Fig 7.2. They are partially convex and intersect pairwise, but the intersection of the three sets is empty.

7.1.2 Sets with Cardinality Restrictions

Motivated by the observation in [vBD97, lemma 3.2 in page 556], we have the following result on arbitrary sets where the only restriction is that each set has a bounded number of elements. The name of the following lemma is after that restriction.

Lemma 5 (Small Sets Intersection) *Let $\mathcal{S}$=$\{E_1, E_2, \cdots, E_l\}$ be a collection of sets. For any $E_i \in \mathcal{S}$, assume E_i is finite and $|E_i| \leq m$ $(< l)$.*

$$\bigcap_{i \in 1..l} E_i \neq \emptyset$$

if and only if the intersection of any $m + 1$ sets from $\mathcal{S}$ is not empty.

Proof.

The necessary condition is immediate.

The sufficient condition is proven by induction on l, the number of sets in $\mathcal{S}$. The base case is $l = m + 1$ and the lemma is trivially true. Assuming that the

lemma is true when $l = k$ $(> m)$, we show that it is also true when $l = k + 1$.

Without loss of generality, the subscripts of the $k + 1$ sets are numbered from 1 to $k + 1$. Let A_i be the intersection of all k sets in S$-\{E_i\}$:

$$A_i = E_1 \cap \cdots \cap E_{i-1} \cap E_{i+1} \cap \cdots \cap E_{k+1}, \ for \ 1 < i \leq k + 1.$$

In accordance with the inductive hypothesis, the intersection of every k sets in $\mathcal{S}$ is not empty since the intersection of every $m + 1$ sets from $\mathcal{S}$ is not empty. Hence, $|A_i| \geq 1$.

If $A_i \cap A_j \neq \emptyset$ for some $i, j \in 2..k + 1, i \neq j$,

$$\bigcap_{i \in 1..k+1} E_i = A_i \cap A_j \neq \emptyset.$$

Assume the contrary that $A_i \cap A_j = \emptyset$ for all distinct i and j. According to the construction of A_i's,

$$E_1 \supseteq \bigcup_{i \in 2..k+1} A_i.$$

Therefore,

$$|E_1| \geq \sum_{i \in 2..k+1} |A_i| \geq k > m$$

which contradicts $|E_1| \leq m$. $\square$

Motivated by [vBD97, lemma 4.1 in page 561], we consider the following restrictions on a collection of sets: 1) each set is with size larger than some number m; but 2) there is a small number of sets in the collection, and 3) the union U of all sets has limited size d. The name of the large sets intersection lemma is after the first restriction. In this case, if the intersection of all sets is empty, then for any $a \in U$, a is excluded by some set E_i. However, since E_i is large, it can exclude at most $d - m$ elements in U. All sets in $\mathcal{S}$ can exclude at most $l \times (d - m)$ elements in U. For l is also small (such that $l(d - m) < d$), some element in U may not be

excluded by any set, which means that the intersection of all sets is not empty.

Lemma 6 (Large Sets Intersection) *For all $E_i \in \mathcal{S}$, assume E_i is finite and $|E_i| \geq m$. Let $|\bigcup_{i\in 1..l} E_i| = d$. If $l \leq \lceil d/(d-m)\rceil - 1$, then*

$$\bigcap_{i\in 1..l} E_i \neq \emptyset.$$

Proof. Let $U = \bigcup_{i\in 1..l} E_i$, and $A_i = U - E_i$ for all $i \leq l$. It is immediate that

$$\bigcup_{i\in 1..l} A_i \subseteq U.$$

We know

$$|\bigcup_{i\in 1..l} A_i| \leq \sum_{i\in 1..l} |A_i|.$$

For $|A_i| \leq d - m$, we have

$$\sum_{i\in 1..l} |A_i| \leq \sum_{i\in 1..l} (d-m) = l(d-m) < d.$$

Hence, $\bigcup_{i\in 1..l} A_i$ is a proper subset of U. There exists $x \in U$ such that $x \notin A_i$ for all $i \leq l$, which implies that $x \in E_i$ for all $i \leq l$. □

The small sets intersection Lemma 5 can be generalized to the following one.

Lemma 7 (Small Set Intersection) *Given a collection of sets $\mathcal{S}$. Assume there is a set $E \in \mathcal{S}$ such that $|E| \leq m < l$.*

$$\bigcap_{i\in 1..l} E_i \neq \emptyset$$

if and only if the intersection of any $m+1$ sets (from $\mathcal{S}$) is not empty.

Proof. When re-examining the proof of Lemma 5, it is easy to find that we only make use of the cardinality of E_1. If we substitute E for E_1 in that proof, it will be a proof for this lemma. □

A special case of this lemma is a set with only *one* element.

Corollary 2 (Singleton Set Intersection) *Given a collection of sets $\mathcal{S}$. Assume there is a set $E \in \mathcal{S}$ such that $|E| = 1$.*

$$\bigcap_{i \in 1..l} E_i \neq \emptyset$$

if and only if all sets mutually intersect.

This result is straightforward. Since E has only one element and its intersection with any other set is not empty, the element in E is the one shared by all sets.

7.2 Set Intersection and Consistency

In this section we relate set intersection and k-consistency in constraint networks. Furthermore, a proof schema is proposed to lift properties on set intersection to properties on consistencies in a particular network.

Underlying the concept of consistency is whether an instantiation of some variables can be extended to a new variable such that all relevant constraints to the new variable are satisfied. A *relevant* constraint to a variable x is a constraint where only x is uninstantiated and all others are instantiated. Each relevant constraint allows a set (possibly empty) of values for the new variable. This set is called the *extension set* below. The satisfiability of all relevant constraints depends on whether the intersection of their extension sets is non-empty (see Lemma 8).

Definition 28 *Given a constraint c_{S_i}, a variable $x \in S_i$ and any instantiation $\bar{a}$ of $S_i - \{x\}$, the* extension set *of $\bar{a}$ to x with respect to c_{S_i} is defined as*

$$E_{i,x}(\bar{a}) = \{b \in D_x \mid (\bar{a}, b) \ satisfies \ c_{S_i}\}.$$

An extension set is trivial *if it is empty; otherwise it is* non-trivial.

Throughout this chapter, it is usually the case that an instantiation $\bar{a}$ of $Y - \{x\}$ is given and $Y - \{x\}$ is a superset of $S_i - \{x\}$. Let $\bar{b}$ be the instantiation obtained by restricting $\bar{a}$ to the variables only in $S_i - \{x\}$. For ease of presentation, we still use $E_{i,x}(\bar{a})$, instead of $E_{i,x}(\bar{b})$, to denote the extension of $\bar{b}$ to x under constraint c_{S_i}. Some of the three parameters i, $\bar{a}$ and x may be omitted from an expression hereafter whenever they are clear from the context.

Example. Consider the network with variables $\{x, x_1, x_2, x_4, x_5\}$:

$$\begin{aligned}
c_{S_1} &= \{(a,b,d),(a,b,a)\}, \quad S_1 = \{x_1, x_2, x\}; \\
c_{S_2} &= \{(b,a,d),(b,a,b)\}, \quad S_2 = \{x_2, x_4, x\}; \\
c_{S_3} &= \{(b,d),(b,c)\}, \qquad\quad S_3 = \{x_2, x\}; \\
c_{S_4} &= \{(b,a,d),(b,a,a)\}, \quad S_4 = \{x_2, x_5, x\}; \\
D_1 = D_4 = D_5 &= \{a\}, D_2 = \{b\}, D_x = \{a,b,c,d\}.
\end{aligned}$$

Let $\bar{a} = (a, b, a)$ be an instantiation of variables $Y = \{x_1, x_2, x_4\}$. The relevant constraints to x are c_{S_1}, c_{S_2}, and c_{S_3}. c_{S_4} is not relevant since it has two uninstantiated variables. The extension sets of $\bar{a}$ to x with respect to the relevant constraints are:

$$E_1(\bar{a}) = \{d, a\}, E_2(\bar{a}) = \{d, b\}, E_3(\bar{a}) = \{d, c\}.$$

The intersection of the extension sets above is not empty, implying that $\bar{a}$ can be extended to satisfy all relevant constraints c_{S_1}, c_{S_2} and c_{S_3}.

Let $\bar{a} = (b, b)$ be an instantiation of $\{x_2, x\}$. $E_{1,x_1}(\bar{a}) = \emptyset$ and it is trivial. In other words, when an instantiation has a trivial extension set, it can not be extended to satisfy the constraint of concern. □

The relationship between *k-consistency* and set intersection is characterized by the following lemma which is a direct consequence of the definition of *k-consistency.*

Lemma 8 (Set Intersection and Consistency; lifting) *A constraint network $\mathcal{R}$ is k-consistent if and only if for any consistent instantiation $\bar{a}$ of any $(k-1)$*

distinct variables $Y = \{x_1, x_2, \cdots, x_{k-1}\}$*, and any new variable* x_k*,*

$$\bigcap_{c_S \in C_Y} E_S(\bar{a}) \neq \emptyset$$

where C_Y *is the set of all relevant constraints to* x_k*.*

The proof is straightforward and omitted. The insight behind this lemma is a view of consistency from the perspective of set intersection.

Example. Consider the example above. We would like to check whether the network is 4-consistent. Consider the instantiation $\bar{a}$ of Y again. This is a consistent instantiation for there is no direct constraint among the variables in Y. To extend it to x, we need to check the first three constraints. The extension is feasible because the intersection of E_1, E_2, and E_3 is not empty. Similarly, by exhausting all consistent instantiations of any three variables, we know the network is 4-consistent. Conversely, if we know the network is 4-consistent, immediately we can say the intersection of the three extension sets of $\bar{a}$ to x is not empty. □

In a word, with this lemma, consistency information can be obtained from the intersection of extension sets, and vice versa. Using this view of consistency as set intersection, some results on set intersection properties, including all those in Section 7.1, can be *lifted* to get various consistency results for a constraint network through the following *proof schema*.

Proof Schema

1. (*Consistency to Set*) From a certain level of consistency *in* the constraint network, we derive information on the intersection of certain extension sets according to Lemma 8.

2. (*Set to Set*) From the *local* intersection information of the extension sets, information may be obtained on intersection of more extension sets according to set intersection properties (for example the lemmas given in Section 7.1).

3. (*Set to Consistency*) From the new information on the intersection of the extension sets, higher level of consistency is obtained according to Lemma 8.

4. (*Formulate conclusion on the consistency of the constraint network*). □

Given the proof schema, Lemma 8 is also called the *lifting* lemma.

In the following sections, we demonstrate how the set intersection properties and the proof schema are employed to obtain both new and well known results on consistency of a network.

7.3 Application I: Global Consistency on Tree Convex Constraints

The notion of *extension set* plays the role of a bridge between the restrictions to set(s) and properties of special constraints. The sets in Lemma 3 are restricted to be convex. If all extension sets of a constraint are convex, the constraint is *row convex*.

Definition 29 *A constraint c_S is* row convex *with respect to x if and only if the sets in*

$$A = \{E_{S,x} \mid E_{S,x} \text{ is a non-trivial extension of some instantiation of } S - \{x\}\}$$

are convex. It is row convex if under a common total ordering on the union of involved domains, it is row convex with respect to every $x \in S$.

Example. Consider the constraint $\{(a,b),(a,c),(b,a),(b,b),(c,b)\}$ on x_1 and x_2 with $D_1 = D_2 = \{a,b,c\}$. Its matrix representation is:

$$\begin{array}{cc} & x_2 \\ & \begin{array}{ccc} a & b & c \end{array} \\ \begin{array}{cc} a \\ x_1\ b \\ c \end{array} & \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

Assume a total ordering $a \preceq b \preceq c$ on the domains. By instantiating x_1, we get extension sets to x_2:

$$E(a) = \{b,c\}, E(b) = \{a,b\}, E(c) = \{b\}.$$

They are convex because $E(a)$ contains all elements from b to c, $E(b)$ from a to b, and $E(c)$ from b to b. In fact, it is clearer to see that from the matrix. When $x_1 = a$, 1's in the first row of the matrix are consecutive. So are the 1's in the second row and third row respectively when $x_1 = b$ and $x_1 = c$ respectively. This is exactly why this kind of constraint is named as *row convex*. Similarly, extension sets of any instantiation of x_2 are also convex. Therefore, the constraint is row convex. However the following constraint

$$\begin{array}{cc} & x_2 \\ & \begin{array}{ccc} a & b & c \end{array} \\ \begin{array}{cc} a \\ x_1\ b \\ c \end{array} & \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \end{array}$$

is not row convex, since no matter what ordering on the domain is used, there always exists some row whose 1's are not consecutive. In other words, there always exists an extension set which missed some value between its least and greatest. For example, under the ordering (a, b, c), $E_{x_2}(c) = \{a, c\}$ where b is missed. □

Definition 30 *A constraint network is* row convex *if and only if all constraints are row convex under a common total ordering on the union of all domains.*

The tree convex set naturally introduces the following special constraint.

Definition 31 *A constraint c_S is* tree convex *with respect to x if and only if the sets in*

$$A = \{E_{S,x} \mid E_{S,x} \text{ is a non-trivial extension set of some instantiation of } S - \{x\}\}$$

are tree convex. c_S is tree convex *iff under a common tree on the union of involved domains, it is tree convex with respect to every $x \in S$.*

Definition 32 *A constraint network is* tree convex *if and only if there exists a tree on the union of all domains in the network such that every constraint is tree convex under the tree.*

The consistency results on those special networks can be derived from the property of set intersection by the proof schema. We now have the main result of this section.

Theorem 12 (Tree Convexity) *Let $\mathcal{R}$ be a tree convex network of constraints with arity at most r. $\mathcal{R}$ is globally consistent if it is strongly $2(r-1)+1$ consistent.*

Proof. The network is strongly $2(r-1)+1$ consistent by assumption. We prove by induction that the network is k consistent for any $k \in \{2r, \cdots, n\}$.

Consider any instantiation $\bar{a}$ of any $k-1$ variables and any new variable x. Let the number of relevant constraints be l. For each relevant constraint there is one

extension set of $\bar{a}$ to x. So we have l extension sets. If the intersection of all l sets is not empty, we have a value for x such that the extended instantiation satisfies all relevant constraints.

(*Consistency to Set*) Consider any two of the l extension sets: E_1 and E_2. The two corresponding constraints involve at most $2(r-1)+1$ variables since the arity of a constraint is at most r and each of the two constraints has x as a variable. According to the consistency lemma, that $\mathcal{R}$ is $2(r-1)+1$-consistent implies that the intersection of E_1 and E_2 is not empty.

(*Set to Set*) Since all relevant constraints are tree convex under the given tree, the extension sets of $\bar{a}$ to x are tree convex. Hence, the fact that every two of the extension sets intersect shows that the intersection of all l extension sets is not empty, in terms of the *tree convex sets intersection* lemma.

(*Set to Consistency*) In terms of consistency lemma, $\mathcal{R}$ is k-consistent. □

Since a row convex constraint is tree convex, we have the following result on row convex constraint originally discovered by van Beek and Dechter [vBD95].

Corollary 3 (Row Convexity) [vBD95] *Let $\mathcal{R}$ be a row convex network of constraints with arity at most r. $\mathcal{R}$ is globally consistent if it is strongly $2(r-1)+1$ consistent.*

Alternatively, this result can be lifted from Lemma 3.

Theorem 7 in Chapter 5, a special case of the corollary above, deals with a network with binary constraints.

7.4 Application II: on Tightness and Looseness of Constraints

In this section, we study constraint networks in terms of the tightness of constraints. Specifically, the *m-tight* and *m-loose* properties [vBD97] of a constraint

are introduced to describe the tightness of a constraint.

7.4.1 Tightness of Constraints

The *m-tight* property of a constraint is related to the cardinality of extension set in the following way.

Definition 33 *A constraint c_{S_i} is* m-tight *with respect to $x \in S_i$ if and only if for any instantiation $\bar{a}$ of $S_i - \{x\}$,*

$$|E_{i,x}| \leq m \text{ or } |E_{i,x}| = |D_x|.$$

A constraint c_{S_i} is m-tight *if and only if it is m-tight with respect to every $x \in S_i$.*

For example, the constraint $x \leq y$, where $x \in \{1, 2, \cdots, 10\}$ and $y \in \{1, 2, \cdots, 10\}$, is 9-tight. Note here $|E_x(10)| > 9$ when $y = 10$. However, the constraint is still 9-tight since $|E_x(10)| = 10$, the size of the domain of x.

Definition 34 *A constraint network is* weakly *m-tight at level k if and only if for every set of variables $\{x_1, \cdots, x_l\}(k \leq l < n)$ and a new variable, there exists an m-tight constraint among the relevant constraints after an instantiation of the l variables.*

Now, Lemma 7 on small set intersection results in the following theorem.

Theorem 13 (Weak Tightness) *If a constraint network $\mathcal{R}$ with constraints of arity at most r is strongly $((m+1)(r-1)+1)$-consistent and weakly m-tight at level $((m+1)(r-1)+1)$, it is globally consistent.*

Proof. Let $j = (m+1)(r-1)+1$. The constraint network $\mathcal{R}$ will be shown to be k-consistent for all k $(j < k \leq n)$.

Let $Y = \{x_1, \cdots, x_{k-1}\}$ be a set of any $k-1$ variables, and $\bar{a}$ be an instantiation of all variables in Y. Consider any additional variable x_k. Without loss of

generality, let the relevant constraints be $c_{S_1}, \cdots, c_{S_l}$, and E_i be the extension set of $\bar{a}$ to x_k with respect to c_{S_i} for $i \leq l$.

(*Consistency to Set*) Consider any $m+1$ of the l extension sets. All the corresponding $m+1$ constraints contain at most $(m+1)(r-1)+1$ variables including x_k. Since $\mathcal{R}$ is $(m+1)(r-1)+1$-consistent, according to the *set intersection and consistency* lemma, the intersection of $m+1$ extension sets is not empty.

(*Set to Set*) The network is weakly m-tight at level $((m+1)(r-1)+1)$. So, there must be an m-tight constraint among the relevant constraints $c_{S_1}, \cdots, c_{S_l}$. Let it be c_{S_i}. We know its extension set $|E_i| \leq m$. For the intersection of every $m+1$ of the extension sets is not empty, all l extension sets share a common element in terms of the *small set intersection* lemma.

(*Set to Consistency*) From the lifting lemma, $\mathcal{R}$ is k-consistent. □

Immediately we have the following result which is a main result in [vBD97].

Corollary 4 (Tightness) [vBD97] *If a constraint network $\mathcal{R}$ with constraints that are m-tight and of arity at most r is strongly $((m+1)(r-1)+1)$-consistent, then it is globally consistent.*

Again this result can be lifted directly from Lemma 5.

It is interesting to observe that Corollary 4 requires every constraint to be m-tight while in the weak tightness theorem it may not be necessary for all constraints to be m-tight. To see the difference, consider the following example.

Example. In this example, we are interested in *how many* tight constraints are necessary to make a constraint network *weakly* m-tight. Therefore we omit the semantics of constraints and focus on the topological structure of the network. Now we construct a constraint network with variables $\{1, 2, 3, 4, 5\}$. In the network, there is a constraint between any pair of variables and among any three variables. Let the network be strongly 4-consistent. The network is shown in Table 7.1.

Binary Constraints on variables	Ternary Constraints on variables
12	123
13	124
14	125
15	134
23	135
24	145
25	234
34	235
35	245
45	345

Table 7.1: A network with complete binary and ternary constraints

Since the network is already strongly 4-consistent, we can simply ignore the instantiations of less than 4 variables. This is why we introduce the level at which the network is weakly m-tight. The interesting level here is 4. For each possibility of extending four instantiated variables to the other one, the relevant constraints are listed in Table 7.2. In the table an entry like $1234 \rightarrow 5$ stands for extending the instantiation of variables $\{1, 2, 3, 4\}$ to variable 5, and an entry like 125 stands for a constraint on variables $\{1, 2, 5\}$. To make the network weakly m-tight at level 4, one choice is to make constraints (suffixed by * in the Table 7.2) on $\{1, 2, 5\}$ and $\{1, 3, 4\}$ m-tight. Alternatively, it is sufficient for the constraints (suffixed by + in the table) on $\{1, 5\}$, $\{2, 3\}$ and $\{3, 4\}$ to be m-tight. However, the tightness corollary requires all binary and ternary constraints to be m-tight. The improvement of the weak m-tightness theorem is significant in this consideration. Further results can be found in the next section. □

7.4.2 Looseness of Constraint

The next result is a consequence of the large sets intersection lemma. For large sets, their intersection is not empty as long as they are large enough. It means that there is certain level of consistency in a constraint network characterized by

extension	relevant constraints									
1234 → 5	125*,	135 ,	145 ,	235,	245,	345,	15+,	25 ,	35 ,	45
2345 → 1	231 ,	241 ,	251*,	341,	351,	451,	21 ,	31 ,	41 ,	51+
3451 → 2	132 ,	142 ,	152*,	342,	352,	452,	12 ,	32+,	42 ,	52
4512 → 3	123 ,	143*,	153 ,	243,	253,	453,	13 ,	23+,	43 ,	53
5123 → 4	124 ,	134*,	154 ,	234,	254,	354,	14 ,	24 ,	34+,	54

Table 7.2: Relevant constraints in extending the instantiation of four variables to the other one

a large set. This is in contrast to the previous results where global consistency is implied by certain level of local consistency.

The *m-loose* property of a constraint is related to the cardinality of the extension set in the following way.

Definition 35 *A constraint c_{S_i} is* m-loose *with respect to $x \in S_i$ if and only if for any instantiation $\bar{a}$ of $S_i - \{x\}$,*

$$|E_i| \geq m.$$

A constraint c_{S_i} is m-loose *if and only if it is m-loose with respect to every $x \in S_i$.*

For example, the constraint $x \leq y$, where $x \in \{1, 2, \cdots, 10\}$ and $y \in \{1, 2, \cdots, 10\}$, is 1-loose.

The *large set intersection* lemma is lifted to the following result on *constraint looseness.*

Theorem 14 (Looseness) *Given a constraint network with domains that are of size at most d and constraints that are m-loose and of arity r, $r \geq 2$. It is strongly k-consistent, where k is the maximum value such that*

$$binomial(k-1, r-1) \leq \lceil d/(d-m) \rceil - 1.$$

Proof. Let $Y = \{x_1, x_2, \cdots, x_{K-1}\}$ be a set of any $K-1$ variables where $K \leq k$, $\bar{a}$ a consistent instantiation of the variables in Y, and x_K be any new variable. Let l be the number of relevant constraints to x_K. It can be shown that

$$l \leq binomial(K-1, r-1) \leq binomial(k-1, r-1) \leq \lceil d/(d-m) \rceil - 1.$$

So, according to Lemma 6, the intersection of extension sets to x_K is not empty. Hence, the constraint network is strongly k-consistent. □

To extend an instantiation of $k-1$ variables to a new variable, the *number* of extension sets (of the relevant constraints) matters when we try to apply the large sets lemma. We introduce the concept of extension degree for this number.

Definition 36 *Given a constraint network $\mathcal{R}$ and a set of variables $Y \subseteq N$. The* involvement degree *of a variable $x \in (N - Y)$ with respect to Y is the number of relevant constraints when extending an instantiation of Y to x. The* extension degree *of Y is the maximum involvement degree of all variables in $N - Y$. The* extension degree *of a positive number $k(< n)$ is the maximum extension degree of all subsets (of N) with k variables.*

Note the extension degree may not be an increasing function of k.

Example. Let $D = \{a, b, c\}$. Define a constraint network $\mathcal{R}$ with variables $\{x_1, x_2, x_3, x\}$, domains being D, and constraints

$$\begin{aligned} \{c_{S_1} &= D \times D \times D - \{(a, a, a)\}, \\ c_{S_2} &= D \times D \times D - \{(a, a, b)\}\} \end{aligned}$$

where $S_1 = \{x_1, x_2, x\}$ and $S_2 = \{x_2, x_3, x\}$. Let us use $\mathcal{R}$ to illustrate the concept of extension degree.

Consider a set of variables $Y = \{x_1, x_2, x_3\}$. The involvement degree of x with respect to Y in $\mathcal{R}$ is two. So, the extension degree of Y is two. The extension

degree of 3 is also two since the extension degree of any subset of $\{x_1, x_2, x_3, x\}$ with 3 variables is at most two. It can be verified that the extension degrees of 1 and 2 are zero and one respectively. □

Now a tighter lower bound of the inherent level of consistency is obtained by using extension degree.

Theorem 15 (Looseness) *A constraint network with domains that are of size at most d and constraints that are m-loose, is strongly k-consistent, where $k = n$ if the extension degree of any number from 1 to $n-1$ is less than or equal to $\lceil d/(d-m) \rceil - 1$; otherwise, k is the least number whose extension degree is greater than $\lceil d/(d-m) \rceil - 1$.*

This theorem is a consequence of Lemma 6.

In fact, the looseness Theorem 14 is a revised version of the following theorem by van Beek and Dechter [vBD97] which may overestimate the level of consistency [ZY03b].

Theorem 16 (Looseness) [vBD97] *A constraint network with domains that are of size at most d and constraints that are m-loose and of arity at least r, $r \geq 2$, is strongly k-consistent, where k is the minimum value such that the following inequality holds,*

$$binomial(k-1, r-1) \geq \lceil d/(d-m) \rceil - 1.$$

Here is a counter-example to Theorem 16.

Example. We construct a new network $\mathcal{R}'$ from $\mathcal{R}$ in the previous example by adding a constraint

$$c_{S_3} = D \times D \times D - \{(a, a, c)\}$$

where $S_3 = \{x_1, x_3, x\}$.

It is easy to verify that in $\mathcal{R}'$, every constraint is 2-loose and the arity of each constraint is $r = 3$.

In terms of Theorem 16, $\mathcal{R}'$ is strongly 4-consistent because the minimum k which satisfies

$$binomial(k-1, 2) \geq \lceil d/(d-m) \rceil - 1 = \lceil 3/(3-2) \rceil - 1 = 2$$

is 4. Consider the consistent instantiation (a, a, a) of variables $\{x_1, x_2, x_3\}$. Its extension sets to x with respect to c_{S_1}, c_{S_2}, and c_{S_3} are $\{b, c\}$, $\{a, c\}$, and $\{a, b\}$ respectively. Their intersection is empty, indicating that (a, a, a) is not extensible to x. Hence, $\mathcal{R}'$ is not strongly 4-consistent.

Theorem 14 implies that $\mathcal{R}'$ is strongly 3-consistent since the maximum k which satisfies

$$binomial(k-1, 2) \leq \lceil d/(d-m) \rceil - 1 = \lceil 3/(3-2) \rceil - 1 = 2$$

is 3. It is not difficult to verify that $\mathcal{R}'$ is strongly 3-consistent. □

Example. To illustrate the difference between Theorem 14 and Theorem 15, look at the network $\mathcal{R}$ again.

According to Theorem 15, $\mathcal{R}$ is strongly 4-consistent because the extension degree of every number from 0 to 3 is not greater than $\lceil d/(d-m) \rceil - 1 (= 2)$. However, it is only strongly 3-consistent in terms of Theorem 14.

It can be verified that $\mathcal{R}$ is strongly 4-consistent.□

Remark. 1) The problem in Theorem 16 does not affect the other theorems and corollary in [vBD97] on looseness of constraint. 2) Theorem 14 deals with a network of constraints with the same arity r while the statement in Theorem 16 deals with a network of constraints with arity of at least r. We find that Theorem 14 could be refined to deal with the latter case. However, the computation of k in

the refined version may be more complex than that in the current version. 3) Theorem 15 gives a more accurate estimation of k—the level of consistency—in a network than Theorem 14. Unlike Theorem 14, the estimation of k in Theorem 15 doesn't depend on the arities of the constraints in the network.

7.5 Application III: Relational Consistency and Directional Consistency

In the study of constraint networks, there are other definitions of consistency in addition to k-consistency used in this thesis. The the lifting lemma can be adapted to them. In this section, we will show the application of lifting lemma in the context of relational consistency and directional consistency.

7.5.1 Relational Consistency

Relational consistency is first introduced in [vBD95]. A weak version, which is used here, is proposed in [vBD97] and mainly serves to make the theory on tightness of constraints more elegant.

Definition 37 [vBD97] *A constraint network is* relationally m-consistent *if and only if given*

(1) any m distinct constraints $c_{S_1}, \cdots, c_{S_m}$, and

(2) any $x \in \cap_{i=1}^{m} S_i$, and

(3) any consistent instantiation $\bar{a}$ of the variables in $(\cup_{i=1}^{m} S_i - \{x\})$,

there exists an extension of $\bar{a}$ to x such that the extension is consistent with the m relations. A network is strongly *relationally m-consistent if it is relationally j-consistent for every $j \leq m$.*

In relational consistency, variables are no longer of concern. Instead, constraints are the basic unit of consideration. Intuitively, relational m-consistency concerns

whether all m constraints meet at every one of their shared variables. It makes sense because different constraints interact with each other exactly through the shared variables.

Relationally 1-, and 2-consistency are also called relationally arc, and path consistency, respectively.

All the results on *small set intersection* and *tree convex set intersection* in Section 7.1 can be lifted to results expressed by relational consistency.

Here is a new version of weak tightness under relational consistency.

Theorem 17 (Weak Tightness) *If a constraint network $\mathcal{R}$ of constraints with arity of at most r is strongly relationally $(m+1)$-consistent and weakly m-tight at level of $(m+1)(r-1)+1$, it is globally consistent.*

Proof. Let $j = (m+1)(r-1)+1$. The constraint network $\mathcal{R}$ will be shown to be k-consistent for all k $(j < k \leq n)$.

Let $Y = \{x_1, \cdots, x_{k-1}\}$ a set of any $k-1$ variables, and $\bar{a}$ be an consistent instantiation of all variables in Y. Consider any additional variable x_k. Without loss of generality, let $R_{S_1}, \cdots, R_{S_l}$ be relevant constraints, and E_i be the extension set of $\bar{a}$ to x_k with respect to R_{S_i} for $i \leq l$.

(*Consistency to Set*) Consider any $m+1$ of the l extension sets. Since the $\mathcal{R}$ is relationally $(m+1)$-consistent, the intersection of $m+1$ extension sets is not empty.

(*Set to Set*) The network is weakly m-tight. So, there must be an m-tight constraint in the relevant constraints $R_{S_1}, \cdots, R_{S_l}$. Let it be R_{S_i}. We know its extension set $|E_i| \leq m$. For every $m+1$ of the extension sets have a non-empty intersection, all l extension sets share a common element in terms of the *small set intersection lemma* (Lemma 7).

(*Set to Consistency*) From the lifting lemma, we have that $\mathcal{R}$ is k-consistent. $\square$

Comparing with the weak tightness theorem in previous section, the exposition of the result is neater and the proof is simpler.

For the sake of completeness, we also give here a new version of the tree convex theorem. The proof is omitted since it is a simplified version of one in Section 7.3 as hinted by the proof above.

Theorem 18 (tree convex) *Let $\mathcal{R}$ be a tree convex constraint network. $\mathcal{R}$ is globally consistent if it is strongly relationally path consistent.*

7.5.2 Make a Constraint Network Globally Consistent

Now let us turn to the main result in this section. Consider the weak m-tightness Theorem 13 based on strong k-consistency. Generally, a weakly m-tight network may not have the level of local consistency required by the theorem. It is tempting to enforce such a level of consistency on the network to make it globally consistent. However, this procedure may result in constraints with higher arity.

For example, consider a network with variables $\{x, x_1, x_2, x_3\}$. Let the domains of x_1, x_2, x_3 be $\{1, 2, 3\}$, the domain of x be $\{1, 2, 3, 4\}$, and the constraints be that all the variables should take different values:

$$x \neq x_1, x \neq x_2, x \neq x_3, x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3.$$

This network is strongly path consistent. In checking the 4-consistency of the network, we know that the instantiation $(1, 2, 3)$ of $\{x_1, x_2, x\}$ is consistent but can not be extended to x_3. To enforce 4-consistency, it is necessary to introduce a constraint on $\{x_1, x_2, x\}$ to make $(1, 2, 3)$ no longer a valid instantiation.

To make the new network globally consistent, the newly introduced constraints with higher arity may in turn require higher local consistency according to Theorem 13. Therefore it is difficult to predict an exact level of consistency (variable based) to *enforce* on the network to make it globally consistent.

Once resorting to relational consistency, it is possible to obtain global consistency through enforcing local relational consistency on the network. In order to achieve our main result, we need a stronger version of m-tightness—*proper* m-tightness.

Definition 38 *A constraint* c_{S_i} *is* properly m-tight *with respect to* $x \in S_i$ *iff for any instantiation* $\bar{a}$ *of* $S_i - \{x\}$,

$$|E_{i,x}| \leq m.$$

A constraint c_{S_i} *is* properly m-tight *iff it is properly* m*-tight with respect to every* $x \in S_i$.

For example, the constraint $x \leq y$, where $x \in \{1, 2, \cdots, 10\}$ and $y \in \{1, 2, \cdots, 10\}$, is 9-tight but not properly 9-tight. It is properly 10-tight since $|E_x(10)| = 10$ when $y = 10$.

Note when the extension set is the same as the domain of the extended variable, it does not affect the m-tightness of a constraint, but it does play a role in deciding the *proper* m-tightness of a constraint.

A constraint is m-tight if it is properly m-tight. The converse may not be true.

Definition 39 *A constraint network is* weakly properly m*-tight at level* k *if and only if for every set of variables* $\{x_1, \cdots, x_l\}(k \leq l < n)$ *and a new variable, there exists a properly* m*-tight constraint among the relevant constraints after an instantiation of the* l *variables.*

Let us re-examine the network in Table 7.1. Two sets of m-tight constraints were given to make the network weakly m-tight in the previous section. A careful look at the Table 7.2 immediately tells many other possible sets of m-tight constraints to make the network weakly m-tight. More specifically, we have the following observation.

Proposition 11 *A constraint network is weakly properly m-tight (and weakly m-tight respectively) if the constraint between every* two *variables is properly m-tight (and m-tight respectively).*

It can be verified that the proper tightness of the binary constraints is preserved during the procedure to enforce local consistency on a network like the one in Proposition 11. So we have the following result.

Theorem 19 (Weak Proper-Tightness) *Given a network whose constraint on every two variables is properly m-tight. It is globally consistent after it is made relationally $m+1$-consistent.*

This theorem follows immediately from the discussion above and Theorem 17. The implication of this theorem is that as long as we have several properly m-tight constraints on certain combinations of variables, the network can be made globally consistent by enforcing relational $m+1$-consistency.

We remark that this result is an improvement over van Beek and Dechter's result on tightness (Corollary 4 in this chapter). Their result requires all constraints to be m-tight. This requirement may be violated by the newly introduced constraints in the process of enforcing the intended level of relational consistency on the network.

7.5.3 Directional Consistency

A description of directional consistency is from [vBD97, page 560]:

> A backtracking algorithm constructs and extends partial solution by instantiating the variable in some linear order. Global consistency implies that for any ordering of the variables the solutions to the constraint network can be constructed in a backtrack-free manner; ..... Dechter and Pearl [DP87] observe that it is often sufficient to be backtrack-free along a particular ordering of the variables and that local consistency can be enforced with respect to that ordering only.

This special kind of local consistency is called directional consistency.

Definition 40 *A constraint network $\mathcal{R}$ is* directionally k-consistent *with respect to an ordering on variables if and only if for any consistent instantiation $\bar{a}$ of any distinct $k-1$ variables, and for any new variable x_k after those variables, there exists $u \in D_k$ such that $(\bar{a}, u)$ is a consistent instantiation of the k variables. $\mathcal{R}$ is* strongly directionally *k-consistent if and only if it is directionally j-consistent for all $j \leq k$.*

A strongly directionally n-consistent network can be solved without backtracking.

The weak m-tightness at level k of a network can be weakened to directionally weak m-tightness in the same way as we weaken k-consistency to directional k-consistency. The tree convexity of a network can also be weakened to directional tree convexity.

The small set intersection lemma and tree convex sets intersection lemma can be lifted to obtain results on directional consistencies in a directionally weak m-tight network and a directionally tree convex network respectively.

Consider the network in Table 7.1 again. Let us order the variables as $(1, 2, 3, 4, 5)$. To make the network directionally weak m-tight, in contrast to the possibilities in Table 7.2, it is only necessary to consider the extension of the instantiations of $\{1, 2, 3, 4\}$ to 5. Furthermore, only one of the relevant constraints to 5 to be m-tight, for example the one on $\{1, 2, 5\}$. In other words, we may need even less number of m-tight constraints to make the network directionally weakly m-tight. The requirement on tightness of a constraint can also be relaxed. In fact, the m-tightness of the constraint on $\{1, 2, 5\}$ *with respect to* 5 is sufficient.

We don't list the results on directional consistency here. However, they are obviously more effective in practice since they need less computation and require weaker properties on a constraint network.

7.6 Summary

We present a relationship between k-consistency and set intersection, and several properties on intersection of tree convex, small and large sets. A proof schema is provided to lift set intersection properties to consistency properties in a particular network. It allows us to study consistency from a new perspective of set intersection. The following are some examples on the relationship between set intersection and consistency. Let us rephrase some results reported in the previous sections in a *binary* constraint network. From Lemma 3, we know that the intersection of *all* n convex sets is not empty if the (*local*) intersection of every *two* sets is not empty. The consistency result is that given a corresponding row convex constraint network, strongly local (2+1)-consistency (or, arc and path consistency) in such a network implies global consistency (strongly n-consistency). Given a collection of small sets with at most m elements, that *every m+1* sets intersect induces the intersection of *all* sets in the collection (Lemma 5). The small set is then used to characterize the m-tightness of a constraint. Global consistency follows strongly (m+1)-consistency in a network with m-tight constraints. In Lemma 6, *all* large sets with at least m elements simply intersect without any local intersection information. The m-looseness of a constraint is characterized by large sets with at least m elements. Hence, a certain level of consistency depending on m is *inherent* in a constraint network with m-loose constraints. It suggests that more consistency results may be obtained by purely inspecting certain set intersection problems. One possible direction is to get a lower requirement on the local intersection information identified in Lemma 5 by imposing some additional structure on the sets.

We have demonstrated how the schema is used to derive existing consistency results [vBD95, vBD97] on networks with special properties. In addition to the results shown here, some other results can also be obtained by the lifting lemma. For example, the work of David [Dav93, Dav95] can be obtained by lifting the

singleton set Corollary 2. The work of Faltings and Sam-Haroud [SHF96] is on convex constraint networks in continuous domains and the idea there is to lift Helly's theorem on intersection of convex sets in Euclidean spaces.

Some new consistency results are discovered through the study of set intersection. Firstly, we generalize convex set to tree convex set while pairwise intersection of such sets still implies the intersection of all sets. Thanks to tree convex set, we identify a class of tree convex constraints which is a superset of row convex constraints [vBD95]. In a tree convex constraint network, global consistency is ensured by a certain level of local consistency.

Secondly, we show that in the small sets lemma, it is not necessary for all sets to be small. It is sufficient for one set to be small. This observation leads to that *in a network of arbitrary constraints, local consistency implies global consistency whenever there are some m-tight constraints on certain variables* (e.g. Theorem 13). This is an improvement over van Beek and Dechter's work on tightness [vBD97] where all constraints in a network are required to be m-tight. When the network does not have the required local consistency, global consistency may not be obtainable by enforcing such a level of local consistency. The reason is that the original property of the network may no longer hold after the introduction of new constraints in the process of enforcing the intended relational consistency. An interesting result we obtain is that as long as the constraint between every pair of variables is *properly* m-tight in an arbitrary network, global consistency can be achieved by enforcing a certain level of *relational* consistency (Theorem 19).

A promising line of work is to find more properties under which a network is weakly properly m-tight. Another direction is to find other classes of sets with intersection properties which will likely give useful consistency results.

Part V

Conclusion

Chapter 8

Conclusion

We have studied the consistency techniques from the following three aspects. The first is on the pruning aspect of consistency. The second is on predicting the level of consistency in a network. The third can be regarded as a special case of the second. It puts more emphasis on identifying tractable problems in CSP and designing efficient algorithms for them.

8.1 On the Pruning Aspect of Consistency

When pruning the search space is a main concern, usually a low level of consistency is enforced on the problem in practice. Specially, arc consistency is a good choice because of its relatively low cost and high pruning ability. Many efforts have been made to obtain efficient arc consistency algorithms. In this research, we develop an algorithm AC-3.1 which can be considered as a natural and simple new implementation of the traditional and influential AC-3. AC-3.1 is of optimal worst case time complexity. The result is surprising because for the last two decades AC-3 is regarded as a non-optimal algorithm but we show that it can be turned into an optimal algorithm. It is also an exciting result because AC-3 is simple, practically efficient, and widely used in the research community. The new implementation

brings AC-3 on a par with other worst case optimal algorithms, for example the AC-6 (considered as the best algorithm by the community), and provides more choices for users.

While worst case time complexity gives us the upper bound on the time complexity, in practice, the running time and the number of constraint checks for various CSP instances are the prime consideration. Our preliminary experiments show that AC-3.1 significantly reduces the number of constraint checks and the running time of the traditional implementation of AC-3 on hard arc consistency problems. Furthermore, the running time of AC-3.1 is competitive with the known best algorithms on the benchmarks from [BFR99]. The experiment done in [BR01] also shows some advantage of AC-3.1 (called AC-2001 in [BR01]) over other algorithms in maintaining arc consistency[SF94] during the search. We believe that AC-3.1 leads to a more robust AC algorithm for real world problems than other algorithms.

We also show how the idea behind AC-3.1 is employed to obtain a new algorithm for path consistency which is of best known worst case time complexity. We conjecture from the results of [CJ96] that this algorithm may also give a practical implementation for path consistency.

For general non-binary constraints, enforcing arc consistency becomes more expensive and its pruning ability may not outweigh its cost. In fact, to enforce arc consistency on a constraint network is NP-hard, which indicates that the pruning as simple as an arc consistency enforcing becomes very time consuming in a non-binary network. This suggests that we should be very careful in choosing a pruning strategy in solving non-binary constraint networks. It also suggests that it is worthwhile to study arc consistency on constraints with special properties. To this end, we identify a class of monotonic constraints on which arc consistency can be enforced in polynomial time. This result immediately implies that the ubiquitous linear inequalities can be made arc consistent efficiently. It also guarantees

that bounds consistency algorithms used by most constraint solvers [VH89] achieve arc consistency on linear inequalities, which may not be realized before. A more aggressive pruning strategy than arc consistency on a network with simple constraints may be intractable. For example, achieving relational path consistency becomes an NP-complete problem even for a network of two non-binary linear inequalities. The work reported here extends the results in [VHDT92] and complements the GAC-schema [BR97].

8.2 On Efficient Solving of Functional Constraints

An elimination algorithm is proposed to solve functional constraints both efficiently and elegantly. Its complexity of $\mathcal{O}(ed)$ to achieve minimality on a static network of functional constraints is the same as that of the best algorithm achieving arc-consistency [VHDT92].

An incremental variable elimination algorithm is designed to meet the requirements of a constraint programming system [JM94]. Its practical feasibility is suggested by the fact that the cost of the incremental algorithm is still much lower than that ($\mathcal{O}(ed^2)$) of a typical operation—arc consistency enforcing—widely adopted in constraint programming systems.

One application of our elimination algorithm for functional constraints is to solve 0/1/All constraints. 0/1/All constraints are studied in [CCJ94] and [Kir93], both of which give a sequential algorithm with time complexity of $\mathcal{O}(ed(n+d))$ to find one solution. In this thesis, we obtain faster algorithms with a time complexity of $\mathcal{O}(en)$ to solve a network with only "All" constraints and $\mathcal{O}(e(d+n))$ to solve a network with 0/1/All constraints. Furthermore, a network of 0/1/All constraints can also be made minimal in the same time complexity. Compared with [CCJ94, Kir93], a higher degree of consistency is obtained with more efficient algorithms.

8.3 On Predicting Consistency in a Constraint Network

As is shown by existing work in constraint networks, studying higher levels of consistency greatly helps to understand how to solve a problem. Progress has been made to understand the relationship between local consistency and global consistency in some constraint networks. In this thesis, a framework is proposed to predict the consistency in a network from a perspective of set intersection. It allows us to look at various results obtained so far, for example, those on row convex, m-tight and m-loose constraints respectively, in a uniform way. It leads to several new results on the level of consistency in a network as well as simplifying the derivations of existing results.

We have presented several properties on set intersection. They are either new or derived from the observations of other researchers. The new results include the tree convex sets intersection lemma and small set intersection lemma. The properties on set intersection are lifted to results on the consistency in a constraint network, through the lifting lemma and the proof schema.

The tree convex sets intersection lemma leads to the result that a network with tree convex constraints is globally consistent if it has a certain level of local consistency. It generalizes the well known result on row convex constraints [vBD95].

The small set intersection lemma leads to the weak tightness theorem which generalizes the result on m-tight constraints by van Beek and Dechter [vBD97]. An interesting new result is that a weakly properly m-tight network can be made globally consistent by enforcing local relational consistency. In the previous work on tightness of constraint, we could only predict the global consistency of a network through a certain level of local consistency *already present* in the network. For a network without the desired level of local consistency, we may not be able to achieve global consistency by enforcing such level of consistency because the enforcing

process may change the property of the constraint network. We have found that for an arbitrary network where the constraint on every two variables is properly m-tight, it can be made globally consistent by enforcing relational $m+1$-consistency on the network.

In summary, the framework not only unifies many existing results, but also exhibits much potential as a general technique for obtaining more results on consistency in constraint networks as shown in this thesis.

Appendix A

List of Symbols

The symbols frequently used in this thesis and their meanings are listed below.

$A \times B$	the Cartesian product of two sets A and B
N	the set of variables in a network
x_i	a variable in a network
i, j, k	simplified notations for variables in a network
x, y, z	variables in a network
D	the collection of domains in a network
D_i	the domain of variable x_i
$i.a$	a value a in the domain of variable i
C	the collection of constraints in a network
c_S	a non-binary constraint on a set S of variables
c_{ij}	a binary constraint on variable i and j
c	a general constraint or a value in a domain, depending on context
$vars(c)$	the set of the variables in constraint c
(i, j)	an arc (directed edge) from variable i to variable j
$c_{jk} \circ c_{ij}$	the composition of constraint c_{ij} and c_{jk}
n	the number of variables in a network
d	the size of the largest domain in a network
r	the maximum arity of the constraints in a network
e	the number of constraints in a network
$a, b, ...$	the values in a domain
$(a_1, a_2, ..., a_l)$	an instantiation of a set of l variables
$\bar{a}$	an instantiation of a set of variables
p_i^j	the pivot of c_{ij} in the domain of variable i
E	the set of edges of a graph before Chapter 7
E_i	an arbitrary set or a general extension set in Chapter 7
$E_{i,x}(\bar{a})$	the extension set of the instantiation $\bar{a}$ to x wrt a constraint c_{S_i}

Bibliography

[AB96] M. S. Affane and H. Bennaceur. A labelling arc consistency method for functional constraints. In *Proceedings of CP-96*, pages 16–30, Cambridge, MA, 1996.

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

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

[Bes94] C. Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 994.

[BFR99] C. Bessiere, E. C. Freuder, and J. Regin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[BMVH94] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(*intervals*) revisited. In *Proceedings of 1994 International Symposium on Logic Programming*, pages 124–138, 1994.

[BO97] F. Benhamou and W. Older. Applying interval arithmetic to real integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[BR96] C. Bessiere and J. Regin. Mac and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, MA, 1996.

[BR97] C. Bessiere and J. Regin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI-97*, Nagoya, Japan, 1997. IJCAI Inc.

[BR01] C. Bessiere and J. Regin. Refining the basic constraint propagation algorithm. In *IJCAI-2001*, pages 309–315, Seattle, WA, 2001. IJCAI Inc.

[BSH99] Christian Bliek and Djamila Sam-Haroud. Path consistency on triangulated constraint graphs. In *IJCAI-99*, pages 456–461, Stockholm, Sweden, 1999. IJCAI Inc.

[BvB98] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of AAAI-98*, pages 310–318, Madison, Wisconsin, 1998. AAAI press.

[CCJ94] M. C. Cooper, D. A. Cohen, and P. G. Jeavons. Characterizing tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[CD96] P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[CdGL+99] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

[CJ96] A. Chmeiss and P. Jegou. Path-consistency: When space misses time. In *Proceedings of AAAI-96*, volume 1, pages 196–201, Portland, Oregon, 1996. AAAI press.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of ACM*, 13(6):377–387, 1970.

[Dan63] G.B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[Dav93] Philippe David. When functional and bijective constraints make a CSP polynomial. In *Proceedings of Thirteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 224–229, Chambery, France, 1993.

[Dav95] Philippe David. Using pivot consistency to decompose and solve functional CSPs. *Journal of Arificial Intelligence Research*, 2:447–474, 1995.

[DBVH97] Y. Deville, O. Barette, and P. Van Hentenryck. Constraint satisfaction over connected row convex constraints. In *Proceedings of IJCAI-97*, volume 1, pages 405–411, Nagoya, Japan, 1997. IJCAI Inc. (See alsoArtificial Intelligence 109(1999): 243–271).

[Dec90a] R. Dechter. Decomposing a relation into a tree of binary relations. *Artificial Intelligence*, 41(1):2–24, 1990.

[Dec90b] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[Dec92a] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wily, New York, second edition, 1992.

[Dec92b] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[Dec99] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[DP87] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[DP89] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[DVHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 693–264, Tokyo, Japan, 1988.

[Fal94] Boi Faltings. Arc-consistency for continuous variables. *Artificial Intelligence*, 65(2):363–376, 1994.

[FBDR96] D. Frost, C. Bessiere, R. Dechter, and J. C. Regin. Random uniform CSP generators. `http://www.lirmm.fr/~bessiere/generator.html`, 1996.

[Fik68] R. E. Fikes. *A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1968.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of ACM*, 21(11):958–966, 1978.

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

[Fre85] E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of The ACM*, 32(4):75–761, 1985.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms. In *Proceedings of CCSCSI-78*, 1978.

[Gas79] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA, 1979.

[GB65] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of The ACM*, 12(5):516–524, 965.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. Freeman, San Francisco, CA, 1979.

[GMP+97] J. P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings of CP-97*, pages 327–340, Cambridge, MA, 1997.

[HE80] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hyv92] E. Hyvonen. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, 58:71–112, 1992.

[ILO00] ILOG. *ILOG SOLVER Reference Manual Version 5.0*. ILOG Inc., Mountain View, CA, 2000.

[JCG97] P. G. Jeavons, D. A. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of The ACM*, 44(4):527–548, 1997.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987.

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

[JMSY94] J. Jaffar, M. J. Maher, P. J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In *Proceedings of the 2nd Workshop on the Principles and Practice of Constraint Programming*, pages 370–395, Rosario, Orcas Island, Washington, 1994.

[KD94] S. Kasif and A. L. Delcher. Local consistency in parallel constraint satisfaction networks. *Artificial Intelligence*, 69:307–327, 1994.

[Kir93] L. M. Kirousis. Fast parallel constraint satisfaction. *Artificial Intelligence*, 64:147–160, 1993.

[KP88] L. M. Kirousis and C. H. Papadimitriou. The complexity of recognizing polyhedral scenes. In *Journal of Computer and System Sciences*, volume 37, pages 14–38, 1988.

[Lau78] J. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[Lho93] Olivier Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of IJCAI-93*, pages 232–238, Chambery,France, 1993. IJCAI Inc.

[Liu95] Bing Liu. Increasing functional constraints need to be checked only once. In *Proceedings of IJCAI-95*, pages 119–125, Montreal, Quebec, Canada, 1995. IJCAI Inc.

[LL98] Y. Lebbah and O. Lhomme. Acceleration methods for numeric CSPs. In *Proceedings of AAAI-98*, pages 19–24, Madison, Wisconsin, 1998. AAAI press.

[Mac77a] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):118–126, 1977.

[Mac77b] A. K. Mackworth. On reading sketch maps. In *Proceedings of IJCAI-77*, pages 598–606, Cambridge, MA, 1977. IJCAI Inc.

[Mac92] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wily, New York, second edition, 1992.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[MF93] A. K. Mackworth and E. C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI-88*, pages 651–656, Munich, Germany, 1988.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications. *Information Science*, 7(2):95–132, 1974.

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

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

[OV93] W. Older and A. Vellino. Constraint arithmetic on real intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming:Selected Research*, pages 175–195. MIT Press, Cambridge, MA, 1993.

[Pap81] C. H. Papadimitriou. On the complexity of integer programming. *Journal of The ACM*, 28(4):765–768, 1981.

[Per92] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[Pro93] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of IJCAI-93*, pages 262–267, Chambery, France, 1993. IJCAI Inc.

[PT93] P. Parodi and V. Torre. A linear complexity procedure for labeling line drawings of polyhedral scenes using vanishing points. In *Proceedings of IEEE International Conference on Computer Vision*, pages 291–295, 1993.

[Reg96] J. C. Regin. Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI-96*, pages 209–215, Portland, OR, 1996. AAAI press.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Boston, MA, second edition, 2000.

[RPD90] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, Sweden, 1990.

[Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of 10th ACM Symposium on the Theory of Computing*, pages 216–226, 1978.

[SF94] D. Sabin and E. Freuder. Contradicting conventional wisdosdom in constraint satisfaction. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 10–20, Rosario, Orcas Island, Washington, 1994.

[SHF96] D. Sam-Haroud and Boi V. Faltings. Solving non-binary convex CSPs in continous domains. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, pages 410–424, Cambridge, Massachusetts, 1996.

[Sin96] M. Singh. Path consistency revisited. *Int. Journal on Art. Intelligence Tools*, 6(1&2):127–141, 1996.

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[Tar75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of The ACM*, 22(2):146–160, 1975.

[Var00] M. Vardi. Constraint satisfaction and database theory: a tutorial. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems*, pages 76–85, Dallas, Texas, 2000.

[vBD95] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of The ACM*, 42(3):543–561, 1995.

[vBD97] P. van Beek and R. Dechter. Constraint tightness and looseness versus local and global consistency. *Journal of The ACM*, 44(4):549–566, 1997.

[VH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[VHD91] P. Van Hentenryck and Y. Deville. Operational semantics of constraint logic programming over finite domains. In *Proceedings of 3rd International Symposium on Programming Language Implementation and Logic Programming*, 1991.

[VHDT92] P. Van Hentenryck, Y. Deville, and C. M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 58:291–321, 1992.

[Wal72] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report MAC-AI-TR-271, MIT, Cambridge, MA, 1972.

[Wal75] D. L. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, Boston, MA, 1975.

[Wal93] Richard J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings of IJCAI-93*, pages 239–247, Chambery, France, 1993. IJCAI Inc.

[WF92] Richard J. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Nineth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, BC, 1992.

[Zha98] Yuanlin Zhang. Consistency techniques for linear arithmetic and functional constraints. Master's thesis, Dept. of Compter Science, Natl. Univ. of Singapore, Singapore, 1998.

[ZW98] Yuanlin Zhang and Hui Wu. Bound consistency on linear constraints in finite domain constraint programming. In *Proceedings of ECAI-98*, Brighton, UK, 1998. John Wiley & Sons, Ltd.

[ZY00] Yuanlin Zhang and Roland H. C. Yap. Arc consistency on n-ary monotonic and linear constraints. In *Proceedings of CP-00*, pages 470–483, Singapore, 2000.

[ZY01] Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI-01*, pages 316–321, Seattle, 2001. IJCAI Inc.

[ZY02a] Yuanlin Zhang and Roland H. C. Yap. Consistency and set intersection. In *Proceedings of AAAI-02*, pages 971–972, Edmonton, 2002. AAAI press.

[ZY02b] Yuanlin Zhang and Roland H. C. Yap. Incrementally solving functional constraints. In *Proceedings of AAAI-02*, pages 973–974, Edmonton, 2002. AAAI press.

[ZY03a] Yuanlin Zhang and Roland H. C. Yap. Consistency and set intersection. In *Proceedings of IJCAI-03*, pages 263–268, Acapulco, Mexico, 2003. IJCAI Inc.

[ZY03b] Yuanlin Zhang and Roland H. C. Yap. Erratum: P. van Beek and R. Dechter's theorem on constraint looseness and local consistency. *Journal of The ACM*, 50(3):1–3, 2003.

[ZYJ99] Yuanlin Zhang, Roland H. C. Yap, and Joxan Jaffar. Functional elimination and 0/1/all constraints. In *Proceedings of AAAI-99*, pages 275–281, Orlando, FL, 1999. AAAI Press.