Order Number 9119641

Anticipatory pruning networks and minimal 2LP (linear programming and logic programming)

Jo, Geun-Sik, Ph.D.

City University of New York, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

Anticipatory Pruning Networks and Minimal 2LP

(Linear Programming and Logic Programming)

by

Geun-Sik Jo

A dissertation submitted to the Graduate Faculty in Computer Science in patial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1991

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

1/24/91
Date

Professor Kenneth McAloon
Chair of Examining Committe

1/24/91
Date

Professor Thomas C. Wesselkamper
Executive Officer

Professor Melvin Fitting
Professor Carol Tretkoff
Professor Howard Wasserman

Supervisory Committee

The City University of New York

Abstract

Anticipatory Pruning Networks and Minimal 2LP

by

Geun-Sik Jo

Advisor : Professor Kenneth McAloon

In this thesis, the notion of the Anticipatory Pruning Network (APN) is introduced and developed for the 2LP system; 2LP (Linear Programming and Logic Programming) is a constraint logic programming system which has been developed and implemented at the Logic Based System Lab at Brooklyn College/CUNY. Using compilation of rules in the style of the Rete algorithm, the APN maps program clauses into a network. The APN prunes a search space by consistency checking and inconsistency propagation through the network and resets itself upon backtracking. The APN extends forward checking to continous constraint domains. Overall, the benchmarks show the APN to be an effective forward checking mechanism for both discrete and continuous problem domains for Simplex based constraint solvers. In particular, the APN is an effective pruning method for constrained optimization problems.

Acknowlegements

I would like to thank my dissertation commitee members, Professors Kenneth McAloon, Carol Tretkoff, Melvin Fitting and Howard Wasserman. I thank Professor Thomas Wesselkamper for his advice and assistance. My special thanks goes to my advisor, Professor Kenneth McAloon. Many of the ideas presented in this thesis originated with and have benefited from him. I gratefully acknowledge the moral support and valuable advice which he has given to me. I also acknowledge financial support from NSF grants.

The work in this thesis was done as a part of the 2LP project at Logic Based System Lab at Brooklyn College/CUNY. I would like to thank current and past members of the Lab - Seth, Beata, Assen, Coskun and Sudhir - for their contributions to my research.

I would like to thank all my friends in New York, especially Young Han Hur for making my life in New York enjoyable, both emotionally and intellectually. My thanks also goes to my family, especially to my grandmother, and to my professors at In-Ha University in Korea.

Finally, I would like to express my special thanks with love to my wife and lovely daughter.

Contents

List of Tables

List of Figures

To my family

Chapter 1

Introduction

1.1 Constraint reasoning

Constraint reasoning is a emerging technology which can apply to many fields in Artificial Intelligence. Applications for constraint reasoning include designing expert systems [Mittal et al 1986], designing a powerful declarative language [Jaffar Lassez 87], option trading [Lassez, McAloon, Yap] and software engineering [Gorlick et al]. [Dincbas et al 1] also introduced a logic programming language which integrates the efficiency of constraint solving with Prolog. The constraint is solved by the term rewriting in Bertrand [Leler] which is a general purpose constraint programming language.

There are several constraint solving techniques in use for the logic programming environment. First is the simplex method which used to be the constraint solving mechanism in linear programming. Although introducing the constraint solving technique in logic programming is rather recent, the Simplex Method which is used in constraint logic programming (CLP) languages was introduced in the 1940's. The CLP languages include CLP(R), 2LP, CAL, Prolog III and CHIP. Second, there is the Constraint Satisfaction Problem (CSP) paradigm which can be viewed as a domain filtering mechanisms rather than an algebraic constraint solving mechanism.

CSP with the tree search strategy is used in CHIP to solve discrete combinatorial problems. Third, there is the local propagation which can be viewed as the delay mechanism. This technique is used in CONSTRAINTS [Sussman Steele].

Constraint reasoning in logic programming was probably a desirable property at the dawn of Prolog because of the declarative power of constraint expression. The CLP languages will probably replace the conventional Prolog in next few years as [Cohen] pointed out.

1.2 Organization of the thesis

In the chapter two, we review several constraint solving techniques which are used in the logic programming environment described in the section one. The production system is also discussed in the section two. The Rete match algorithm which is considered as one of the best pattern matching algorithms in production systems is presented in detail in section two.

Constraint Logic Programming languages which utilized the constraint solving techniques described in the previous chapter are briefly reviewed. These include CLP(R), 2LP and CHIP. The two principal different approaches to constraint solving in CLP languages, i.e., the Simplex method and Constraint Satisfaction Problems (CSP), are also compared in section 3.3.

Our research concentrated on the Linear Programming and Logic Programming (2LP) system developed at the Logic Based System at Brooklyn College. The minimal 2LP system is to design the abstract machine architecture for the efficient implementation of CLP languages. Chapter four defines the 2LP class of languages. The theorectical aspect of constraint logic programming languages are also explored in this chapter.

The Anticipatory Pruning Network (APN) for the 2LP system is presented in the chapter 5. The APN is a kind of Forward Checking mechanism in 2LP language. The compilation of a 2LP program into the linearized network is described in detail in chapter five. The APN has two major components which are described in section 5.2. One is consistency checking with the current environment. The other is inconsistency propagation through the compiled network if inconsistency is found. The compilation also can be done in terms of the stratification which is defined in the section 5.3.

The chapter six, benchmarks are presented to show the effectiveness of the APN in the 2LP system. These benchmarks include some puzzles and real world linear programming problems. Chapter seven describes the related works in Forward Checking mechanisms. The APN is compared with the Forward Checking mechanism in Constraint Handling In Prolog (CHIP) and other mechanisms. This chapter also presents several different problem solving techniques in existence for comparisons.

Chapter eight briefly describes the parallel implementation of the APN algorithm for the future researches. As a conclusion, the APN is a powerful domain-independent Forward Checking mechanism for Simplex based constraint solvers.

1.3 Thesis summary

Constraint-directed searching is assisted by using the Anticipatory Pruning Network in the 2LP system. The APN is a linearized network which contains information about the availability of rules for Prolog-like top-down processing. In the tree search spaces, let the bottom-most nodes which have not been explored yet be called the *quick list*. And as each node is visited using a depth-first search strategy, the *quick list* is forced to perform consistency checking with the current environment (current Simplex configuration). As the system finds the constraints inconsistent with the

current environment, the inconsistency is propagated through the APN. As a result of the propagation of inconsistency, some search spaces can be pruned *a priori*. The benchmarks are performed to find the effectiveness of the APN in this research. The benchmarks show that the APN is an effective pruning method for the most of the problems. Especially, the APN is very effective on optimization problems in cases where the problems require implicit enumeration of the whole tree search spaces. In conclusion, APN is a domain-independent and flexible forward checking mechanism for Simplex based constraint solvers.

Chapter 2

Review

2.1 Constraint directed reasoning

2.1.1 Imperative versus constraint

The imperative assignment statements have dominated in conventional languages such as Fortran and Pascal. They have used the equal sign "=" for assignment in Fortran. The notation is often confused with the mathematical definition. A statement such as X = X + 1 is a source of confusion to the beginning programmer and is often used in conventional language for incrementing the variable X by 1. However, this statement, X = X + 1, is never true if we are considering the statement in terms of mathematics. The new notation ":= " for assignment is used in Pascal and Algol. In the imperative assignment, the statement should be of the form, < *Variable* = *Expression*>, with the condition that all the variables appearing in the expression should be grounded. The concept of mathematical definition is used as a guard in the conditional part of the if-statement although all the variables appearing in the conditional part should have values for evaluation in a conventional programming language. If any of the variables in conditional part of if-statement is not grounded, the error messages are generated in languages such as Pascal or Fortran.

Constraint solving is a good paradigm of declarative programming since we can

just describe the relationship among the variables or objects instead of arranging the assignment instruction manually to solve the problem.

Let us consider the statement, $X = Y + Z$, which is an equality constraint. To solve this constraint using imperative assignments in conventional language, the system should perform the following statements even if we assume that exactly two variables are grounded among the three variables X, Y and Z.

if Y is grounded and Z is grounded then $X = Y + Z$ else

if X is grounded and Y is grounded then $Z = X - Y$ else

if X is grounded and Z is grounded then $Y = X - Z$.

If we want to have the symbolic output when only one of variables is grounded, for example, as the constraint solver finds $X = 1$, the solver should output the relationship "$X = Z + 1$", then the translation into an equivalent set of cases in an imperative statements becomes much more unwieldy. In addition to that, to deal with more than one equality constraint, the constraint solver should be able to satisfy the simultaneous equations. The constraint solver should also take account for the different relations such as disequality ($\neq$) and inequality ($\geq$) which can vary on the domains of computation under consideration.

The computational complexity grows not only for the number of constraints which are involved, but it is also commensurate with the relations which appear in the constraints such as $\geq, =, \neq$ *and* $>$.

In the subsequent sections, various different constraint solving methods are explored. Constraint solvers which are used in the logic programming environment are explained in detail.

2.1.2 Constraint solving

The simplex method which was being mostly used in linear programming is used nowadays to replace the syntactic unification on the Herbrand Universe of logic programming. Variations of Simplex methods have been implemented in constraint logic programming systems as a major engine of computation such as CLP(R), Prolog III and CHIP. In addition to the basic concept of the Simplex method, some different techniques of constraint satisfactions problems in AI are explained concisely in this and the following section.

Local propagation

To preserve the concept of mathematical definition, the constraint should be of the form, <*Expression relation Expression*>, which is much more general than the imperative assignment. Moreover, the solver should infer the value of any variables as soon as the solver gets enough information about the values in the expressions. For instance, if we know that Y = 4, W = 7 and Z = 0 with the given constraint X − Y = W + Z, then the solver could infer X = 11.

The implementation of this kind of reasoning with delaying method is called *local propagation*. Graphically, the statement X − Y = W + Z can be represented in figure 2.1.

When the node receives a value, the value should be propagated along the arcs. After the node receive enough information from the arcs, the solver can infer the value of the node.

The local propagation is implemented in [Steele 1] to design a general purpose

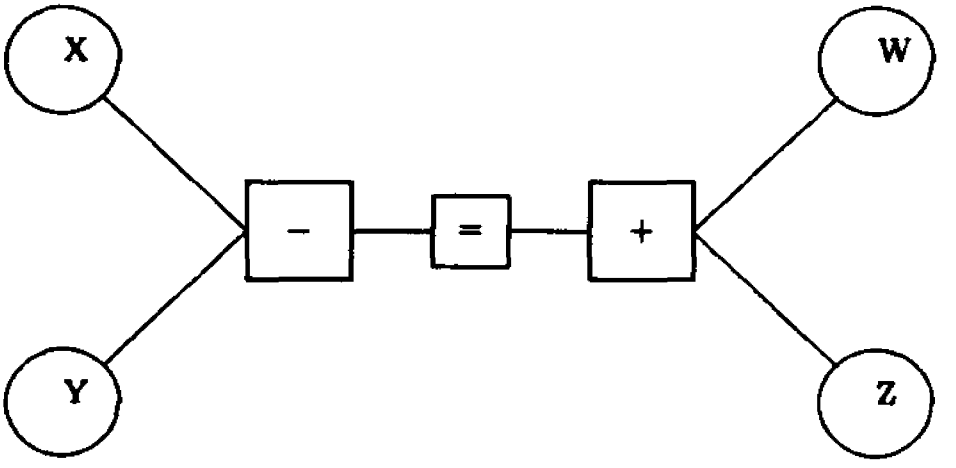

Figure 2.1: Graphical representation for X − Y = W + Z

language based on constraints. The major disadvantage of local propagation is that the value inference is basically local to each constraint. Therefore, the solver can not reason with the global constraint satisfaction among constraints themselves such as simultaneous equations.

Relaxation technique which is implemented in [Borning] is used to remedy the disadvantage of local propagation. The solver guesses the initial value of variable and propagates it and estimates the error which is caused by the initial guess and can guess again based upon the estimation. However, the relaxation method is slow even for linear equations. In non-linear equations, the relaxation technique may not converge. Moreover, the relaxation technique can only be apply to continuous numerical variable.

Simplex method

Linear programming which deals with problem optimization is concerned with the class of mathematical problems in which the relations among the variables are lin-

ear. Linear programming is mainly used to finding the best possible allocations of resources, machines or materials, etc., under the given set of restrictions (a set of constraints).

In general, linear programming is of such forms as the following.

Maximize (or Minimize) f(X1,X2,..... Xn)

where f(X1, X2, ... Xn) is linear.

Subject to $\sum$ AiXj $\geq Bi(i = 1, 2, ...m)$

$$Xj \geq 0(j = 1, 2, ..., n).$$

The most common method of solving linear programming is known as Simplex method which was developed by G. B. Dantizig in the 1940's. There are also interior point methods based on the Karmarkar algorithm for linear programming. From the computational complexity point of view, it is interesting that the Simplex method is a non-polynomial algorithm, where as the Karmarker algorithm is polynomial. However, the Simplex method usually runs in polynomial time. It is not clear that the Karmarker algorithm can outperform the Simplex method in the average case. Moreover, it is currently not known how to use interior point methods in logic programming environments.

In the Simplex method, the inequality constraint ($\geq$) is converted into equations by introducing slack variables. For instance, let us consider an inequality constraint, $A_1X_1+A_2X_2+\ldots+A_mX_n \leq B_1$. By introducing a slack variable,X_{n+1}, we can convert the inequality constraint into an equality constraint, $A_1X_1 + A_2X_2 + \ldots + A_mX_n + A_{m+1}X_{n+1} = B_1$. There should be m slack variables for m inequality constraints with nonnegative constraint for each slack variables for conversion. For each iteration in the Simplex method, one feasible solution $x_1, x_2, \ldots, x_{n+m}$ moves to another feasible

solution $x'_1, x'_2, \ldots, x'_{n+m}$ which is better than the previous one in terms of the given objective function. [Chvatal] is a valuable textbook for the detailed implementation of the Simplex method.

2.1.3 Constraint satisfaction problems

Introduction

Many important AI problems can be formulated as a constraint satisfaction problems (CSP). Examples of these problems, which have been researched for many years, include scheduling problems, vision and scene interpretation, and configuration problems. Along with Operation Research approaches to constraint solving, the AI community is doing research on constraint satisfaction problems which for most part deal with discrete domains of variables. However, although research in AI used to be separate from research on the constraint solving in O.R., the two disciplines of constraint solving recently started to merge in solving AI problems.

In standard constraint satisfaction problems, there are initially a fixed number of constraints which represent the binary relationship between variables. The domains and variables are known and fixed beforehand. Those variables range over discrete domains. CSP is different from any other constraint solver since the domain filtering is the only computation mechanism to solve constraints instead of using algebraic operations to reason with constraints.

Let us consider the N-Queens problem which can be perfect example for CSP since the constraints in this problem are all binary constraints which are fixed initially and the domain of variables are known beforehand. For simplicity, let N = 4. The 4-Queens problems can be described as the following way in terms of CSP.

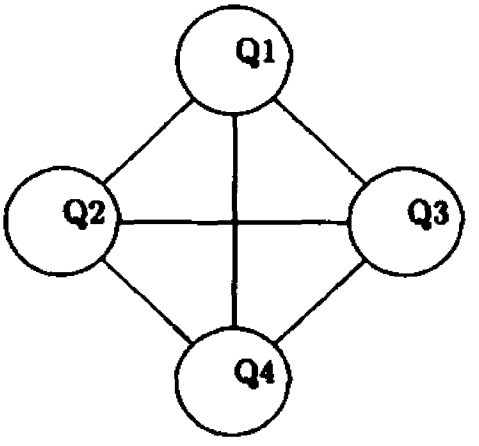

Figure 2.2: Constraint graph for 4-Queens problem

- Variables = {Q1, Q2, Q3, Q4}

- Domain Values = Q1, Q2, Q3, Q4 $\in$ [1,2,3,4]

- Constraints = $\{Q1 \neq Q2, Q2 \neq Q3, Q3 \neq Q4, Q1 \neq Q2 - 1,$
  $Q1 \neq Q2 + 1, Q1 \neq Q3 - 2, Q1 \neq Q3 + 2,$
  $Q1 \neq Q4 - 3, Q1 \neq Q4 + 3, Q2 \neq Q3 - 1,$
  $Q2 \neq Q3 + 1, Q3 \neq Q4 + 1, Q3 \neq Q4 - 1\}$

It is also possible to represent CSP problem as a constraint graph. Each node in the graph represents the variables which ranged over the discrete domain and the arcs denote the constraints. For instance, with 4-Queens problem, one of the possible constraint graphs is shown in the figure 2.2. Satisfying the constraints can be viewed as consistency checking in a constraint graph which is explained in the next section.

Extensions of the standard constraint satisfaction problem have been made by

several people such as dynamic constraint satisfaction [Mittal, Falkenhainer] and continuous-value CSP [Eastman] for two dimensional space planning problems.

Representation of constraints in CSP

Several researchers including Mackworth [Mackworth], Montanari and Freuder worked on the consistency of constraint networks. In this section, the basic concept of the representation of constraints is explored with respect to tree search algorithms.

To filter out the inconsistent value efficiently from the finite domain in the constraint network, the binary relation matrix is introduced. The relational matrix, Rij, represents the binary constraint Cij where i is an entry to the i-th value in the domain Vi and Vj represent an entry to the j-th value in the domain Vj. Furthermore, Rij(k,l) = 0 represents the fact that the k-th value in domain the Vi is incompatible with l-th value in the domain Vj. Rij(k, l) = 1 represents the fact that the k-th value in domain Vi is compatible with the l-th value in the domain Vj.

For instance, in 4-Queens problem, one of the constraint C1(Q1,Q2) = Q1 $\neq$ Q2 can be represented with the following binary matrix.

$$C1(Q1,Q2) = \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$

How do we reason with these binary matrices ? There are basically two operations on the matrix which are the intersection of relations and composition of relations. First, consider the intersection of relation with 4-Queens problem. There are three constraint between Q1 and Q2 which are C1(Q1,Q2) is Q1 $\neq$ Q2, C2(Q1, Q2) is

C1(Q1,Q2) = 0 1 1 1	C2(Q1,Q2) = 1 0 1 1	C3(Q1,Q2) = 1 1 1 1
1 0 1 1	1 1 0 1	0 1 1 1
1 1 0 1	1 1 1 0	1 0 1 1
1 1 1 0	1 1 1 1	1 1 0 1

Therefore, C123(Q1,Q2) = 0 0 1 1
0 0 0 1
1 0 0 0
1 1 0 0

Table 2.1: The matrix computation for C123(Q1,Q2)

Q1 $\neq$ Q2 - 1 and C3(Q1,Q2) is Q1 $\neq$ Q2 + 1. We could get the intersection of constraint, C1(Q1, Q2) and C2(Q1,Q2) and C3(Q1,Q2) by doing "and" operations which is shown in Table 2.1.

The second operation is to compute the arc-consistency in the network of constraint. The directed Arc(Vi,Vj) is consistent iff there exist any value in domain Vj to satisfy the constraint C(Vi,Vj) for any value in domain Vi. We should understand that the arc-consistency can not be always maintained after another value of a domain is removed to maintain the consistency of the network. For instance, if there are two arcs, Arc(Vi,Vj) and Arc(Vj,Vk), then Arc(Vi,Vj) may not be consistent after some values are removed from the domain Vj to maintain the consistency of Arc(Vj,Vk).

Consider the 4-Queens problem with the positions of Queens which is shown in Figure 2.3. In this example, placing the first queen in the first column, second row can remove all the domain of the second queen except the fourth row. This also does not totally remove the domains for the rest of Queens. Therefore, all the Arcs are still consistent. In other words, placing the first queen in the second row does not prevent totally from placing the rest of queens in some position in each column. However, if

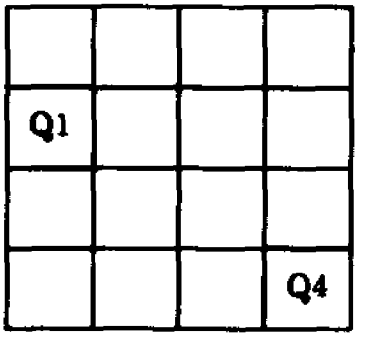

Figure 2.3: 4-Queens problem

we place the second queen in the fourth column and fourth row in next move, the domain of the second queen is removed totally. Therefore, the Arc(Q1,Q4) is not consistent anymore.

To compute these kinds of reasoning with binary relational matrix in 4-Queens problem, we need to compute R123'(Q1,Q4) = R123(Q1,Q4) and R123(Q1 Q2) * R123(Q2 Q4) which is shown in Table 2.2.

Note that the second row, fourth column of R123'(Q1, Q4) is 0 which means that placing the first queen in the first row, second column and the second queen in fourth row, fourth column makes placing any queen in the second row impossible.

2.1.4 Constraint programming languages

Various kinds of inference based on constraints are investigated in the AI literature. There is also research on programming languages based on constraints. These are mostly application oriented such as ThingLab [Borning] and Ivan Sutherland's Sketchpad for computer graphics. The Visicalc family of spreadsheet is also an elementary constraint based programming setup.

ThingLab, which is written in Smalltalk, is also an extension to Smalltalk-76 which

R123(Q1,Q2) = 0 0 1 1	R123(Q2,Q4) = 0 1 0 1	R123(Q1,Q4) = 0 1 1 0
0 0 0 1	1 0 1 0	1 0 1 1
1 0 0 0	0 1 0 1	1 1 0 1
1 1 0 0	1 0 1 0	0 1 1 0

Therefore, R123(Q1, Q2) * R123(Q2,Q4) = 1 1 1 1
1 0 1 0
0 1 0 1
1 1 1 1

R123'(Q1, Q4) = 0 0 1 0
1 0 1 0
0 1 0 1
0 1 1 0

Table 2.2: The binary matrix computation for R123'(Q1,Q4)

is an object-oriented programming language. In ThingLab, constraints are used to specify the relation between objects which are integers and texts. The combination of local propagation and relaxation techniques is implemented to solve constraints in ThingLab.

In Guy Steele's dissertation [Steele 1], a general purpose programming language based on constraints was defined and designed to be implemented on special purpose hardware. His attempt was to solve constraint by local propagation. However, the simultaneous equation was not addressed even if it often appears in the domain of constraint solving. Because of cyclic dependency, simultaneous equations cannot be solved by local propagation.

On the other hand, researchers [Jaffar Lassez 86], [Colmerauer 87],[Dincbas et al 1] recently have developed more powerful declarative programming languages than Pro-

log by exploiting constraint solving techniques. Those three systems include CLP(R), Prolog III and CHIP which are supersets of Prolog. These systems are using simplex-based constraint solving which is more powerful and general than any other constraint solving methods so far, according to my best knowledge. CSP technique is also used in CHIP to prune the domains of the search space for efficiency.

2.2 Production systems

2.2.1 Introduction

The production systems has been the critical domain of AI research for building expert systems and modeling intelligent human behavior. The production systems is generally composed of a set of if-then rules, the global databases associated with the rules and the interpreter which controls the implementation of production systems. Each of the if-then rules, called productions, consists of conjunctions of condition elements, which are if-part of the rules, and the actions, which are the then-part of rules. The if-part of the rules and the then-part of the rules are called LHS (Left Hand Side) and RHS (Right Hand Side) of the production, respectively.

2.2.2 OPS5

In OPS5, the global database is called working memory. The working memory in OPS5 is the list of zero or more of attribute pairs associated with a constant symbol which is called the class or type. For instance, the working memory with class name "person" associated with the attributes such as name and age and the value of name and age can be represented as (person ↑name Geun-Sik ↑age 10). The changes to the working memory are performed by propagating tokens through the network. The tokens here are the ordered pair of a tag and a data element. The tag '+' represents

the addition of data elements into working memory set. The tag '-' represents the deletion of data element from the working memory set. For instance, to modify the working memory element of (Person ↑name Geun-Sik ↑age 10) to (Person ↑name Geun-Sik ↑age 20), we need to process two tokens in OPS5 such as <- (person ↑name Geun-Sik ↑age 10)> and <+ (Person ↑name Geun-Sik ↑age 20)>.

The underlying mechanism which controls the execution of the processing of tokens over the Rete network is called the interpreter. The interpreter which is also known as the inference engine, performs the following recognize-act cycle:

1) Match : The LHS conditions of rules are matched against the working memory elements.

2) Conflict resolution : One of the rules is selected for execution among the satisfied LHS of the rules. If none exists, the interpreter halts.

3) Action : The action part of the rules is performed. In this phase, the contents of the working memories may be changed.

4) Go to 1)

2.2.3 Rete algorithm in production systems

The matching phase in production systems is critically intensive in the computation that it can take 90 percent of the total computation time in executing production systems even if we are considering the best known algorithm. One of the best known pattern matching algorithms is the Rete algorithm which is proposed by Forgy in [Forgy 79] and used in the OPS5 family of language. Forgy proposes a special internal representation for the LHS of the rule in memory to minimize the computation in the matching phase of the production systems.

The translation of rules into the internal representation is called the compilation

of rules which is used in the internals of the OPS5. Therefore, the compilation in OPS5 is nothing more than translation of rules into the representation of the set of intermediate instructions for checking a group of features in rules: Thus when the system loads the rules into memory for execution, it compiles them into a binary discrimination network in the form of the linearized set of intermediate instructions and loads them into the memory for checking a set of features of patterns of the rules. After compilation, the interpreter only performs the matching operations on the linearized intermediate instructions. However, in OPS83 which is the later version of OPS5, the rules were directly compiled into the machine executable code. Therefore, the interpreter was not necessary for OPS83. The linearized representation of the Rete network is considered as the abstract machine architecture for the fast implementation of production systems. What the production systems is to the linearized Rete network, Prolog is to the Warren Abstract Machine [Warren] instructions. Further information about rule compilation in OPS5 can be obtained from [Forgy 79], [Forgy 82]. A variation of the Rete network in the propositional case is implemented in our research. It is well described in the chapter four.

The three advantages of the Rete matching algorithm are summarized as the following:

First, The Rete algorithm takes advantages of the temporal redundancy of matching processes occurring in the cycles of the interpreter. Only the changed working memory elements are matched against the LHS of rules since the rest of matched elements are saved in the networks.

Second, The same condition elements of rules are shared to some extent by compilation in order for the actual matching in production systems to be reduced during run time.

Third, the networks save the information about the partial consistent bindings so that the binding is not necessarily recomputed in the subsequent cycles.

The improvement of the Rete algorithm has been studied by several researchers [Schor, Daly, Lee, Tibbitts]. The variations of the Rete algorithm were implemented in several different places such as IBM T.J. Watson Research Center and Columbia University. The parallel implementation of production systems based on the Rete algorithm was researched in [Gupta].

Chapter 3

Constraint Logic Programming

Constraint logic programming is very interesting in several different ways. The constraint solver in constraint logic programming is a variation of Simplex method in linear programming which is a major field of operations research, management science, mathematical programming, and industrial engineering. From a high-level point of view, the Simplex based constraint solver is inputted into a logic program in the design of languages such as CLP(R) and Prolog III.

From the programming language point of view, we can develop powerful declarative and expressive computer programming language such as CLP(R) by integrating a variation of Simplex with Horn clause logic. The constraint solver in constraint logic programming is mainly related to the declarativeness and invertability of the programming language. However, the purpose of using Simplex in operations research is to optimize the given costs or profits function associated with a set of constraints.

In addition to the Simplex based constraint solver in constraint logic programming, the concept of CSP is implemented in CHIP to combine the logic programming with the efficiency of constraint solving.

3.1 CLP(R)

3.1.1 Prolog vs. CLP(R)

It is widely known that Prolog is a kind of non-procedural and declarative programming language. However, constraint solving is avoided in Prolog. For instance, let us consider the following Prolog statements.

Example 1) g(X) :- $X >= 1, X + 1 >= X$.

Example 2) f(X, Y, Z) :- X is Y + Z.

Example 3) fact(N, F) :- N >= 1, Nprime is N − 1, fact(Nprime, M)
F is N * M.

fact(0,1).

The statement in example 1) is always true in term of constraint solving, but, in Prolog, it is true only if X has a value. Furthermore, in the second example, Prolog systems fail to evaluate the value of Y even if X and Z have ground values. Due to this fact, a Prolog system could not evaluate the value of X by querying in fact(X, 6) even if the system could evaluate fact(3, F). The constraint such as N >= 1 is of the form, <*Expression Relation Expression*>, and is evaluated successfully only if every variable has a ground value. This can be viewed as the condition part of the if-statement in conventional languages such as Pascal and Fortran. The "is" predicate in Prolog is practically same as imperative assignment in Pascal or Fortran. Therefore, numerical constraint solving in Prolog is as primitive as in a conventional language. This argument implies that conventional Prolog is a procedural language in solving the numerical constraints. To improve this kind of weakness in constraint solving in

Prolog, new styles of Prolog which the constraint solver is embedded in have been reserached and implemented. These include CLP(R), CHIP and Prolog III.

3.1.2 Overview of CLP(R)

The CLP scheme [Jaffar Lassez 87] defines a class of programming language based on constraint solving and logic programming. The framework which Jaffar and Lassez defined can be represented as CLP(X) where X is the domain of discourse over constraints. One instance of the CLP paradigm is CLP(R) where R represents the real numbers. The prototype version of the CLP(R) interpreter was implemented at Monash University in Australia. Later, a compiled version of CLP(R) was implemented at the IBM Thomas J. Watson Research Center at Yorktown Heights.

Unification in Prolog can be viewed as solving equality constraints over the Herbrand universe. In the CLP scheme, unification is replaced by solvability of constraints which is a more general computation scheme. Moreover, it yields a more declarative programing scheme than the conventional logic programming from the user's point of view. To consider the examples shown in the above section, in CLP(R), example 1 returns "true", example 2 outputs the answer properly if we use "=" instead of "is", and example 3 with query fact(X, 6) returns X = 3.

The CLP(R) interpreter consists of three different components of software modules which are the Prolog-like inference engine, the constraint solver, and the interface between the two. The inference engine also performs simple pattern matching, recognizing the constraints and solving simple constraints. The constraint solver is a variation of simplex method which also provides a delay mechanism for nonlinear situation. The interface between the engine and constraint solver is designed to transform the constraints into a canonical form.

The constraint solver in CLP(R) is adapted for the logic programming environment in order to manage the incrementality and backtracking behaviors of logic programming. The relations which appear in constraints such as $=$ and $\geq$ are natural to the simplex based constraint solver. However, the constraints which cannot be directly managed in the Simplex based method are solved in a different way. For instance, the nonlinear constraints are delayed until the constraints become linear, which can be viewed as a local propagation. The strict inequality constraint, $AX < b$ can be rewritten into "$AX \geq b$ and $AX \neq b$" which can be solved by simplex and delay mechanism.

In considering the relatively short history of CLP(R), the system is successful in both application and research aspects of new waves of programming language. Applications of CLP(R) include option trading in a financial market [Lassez, McAloon, Yap], software testing [Gorlick et al] and some electrical engineering problems [Heintze, Michaylov, Stuckey].

The major advantages of CLP(R) over the conventional Prolog can be summarized as follows.

1. The invertability of a program involving mathematical variables: The input and output are not distinguished from the user's point of view.

2. The expressive power of the language : The system allows users to express the implicit information as well as explicit information.

3. The relational symbolic output: Instead of just outputting the substitution of variables as a result of unification in conventional Prolog, the system can output the relationships of the variables.

3.2 Constraint Handling in Prolog (CHIP)

3.2.1 Overview of CHIP

CHIP is another constraint logic programming language which is developed by European Computer-Industry Research Center (ECRC). In the CHIP system, the declarative aspects of logic programming are combined with the efficient implementation of constraint solving technique such as CSP by introducing the domain concept in Prolog. The CHIP system has been implemented with several different constraint solving techniques which depend on the computation domains which they are dealing with, such as CSP for finite domain, *local propagation* for Boolean terms, and Simplex for rational terms. They use multiple independent constraint solvers in one programming language system. This is to achieve efficient implementation of constraint solving in the logic programming environment by embedding application specific constraint solvers in Prolog. This is somewhat analogous to the CLP scheme, which was described in the previous section. However, each constraint solver in CHIP system is more specific and more application oriented than those of the CLP scheme. For instance, the CSP with tree search strategy is used for integer programming and Boolean unification is used for digital circuit design. Comparing these two system, i.e.,the CLP scheme and CHIP, ends up with generality versus efficiency.

The CHIP system introduced the consistency technique which is the combination of the tree search strategy in Prolog and the concept of CSP. This is described in detail in the next section.

CHIP is quite successful in applications [Dincbas et al 2] for integer programming; circuit design such as circuit simulation; and microcode label assignment.

3.2.2 Consistency technique in CHIP

The consistency technique [Van Hentenryck] in CHIP is designed to solve combinatorial problems. In the tree search strategy in Prolog, there are three disadvantages: (1) the continual rediscovery of same fact, (2) late detections of failure, and (3) bad backtracking point. The motivation behind the consistency techniques is to minimize some disadvantages of Prolog, like search strategy, i.e., generate and test with backtracking.

The new features of the consistency techniques in CHIP over Prolog can be summarized as the following.

1. Domain concept is built into Prolog together with unification over the domain.

2. Implementation of CSP with tree search strategy in Prolog; The efficiency of CHIP is mainly related to the representation of constraints over finite domains since the operations for the active use of constraints are relational binary matrix manipulations.

3. Several built-in inference rules such as forward checking, looking ahead and partial looking ahead; the users can have the control over the inferences so that they can take advantages of the system for their applications.

3.3 Simplex based constraint solving vs. Constraint Satisfaction Problems (CSP)

Three constraint solving techniques such as local propagation, Simplex and CSP are used in the constraint logic programming environment. Variations of the Simplex method are mainly used for constraint solving in systems like CLP(R), 2LP and

Prolog III. However, in CHIP [Dincbas et al 1], the main constraint solver is the CSP with a Prolog like tree search strategy although CHIP provides a constraint solver based on the Simplex method for rational terms according to their specification in the language.

Empirical research [Van Hentenryck, Carillon] shows that the branch and bound method in linear programming cannot be as efficient as CSP in CHIP if we are dealing with integer programming. CSP can be much more efficient than the Simplex method in integer programming. However, there are some limitations in CSPs. First of all, the variables must range over a discrete domain. Moreover, the domain and variables appearing in the constraints are known and fixed initially. The constraints here are all static, which means that all the initially introduced variables and constraints should be used in problem solving. However, the Simplex based constraint solver allows us to introduce variables naturally and dynamically. There are some extensions of the standard CSP such as dynamic constraint satisfactions [Mittal, Falkenhainer] optimizing CSP [Navinchandra] and hierarchical-domain CSP [Mackworth], [Mittal].

However, those extensions are all separate constraint solvers for different computation domains, whereas the Simplex based method is one solver which can be used in all the domains of computation. The Simplex based constraint solver is more flexible and powerful than CSP. However, there are two fundamental disadvantages in a Simplex based constraint solver. One is that the Simplex method in logic programming environment is overly general and complex. It has been described as "killing a rabbit with an elephant gun" in [Cohen]. This inefficiency can be resolved by means of hardware and software development. On the other hand, is currently there any other constraint solver which can be integrated naturally into the logic programming environment ? The other disadvantage is that there are some relations which are

not natural in the Simplex method. Those relations are disequality ($\neq$) and strict inequality ($>$). In CLP(R) these relations can be solved effectively by introducing delay mechanisms which can be viewed as *local propagation*.

Can we integrate the generality of the Simplex based constraint solver with the efficiency of CSP ? One of the main concerns for CLP(R) implementors is efficiently testing the satisfiability of constraints. These implementation considerations include incrementality, simplification of constraints, and canonical forms. From the satisfiability point of view, the CSP in CHIP is efficiently and *a priori* to test the satisfiability of domains by pre-ordering the satisfiability of domains or manipulating the relational binary matrices. Some level of integration of CSP and the Simplex based constraint solver is addressed in the APN algorithm which is explained in chapter 7.1.3.

Chapter 4

Linear Programming and Logic Programming (2LP)

4.1 Preliminaries and Definitions

A domain $\mathcal{D}$ is given as a tuple $\mathcal{D} = (\mathbf{D}, R_1, \ldots, R_k, f_1, \ldots, f_l, a_1, a_2, \ldots)$ where $\mathbf{D}$ is a nonempty set, the R_i are relations on $\mathbf{D}$, the f_j are operations on $\mathbf{D}$ and the a_i are distinguished elements of $\mathbf{D}$. We also use $R_1, \ldots, R_k, f_1, \ldots, f_l, a_1, a_2, \ldots$ as symbols of the first-order language of the domain $\mathcal{D}$. Terms of this language are defined inductively: variables and individual constants are terms and $f_j(t_1, \ldots, t_n)$ is a term if f_j has arity n and $t_1, \ldots, t_n$ are terms. A term with no free variables is called a *closed* term or a *ground* term. Atomic formulas have the form $R_j(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and R_j is an n-ary relation. Atomic formulas are also called *constraints*. Formulas are obtained from atomic formulas by closure under boolean connectives and quantification. If the free variables of a formula F are $X_1, \ldots, X_n$ we write $F(X_1, \ldots, X_n)$ for F. An assigment is a map from a finite set of variables to $\mathbf{D}$; if the assignment θ is defined on the free variables of F we write $\mathcal{D} \models F\theta$ iff the assignment θ satisfies F in $\mathcal{D}$. The set of closed formulas G of the form $\exists X_1 \ldots \exists X_n C(X_1, \ldots, X_n)$ such that C is a conjunction of constraints and $\mathcal{D} \models G$ is

called the *constraint theory* of the domain $\mathcal{D}$.

In what follows, **2** denotes $\{0,1\}$, **N** the natural numbers, **Z** denotes the integers, **Q** denotes the rational numbers, **R** the real numbers, **C** the complex numbers and **AN** the algebraic numbers. To fix notation in the case of classical logic programming, we consider the language with constant symbol *nil* and binary functor *cons* and let **H** be the the Herbrand universe of ground terms in *cons* and *nil*. The constraint domain of interest here is $(\mathbf{H}, =, cons, nil)$; a set of constraints is then solvable if the corresponding unification succeeds.

It is also helpful to restrict the typing of certain binary operators. If f is such an operator on a domain, we write $\check{f}$ to restrict the first argument of f to be an individual constant and $\hat{f}$ to restrict the second argument of f to be an individual constant. Thus for multiplication $\check{*}$ allows multiplication by the distinguished coefficients from the domain. Thus for example the domain $(\mathbf{R}, =, +, \check{*}, 0, \pm 1, \pm 2, \ldots)$ is the additive group of the reals as a module over **Z**. This notation is also suitable for dealing with complexity considerations. Thus the domain $(\mathbf{R}, =, +, -, *, /, \hat{**}, 0, 1, 2, \ldots)$ of the reals as a field allows constraints with exponents written in binary as well as coefficients. It also serves to obviate the need for new notation; thus, $(\mathbf{N}, =, +, 1)$ denotes the natural numbers with the successor operation.

In the definitions that follow, $p, q_1, \ldots, q_t$ denote uninterpreted predicate symbols. Their semantics are determined by inductive definitions given in terms of logic programming rules. A CLP language is specified by a constraint domain $\mathcal{D}$ and conditions on the allowable rules to interpret predicate symbols..

Definition 1 *A vanilla – $CLP(\mathcal{D})$ program is given by a finite set of rules of the form*

$$p(X_1,\ldots,X_n) :- c_1(Y_{1,1},\ldots,Y_{1,k_1}),\ldots,c_s(Y_{s_1},\ldots,Y_{s,k_s}),$$
$$q_1(Z_{1,1},\ldots,Z_{1,m_1}),\ldots,q_t(Z_{t,1},\ldots,Z_{t,m_t}).$$

where the $X_1,\ldots,X_n,Y_{1,1},\ldots,Y_{s,k_s}$ *are variables and* $Z_{1,1},\ldots,Z_{t,m_t}$ *are variables or terms of the language of* $\mathcal{D}$.

Note that in the above definition, it is not required that the variables and terms $X_1,\ldots,X_n, Y_{1,1},\ldots,Y_{s,k_s}, Z_{1,1},\ldots,Z_{t,m_t}$ be distinct.

The use of the qualifying term "vanilla" is due to the fact that the notation CLP($\mathcal{D}$), in particular CLP($\mathcal{R}$), is used to denote the situation where Prolog structures form part of the constraint domain and the language is a superset of Prolog. Thus if we denote by $\mathbf{H}*\mathbf{R}$ the two sorted universe of structures and Real Numbers with $=$ as a shared relation, then CLP($\mathcal{R}$) is vanilla-CLP($\mathbf{H}*\mathbf{R},=,cons,nil,<=,<,+,-,*,/,0,1$). In [Sakai Aiba], the system *CAL* as defined is vanilla-CLP($\mathbf{AN},=,+,-,*,0,1,\ldots$).

Complexity analysis of constraint logic programming languages with many sorted constraints is a further topic of research.

If the constraint domain $\mathcal{D}$ does not have structures or lists, then vanilla-CLP($\mathcal{D}$) will be called Datalog($\mathcal{D}$). Classical Datalog, *e.g.* [Maier Warren], is a superset of vanilla-CLP($\mathbf{N},=,0,1,2,\ldots$).

In [Jaffar Lassez 86], [Jaffar Lassez 87] minimal model semantics for the CLP scheme are set out and completeness theorems are proved. Moreover, all examples of constraint systems explicitly considered here are solution compact with satisfaction complete theories and so the completeness results of [Jaffar Lassez 86], [Jaffar Lassez 87] on Negation-as-Failure also hold.

A vanilla-CLP($\mathcal{D}$) program P is interpreted in its minimal model where each

predicate symbol $p(X_1, \ldots, X_n)$ is interpreted as an n-ary relation $\bar{p} \subseteq \mathbf{D}^n$. We recall the minimal model construction of [Jaffar Lassez 86]: Inductively, define for each natural number m, and for all predicate symbols p of the program an approximation $\bar{p}^m$ as follows:

Step m = 0.

$\bar{p}^0 = \emptyset$

Step m = k+1.

For $d_1, \ldots, d_n \in \mathbf{D}$, $\bar{p}^m(d_1, \ldots, d_n)$ holds iff $\bar{p}^k(d_1, \ldots, d_n)$ holds or for some rule in P of the form

$$p(X_1, \ldots, X_n) :- c_1(Y_{1,1}, \ldots, Y_{1,k_1}), \ldots, c_s(Y_{s,1}, \ldots, Y_{s,k_s}),$$
$$q_1(Z_{1,1}, \ldots, Z_{1,m_1}), \ldots, q_t(Z_{t,1}, \ldots, Z_{t,m_t}).$$

and some assignment θ where $d_i = X_i\theta$ for $i = 1, \ldots, n$ we have $\mathcal{D} \models c_j\theta$ for $j = 1, \ldots, s$ and $\bar{q}_j^k(Z_{j,1}\theta, \ldots, Z_{j,m_j}\theta)$ for $j = 1, \ldots, t$.

So finally set $\bar{p} = \bigcup_m \bar{p}^m$.

Following [Jaffar Lassez 87] with appropriate change in notation, a *goal* is defined to be a set of the form

$$c_1(X_{1,1}, \ldots, X_{1,s_1}), \ldots, c_r(X_{r,1}, \ldots, X_{r,s_r}), p_1(Z_{1,1} \ldots, Z_{1,m_1}), \ldots, p_t(Z_{t,1}, \ldots, Z_{t,m_t})$$

where the $X_{i,j}$ are variables and the $Z_{k,l}$ are terms. Suppose we are given a vanilla-CLP($\mathcal{D}$) program P, a goal Γ, $p_j(Z_{j,1}, \ldots, Z_{j,m_j})$ in Γ and a variant of a rule P with no variables in common with Γ of the form

$$p_j(U_1, \ldots, U_n) :- c_1(Y_{1,1}, \ldots, Y_{1,k_1}), \ldots, c_s(Y_{s,1}, \ldots, Y_{s,k_s}),$$
$$q_1(T_{1,1}, \ldots, T_{1,n_1}), \ldots, q_t(T_{t,1}, \ldots, Z_{T,n_t}).$$

Set θ to be the substitution $\{U_1/Z_{j,1}, \ldots, U_{m_j}/Z_{j,m_j}\}$ which replaces each U_i by the term $Z_{j,i}$ for $i = 1, \ldots, n$. Then a *derivation step* is made by deleting $p_j(Z_{j,1}, \ldots, Z_{j,m_j})$ from the goal Γ and inserting

$$c_1(Y_{1,1}), \ldots, Y_{1,k_1})\theta, \ldots, c_s(Y_{s,1}, \ldots, Y_{s,k_s})\theta,$$

$$q_1(T_{1,1}, \ldots, T_{1,n_1})\theta, \ldots, q_t(T_{t,1}, \ldots, T_{t,n_t})\theta$$

to form a new goal. A sequence of goals starting with an initial goal and generated by derivation steps is called a *successful derivation* or simply a *derivation* if the final goal consists of a set of constraints whose conjunction is satisfiable in $\mathcal{D}$. In particular, a successful derivation of a goal $q(Z_1, \ldots, Z_n)$ from a program P yields a set of constraints

$$c_1(X_{1,1}, \ldots, X_{1,s_1}), \ldots, c_r(X_{r,1}, \ldots, X_{r,s_r})$$

whose conjunction is satisfiable in $\mathcal{D}$ and which are such that the universal closure of the following implication holds in the minimal model of the program:

$$c_1(X_{1,1}, \ldots, X_{1,s_1}) \wedge \ldots \wedge c_r(X_{r,1}, \ldots, X_{r,s_r}) \rightarrow q(Z_1, \ldots, Z_n)$$

To fix notation, we define a *query* to be a goal of the form $q(Z_1, \ldots, Z_n)$ where Z_i is either a variable or a ground term, $i = 1, \ldots, n$; if every term is a ground term, then the query is a *ground* query.. The query is *accepted* by the program if its existential closure is satisfied in the minimal model of the program. So, the program P accepts the query $q(Z_1, \ldots, Z_n)$ if and only if for some m the approximation $\bar{q}^m$ is not empty. Similarly, the program P accepts the ground query $q(t_1, \ldots, t_n)$ if and only if for some m, we have $\bar{q}^m(d_1, \ldots, d_n)$ where d_i is the denotation of the closed term $t_i, i = 1, \ldots, n$.

Thus, given a program P and query $q = q(Z_1, \ldots, Z_n)$, by the Succesful Derivations Lemma of [Jaffar Lassez 86], the existential closure of q is true in the minimal model of P if and only if there is a derivation of q from P. Note that this completeness result establishes a link between satisfiability in the minimal model of a constraint program and alternating Turing machine computation; *cf.* [Chandra Lewis Makowsky], [Shapiro] for the case of classical logic programming.

The *query complexity* of a given CLP language L is the computational complexity of the formal language consisting of all pairs (P, q) such that P accepts q. The *ground query complexity* of a CLP language is the computational complexity of the formal language consisting of all pairs (P, q) such that q is ground and P accepts q.

We will make use of standard complexity theoretic terminology and complexity classes such as P, NP, PSPACE, RE (Recursively Enumerable), etc.

We summarize the discussion thus far with the following:

Theorem 1 (Completeness Theorem) *Let $\mathcal{D}$ be a domain with constraint theory T. Then for every vanilla-CLP($\mathcal{D}$) program P and every query q, P accepts q if and only if there is a derivation of q from P. Moreover, if T is in RE, then the query complexity of the language vanilla-CLP($\mathcal{D}$) is also in RE.*

For further completeness results for constraint logic programming, see [Maher]; for classical logic programming see [Lloyd].

We now give a classification of CLP languages over a domain $\mathcal{D}$ by means of simple syntactic criteria which turn out to be equivalent to distinctions in storage management and in logic.

Definition 2 *A* minimal-*CLP($\mathcal{D}$) program is given by a finite set of rules of the form*

$$p(X_1,\ldots,X_n) :- c_1(Y_{1,1},\ldots,Y_{1,k_1}),\ldots,c_s(Y_{s,1},\ldots,Y_{s,k_s}),$$
$$q_1(X_1,\ldots,X_n),\ldots,q_t(X_1,\ldots,X_n).$$

where the $X_1,\ldots,X_n$ are variables and for each i, the $Y_{i,1},\ldots,Y_{i,k_i}$ form a subset of $X_1,\ldots,X_n$.

Note that this means that the variables $X_1,\ldots,X_n$ are all global variables of the program and that there are no local variables. The condition that the variables appear in the same order in all the procedures of the program makes the unifications trivial and thus the logic reduces to propositional logic. In particular a minimal-CLP program without constraints reduces to a Proplog program. However, the presence of constraints makes minimal-CLP much more powerful than Proplog, *cf.* Theorem 5, below. The restriction on the variables entails, however, that the storage locations $X_1,\ldots,X_n$ are the only ones that can be addressed in the course of a computation. However, the memory is not static in the sense that the least size of solutions to the set of constraints generated by a successful computation is not bounded *a priori*. On the other hand, we show that, in the cases of interest here, the size of the storage required for such solutions can be, in fact, bound as a function of the constraint domain and the program itself.

We analyze the complexity of minimal CLP languages.

Theorem 2 *The languages minimal-CLP*$(\mathbf{N},=,0)$ *and Proplog have P-Complete query complexity.*

The proof is classical, [Jones Laaser].

Theorem 3 *The following languages have P-Complete ground query complexity:*

minimal-CLP(**2**, =, 0, 1)

minimal-CLP(**N**, =, <=, <, +, *, $\widehat{**}$, 0, 1, ...)

minimal-CLP(**Q**, =, <=, <, +, −, *, 0, ±1, ±2, ...)

minimal-CLP(**R**, =, <=, <, +, −, *, 0, ±1, ±2, ...)

minimal-CLP(**C**, =, +, −, *, /, $\widehat{**}$, 0, 1, ...)

minimal-CLP(**H**, =, *cons*, *nil*)

Proof: Because the logic is propositional in minimal-CLP languages, when the queries are ground, the values of $X_1, \ldots, X_n$ can be used to prune all rules whose constraints are not satisfied by the given instantiation of $X_1, \ldots, X_n$ and to eliminate the constraints from the remaining rules. This reduces the computation to the Proplog case. □

Theorem 4 *The following languages have NP-Complete query complexity:*

minimal-CLP(**2**, <)

minimal-CLP(**2**, =, 0, 1)

minimal-CLP(**N**, =, <=, <, +, *, 0, 1, ...)

minimal-CLP(**Q**, =, <=, <, +, −, *, 0, ±1, ±2 ...)

minimal-CLP(**R**, =, <=, <, +, −, *, 0, ±1, ±2, ...)

minimal-CLP(**H**, =, *cons*, *nil*)

Proof: The proof is given in [Cox, McAloon, Tretkoff]. □

We remark that the complexity of minimal-CLP(**R**, =, <=, <, +, −, *, 0, 1, ...) does not change if we allow negative constraints in the sense of [Lassez McAloon].

These results clarify the relationship between linear and integer programming, on the one hand, and constraint logic programming on the other. Phase I of linear programming is minimal-CLP($\mathbf{Q}, =, <=, +, -, *, 0, 1, \ldots$) with only one rule and with only constraints on the right-hand-side. This language has PTIME query complexity. Phase I of integer programming is minimal-$CLP(\mathbf{N}, =, <=, +, *, 0, 1, \ldots)$ with only one rule and with only constraints on the right-hand-side. This language has NP query complexity.

4.2 Motivation of the 2LP language

The 2LP system [McAloon Tretkoff] is designed to provide flexible and programmable tools in solving mixed integer linear programming problems although its applications are not limited to them. 2LP system has been implemented at Logic Based System Lab. at Brooklyn College of the City University of New York.

More specifically, the 2LP system is designed to overcome or improve the disadvantages of CLP(R) which are considered as the following;

1. CLP(R) is not designed to deal with a large number of variables with many constraints.

2. Constraint solver is a black box from the users point of view. Therefore, the users cannot directly use the solver for their own purposes.

3. The system cannot optimize for given functions. The optimization function which is a major feature of linear programming is abandoned in designing the CLP(R) system.

4. The output of constraints is another problem. It is easy to lose control of the

true inequality constraint on the problem variables because Fourier's elimination is expensive.

In summary of the disadvantages in CLP(R), the power of constraint solver is undermined in the system although CLP(R) system has increased the declarative and expressive power of programming language by using a variations of Simplex method in solving constraints.

The 2LP of language is also designed to integrate the expressive power of logic with the declarative power of constraint expressions. In our current implementation of 2LP system, the logic is propositional and the domain is real. [Cox, McAloon, Tretkoff] provides the hierarchy of 2LP language with a complexity analysis of constraint logic programming in general.

Chapter 5

APN algorithm

Constraint based searching in top-down processing is assisted by the Anticipatory Pruning Networks (APN) to minimize backtracking. The APN algorithm computes the consequences of the constraints inconsistent with the current environment. The APN also explicitly saves the consequences of the inconsistency in the network. Furthermore, the APN algorithm can undo the processes of computing and saving the consequences of the inconsistent constraint to meet the backtracking behavior of logic programming.

This chapter describes implementation details of the APN algorithm including the compilation of the rules in the 2LP system, consistency checking and the relationship with the stratification.

5.1 Rule compilation in the 2LP system

Since there are no variables in the condition elements of the rules, the consistent bindings are not necessary to compute in the 2LP system. However, the first and second advantages of the Rete algorithm explained in 2.2.3 are directly applicable to the propagation of the inconsistency in the 2LP system.

The compilation of rules in the 2LP system is simpler than the compilation done

in the OPS5 system since we are only dealing with propagating inconsistency in constraint solving. The condition elements of the rule here have only the name of the class and do not have any attributes in it if we consider it in the OPS5 context.

More precisely, a 2LP program consists of a set of then-if Horn-clause rules which have the following form;

P $\leftarrow C_1, C_2, \ldots, C_n, Q_1, Q_2, \ldots, Qn$. where C_i are linear constraints with n ≥ 0 and Q_i, $i \geq 0$, are atoms.

In compilation, the constraint elements of C_i are treated as another logical atomic element to form the network from the body of the rule since the constraint is either consistent or inconsistent with the current environment. Furthermore, the same condition elements in the bodies of the set of rules are to be factored out so that the test can be performed once during the execution time. To maximize the sharing of the same condition elements and even the set of condition elements in different rules and minimize the computation involved in finding the set of the same combination of atomic elements, the condition elements of the rules are sorted before the compilation.

In APN, the rules are compiled into the data flow network which is logically equivalent to the set of rules. However, only the LHS of rules are compiled into networks in the Rete match algorithm. The nodes of APN are represented in a form of von Neumann machine instructions as in OPS5. The links between the nodes are not maintained explicitly. Rather, the successor nodes are placed at the immediate adjacent memory location for navigating the nodes efficiently in the von Neumann machine. The compiled APN here is represented by the linearized instruction set which can be executed by the interpreter. However, it is not enough to design the network by the linearization of the instructions if we consider the fact that some of the nodes in APN have more than one predecessor and successor. Therefore, two

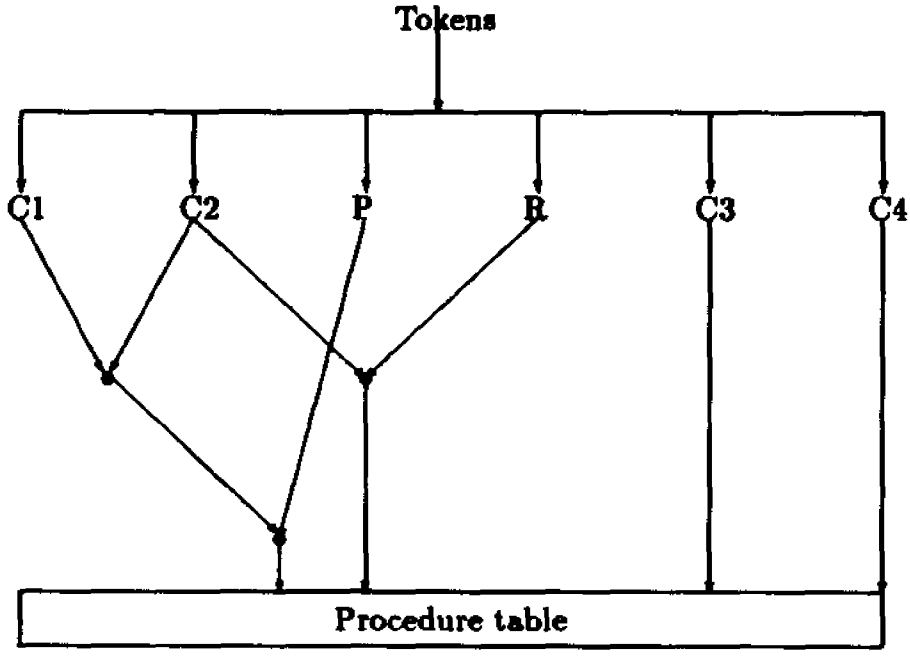

Figure 5.1: Binary Networks for KB1

more nodes are introduced. One is the Fork node which represents the fact that the node has more than one successor. The other one is the Merge node which represents the fact that the node has more than one predecessor. Figure 5.1 shows an example of the network for the set of rules, KB1 which is defined as the following:

KB1 = {Q ← C1, C2, P.

Q ← C2, R.

P ← C3.

P ← C4.}

The linearization of the network in an efficient machine executable form is important for real applications. For the basic human readable forms of linearized network in APN, the following five different types of nodes are necessary to linearize the APN:

1. (Fork label) ; The label represents the position of a node for another successor.
2. (EQ atom) ; One-input node for testing for equality.
3. (AND Left-memory Right-memory) ; Two input node which has the left memory and right memory. The left-memory takes input from the previous node and the right memory takes input from the Merge node.
4. (Join label) ; This node is the same as the *goto* instruction in the von Neumann machine architecture, but affecting only the right-memory of the AND node.
5. (Update rule-id head) ; Update the counter of head of the rule and the list of available rules for selections.

The illustration for the linearization of the network for the KB1 is shown in the Table 5.1.

5.2 Anticipatory Pruning Networks

There are two functionally different components of computation in APN. One is finding the constraints inconsistent with the current environment. The other is forward reasoning with these inconsistent constraints. In testing consistency, an incremental version of the Simplex algorithm is used. By propagating inconsistency, all deterministic goals are selected and resolved at once. Moreover, failure can be detected easily

	Fork	1	
	EQ	C1	
2	AND	T	T
3	AND	T	T
	Update	R1	Q
1	Fork	4	
	EQ	C2	
	Fork	6	
	Merge	2	
4	Fork	5	
	EQ	P	
	Merge	3	
5	Fork	8	
	EQ	R	
7	AND	T	T
	Update	R2	Q
6	Merge	7	
8	Fork	9	
	EQ	C3	
	Update	R3	R
9	EQ	C4	
	Update	R4	P

Table 5.1: Linearization of network for KB1

if an unsolvable atom occurs in the goal list.

5.2.1 Consistency checking with the current environments

Forward checking is successfully implemented and utilized in CHIP for an application to solve discrete combinatorial problems. CHIP introduced the domain concept in Prolog for users to specify the range of a discrete variable. With the active use of constraints, the problem can be pruned to reduce the search space for the given problem. The general concept of the various searching mechanisms including forward checking is well described in [Haralick, Elliott]. The Anticipatory Pruning Networks here is somewhat analogous to the consistency technique used in CHIP. Comparisons of both techniques are discussed in chapter 7.

In the 2LP language, the number of constraints is fixed initially by the programmer. Moreover, unlike CLP(R), the 2LP system is not supposed to generate new variables and constraints at run time. The variables appearing in a 2LP program as well as classic linear programming are all global. In 2LP, a set of constraints is fixed initially by the program and the system generates a consistent subset of these constraints to solve a problem.

Let us consider the consistency checking in the 2LP system. Let the active constraints (AC) be the constraints which enforced by the logic interpreter (Mission Control module in 2LP). Let us call the set of all constraint for the given program the quick constraints (QC). The quick constraints are also called *quick list*. Finally, let us call the constraints inconsistent with the current environment the dead constraints (DC). Whenever the constraint is enforced by the logic interpreter, the system checks consistency with the current environment. If the enforced constraint is consistent, the current environment is updated. Furthermore, the constraint is removed from

the *QC* and added to the *AC*. To find out the inconsistent constraints with the current environment, the system performs consistency checking with the rest of the *QC*. If the enforced constraint is not consistent, it is also removed from the *QC* and added to the *DC*. Then the consistency checking in 2LP is to check the consistency of all the constraints in *QC* with the newly updated environment. Therefore, everytime the current environment is updated, consistency checking is performed.

After the system finds the inconsistent constraints within each environment, the inconsistency is propagated through the precompiled network(APN) which is explained in the subsequent section. Some of the search tree can be pruned out *a priori* as a result of the inconsistency propagation. This is called Anticipatory Pruning.

Instead of checking consistency on all the constraints in the *QC* with the current environment, we can have the system choose the several different set of constraints. One of them is to compute the relevant constraints with the goal list which are the constraints set that can be derivable from the left most branch of the search space. This is to detect the early failure in the current direction of search space. This is called Partial Anticipatory Pruning. If we check the consistency with all constraints in *QC*, it is called Full Anticipatory Pruning. The benchmark results for each technique are demonstrated in Chapter 6.

5.2.2 Inconsistency propagations in APN

Since we are interested in the inconsistent constraint set to prune unnecessary searching, all the two-input memories are initially set to be available, which means that every rule is available for selection at the beginning. As the logic interpreter finds the inconsistent constraints with the current constraint environment, the system generates the negative token and the interpreter of APN propagates the negative token down

to the APN. The interpreter also can undo what has been done by propagating the positive tokens to take care of the backtracking behavior of logic programming style of control mechanism.

One of the most significant differences between APN and production systems in control structure is that APN does not resolve the conflict for the successful matchings, unlike the production system in general. Rather, APN propagates the inconsistency for all the possible branches of searching to prune the search space as much as possible. This is also computationally feasible since we are dealing with only propositional Horn clause logic. But, they have the similarity between the inconsistency propagation in APN and the control structure in production systems.

The APN interpreter basically performs the following *inconsistency propagation-update cycles.*

1. Match : The condition elements in the body of rules are matched against the atoms which refer to the inconsistent constraints with the current environment.
2. Propagating and updating the procedure table : The successfully matched atoms are propagated depending upon the inconsistency found in the condition elements in the same rule. If the inconsistency is propagated successfully, the counter associated with the head is decreased by 1 and the index of the rule associated with the head is removed from the available rule list for that head. If the backtracking occurs, the counter associated with the head is increased by 1, and the index of the rule is added to the available rule list.
3. Selection : The head with the counter changed to 0 or the counter changed

to 1 from 0 is selected for the further propagation.

4. go to 1.

To perform the matching in the APN algorithm efficiently, the token pair (tag atom) is hashed into the location of the atom in the linearized APN, where the tag = '+' indicates that backtracking has occurred and tag = '-' indicates that the system has found a constraint inconsistent with the current environment. Let the set of nodes of the linearized APN which resides in the memory be called *dna* and let the pointer of the current node being processed be called *ctr*, which represent the node pointer. There are two stacks which are declared globally in the interpreter. One is named *br-stack* which has the list of unprocessed nodes. The other one is named *selection-set* which is for the created tokens by propagating token in the previous cycle. In the selection phase, the heads of the rules which the counter changed to 0 from 1 or changed to 1 from 0 are chosen for the further propagation update. However, the chosen head which has the counter changed to 0 has the tag = '-', such as <- head>. The chosen head which has the counter changed to 1 from 0 has the tag = '+', such as <+ head>. These tokens associated with the tag are pushed into *selection-set* for propagation to the next cycle. The description of implementation in the APN interpreter is provided by the Lisp code in the Appendix A.

5.2.3 Early detection of failure and deterministic goal selection

The APN allows the logic interpreter (in 2LP, Mission-Control module) to select the goal deterministically by sharing the procedure table or by communicating with the logic interpreter. The heads of the rules are sorted out so that the same atomic elements can be shared by the same procedure table. We do not maintain the counter

associated with each procedure in the real implementation. However, each procedure virtually maintains the counter which indicates the number of the same head element existing in the program. Moreover, each corresponding head has the list of rules which represents the availability of the rules for selection. All the rules are initially available for selection. The data structure for each procedure in terms of Lisp syntax is created during the compilation step such as

(*atom counter* (index-of-rule(1) index-of-rule(2) ... index-of-rule(n)),

where the atom represents the name of the procedure appearing in the head of the rule and index-of-rule(i) represents the indexes of the available rules for selection for that head. Moreover, the interpreter constantly updates the counter and the list of available rules as the logic interpreter generates negative and positive tokens for each cycle.

The goal list in the 2LP system consists of the collections of constraints and atoms. This is of the form

? - $C_1, C_2, \ldots, C_n, A_1, A_2, \ldots, A_n$.

where C_i, $0 \leq i \leq n$, are constraints, A_i, $0 \leq i \leq n$, are atoms (procedures), $n \geq 0$, $m \geq 0$ and $n + m > 0$.

To resolve the goal list, we have to choose an atom among A_i, $1 \leq i \leq m$, for derivation and to check for the consistency with the current environment. The strategy for choosing an atom from the goal list for derivation is called the selection strategy. Since there are several ways of proving each atom, the system has to determine which rule should be chosen for derivation. This is called the search strategy. By maintaining the procedure table for each atom in APN, we can choose an atom with the smallest counter in it. If the counter for any atom in the goal list is zero, then the system should backtrack since it means there is no way to prove all the atoms with

those combinations of search. We can deterministically choose an atom by selecting an atom which has counter = 1. If a goal is deterministic, the system selects it at once. This does not change the operational semantics of the language.

5.3 APN in terms of stratification

The rules in the 2LP system are all definite Horn-clauses and their ground atoms are nothing but constraints. The program in the 2LP system is always stratified by the definition of stratified program in [Apt, Blair, Walker] since there are no negative condition elements in the program. Recently, [Lassez, McAloon, Port] extended the stratification defined by [Apt, Blair, Walker]. Their definition allows a fine grain stratification which accounts for recursion through positive occurrences of procedure. We want to restrict the stratified program in such a way that there are no negative or even positive cycles in the dependency graph of the program. If there is some program which has such a cycle, then the APN would not have the property of truth maintenance behavior after the inconsistency propagation and backtracking. For instance, let us consider the program KB2 = {R ← C1, Q. Q ← C2, R}. If the system propagates the inconsistency of constraint C1 and backtracks afterward, then the system cannot get back to the same list of available rules. If there are some rules involving in cycles, these rules are excluded in constructing the network. Therefore, only the remaining rules are compiled into the linearized network.

5.3.1 Compilation of stratification

The rules in the Anticipatory Pruning Network (APN) consist of the set of constraints which can be regarded as the set of atomic assertions and the propositional rules which are the following;

1. a set of atomic assertions which are constraints.
2. a non-empty set of definite Horn-clause rules

   Qi $\leftarrow C_1, C_2, \ldots, C_m, P_1, P_2, \ldots, P_n$

   where $n \geq 0$ and $m \geq 0$.

The constraint elements $C_1, C_2, \ldots, C_n$ of the above program P all lie in the same stratum. Let $\geq$ denote a reflexive relation. The strict relation (>) A > B is defined as A $\geq$ B and A $\neq$ B.

Definition 3 *In Horn-clause rules in 2LP, the relation $A \geq B$ means that A refers to B such that either $\{A \leftarrow B\}$ or there exists some element C such that $\{A \leftarrow C, C \leftarrow B\}$ in the given program.*

Definition 4 *The atomic element A is called minimal if there is no element B such that $B < A$ in the Knowledge Base. It is called a minimal stratum if the stratum only consists of minimal elements.*

The stratification of Horn-clause propositional logic in 2LP is a reflexive and transitive relation, $\leq$, on the elements of Horn-clauses which satisfies the following:

1. For all Horn-clause rules, Q $\leftarrow C_1, C_2, \ldots, C_n, P_1, P_2, \ldots, P_n$., We have the relation Ci $\leq$ Q and Pi $\leq$ Q.
2. All the constraint elements are in the minimal stratum.

The two-input nodes in Rete algorithm are used to save the information about the working memories so that the interpreter can avoid the repetitive LHS conditions matching and computing of the consistent bindings against the working memories between cycles. Therefore, the two-input nodes which have the left and right memory

are initially empty. Moreover, the memories of the two-input nodes are gradually filled with the information about the working memories as the match begins with the initial working memories and the working memories may be created depending on the conflict resolutions if the conditions are satisfied. Some elements of each stratum are created as the recognize-act cycles in Rete algorithm. However, every element of each stratum is not created in OPS5 anyway, even if we are dealing with the stratified knowledge base, since only the actions of the most relevant rules are selected by the conflict resolution strategy.

In APN of the 2LP system, the joins of two-input nodes are much simpler in computation and do not require the some amount of potential memory space for them since the consistent bindings of left and right memories do not exist. The left and right memory of two-input nodes in APN can be represented as a bit of each since the condition elements of rules consist of only the name of class. The two-input nodes of APN are set to be initially true and the procedure tables have the information during compilation as if every condition on the rules were satisfied. As the inconsistencies are found, some rules are not available for selection as determined by propagating the inconsistencies through the network. However, it is not logically necessary to fill the memories of two-input nodes in the network during the compilation time; but it is done to fill the memories for efficiency reasons. We can start with the empty memories as in the Rete algorithm, but we have to propagate all the elements of minimal stratum during run time. Even if we start with the APN approach, all the two-input nodes could become empty if every element of minimal stratum turns out to be inconsistent. The following proposition holds in this context. This is a very important property to support the truth maintenance mechanism of APN for propagating inconsistency and to undo these propagations to take care of the logic

programming style of backtracking.

Proposition 1 *The APN with all the provided elements of the minimal stratum as a set of assertions and executing the APN algorithm generates all the elements of each stratum by performing the propagation-update cycle.*

Proof: If all the elements of the minimal stratum are provided initially for propagation, it means in APN that all the two-input nodes are empty at this moment in order to make sense semantically for the APN and also that the system is backtracking all the way to the beginning for a fresh new start for the proving of the goal. Therefore, the tag of each token is marked as '+'. As the tokens are propagated, each right-memory or left-memory of the two-input nodes is set to true. The join operations are performed by the interpreter with the following instructions.

if (TL exclusive-or TR) and (left-memory and right-memory) then propagation-success else propagation-fail;

where TL is the contents of left-memory before setting it and TR is the contents of right-memory before setting it.

The condition in the above statement, (TL exclusive-or TR), is to prevent producing the multiple copies of the same atom. Therefore, this condition can be ignored since we are only considering the elements in each stratum. Suppose that $T_0, T_1, T_2, \ldots, T_n$ represent all the elements of each stratum respectively. Then, T0 is provided by the initial assumption. Moreover, $T_0, T_1, T_2, \ldots, T_n$ are defined as the following way.

T_1 = (Some atom Q: there exist some Horn clause rules such that $(P_1, P_2, \ldots, P_n) \rightarrow$ Q and $P_1, P_2, \ldots, P_n \in T_0$),

T_2 = (Some atom R : there exist some Horn clause rules such that $(Q_1, Q_2, \ldots, Q_n) \rightarrow$ R and $Q_1, Q_2, Q_3, \ldots, Q_n \in T_0, T_1$),

..., and

T_n = (Some atom L: there exist some Horn clause rules such that $(R_1, R_2, \ldots, R_n \rightarrow$ L and $R_1, R_2, \ldots, R_n \in T_0, T_1, \ldots, T_{n-1}$).

Since every token with tag = '+' is propagated through APN sequentially, all tokens <+ P1>, <+ P2>, ..., <+ Pn> are generated and set their memories. The join operations here in binary discrimination network which are

(And ....(And (And P1 P2) P3) ...Pn)

are performed successfully for each Horn clause since $P_1, P_2, \ldots, P_n$ are provided. So, T1 is generated. If there are some atoms which are not used for generating any of T_1, they are saved in the network for the subsequent cycles. Suppose that T_i, i < n is produced. Then, consider T_{i+1}, i+1 $\leq$ n. To produce T_{i+1}, the join operations

(And ...(And (And A1 A2) A3) ... An)

are performed where $A_1, A_2, \ldots, A_n \in T_1, T_2, \ldots, T_i$ and all A_i, 0 $\leq$ i $\leq$ n, where the AND nodes here are the nodes which were not performed successfully in the previous cycle for producing $T_0, T_1, T_2, \ldots, T_i$, since the left or right-memory of the AND node was not presented. To produce the T_{i+1} correctly the T_i, $T_{i-1}, \ldots, T_1$ should have been produced correctly and been saved in the network. But, by the induction hypothesis, Ti is produced in the cycle i which means that $T_{i-1}, T_{i-2}, \ldots, T_1$ have been produced correctly for any KB. Since $A_1, A_2, \ldots, A_n$ are some of the elements of T_j, 0 $\leq$ j $\leq$ i, and it is set to be true and saved in the network, the join

operations are performed successively. Therefore,T_{i+1} is also produced. Since we are dealing with the finite set of rules with no loops in them, there should be some $T_{n+1} = \emptyset$. ‡

□

The stratification of the APN algorithm with partially supplied minimal elements as the set of assertion produces at least the subset of elements of some stratum as the system executing the propagation-upate cycle. Consider that the initial given facts are the subset of the minimal stratum, then as it performs propagation-update cycles, the subset of each stratum is produced and it goes to the higher stratum to the certain point or it would produced all elements of each stratum to the top such that the KB3 = {A → C, B → C, C → D} as S1 = {A, B} as a minimal stratum, S2 = {C} and S3 = {D}, but with only the minimal element {A} can produce S2 = {C} and S3 = {D}.

5.3.2 The Stratified Table

In our current implementation of APN, the Rete-based algorithm is used for the compilation of the 2LP program as described in the above section. However, there is another possible implementation idea which can save compilation time. There may be some rules which are not stratified in a program. We call the stratified rules the *stratified kernel*. By using the stratification algorithm which is provided in a Lisp program in Appendix B, the *stratified kernel* can be extracted from the stratification algorithm.

As the stratification algorithm finds the *stratified kernal*, the system can build a stratified table which is analogous to the AND operation truth table in switching theory. Each element of stratum, $Tn, n > 0$, has a stratified table except the minimal

T0	*T1*
C3	B
0	0
1	1

T0, T1			*T2*
C1	C2	B	D
1	X	1	1
1	1	X	1
0	X	X	0
X	X	0	0
X	0	X	0

T0,T1,T2	*T3*
D	F
0	0
1	1

Table 5.2: Stratified Table for KB

stratum, T0, where the stratified table $Tn, n > 0$, provides a listing of every possible combination of the variables from $T_0, T_1, \ldots, T_{n-1}$ and lists the resulting AND operations for each combination. For instance, let us consider the following knowledge base:

KB3 = {C3 → B. C1, B → D. C1, C2 → D. D → F.}

With the given KB3, the following stratified tables are generated in Table 5.2, where the minimal element is {C1,C2,C3}.

The top-down reasoning as in Prolog with respect to the stratification begins with the stratum to which the query belongs. To match the query with the head of the rule is actually equivalent to finding the stratum which has the atomic elements. To derive the query logically in Prolog, the system should traversed down to the lower stratum until the stratum it belongs to is finally traversed down to the minimal stratum. Since each stratum has the set of atomic elements and in it mostly contains

the same elements, the system has to choose one of them which leads to the different branches of proving procedures. If the system cannot reach to the minimal stratum with that branch or the element of minimal stratum is not provided as a fact, it has to backtrack and choose the same element but with a different branch for derving it. The cost of backtracking depends upon the number of stratum it has traversed down and also the number of the atomic elements associated with the head. Those backtracking procedures are usually quite expensive and unavoidable.

With Anticipatory Pruning Network (APN), the backtracking is minimized in the 2LP system by propagating the negative tokens through the compiled networks and eliminating the domain of each stratum. There are only two explicit strata in APN. One is the stratum which consists of the set of constraints and all the elements in $(n-1)$ stratum. The other one is the procedure table associated with each atomic element. The procedure table is updated during the selection phase. The strata among the procedures are implicitly maintained and communicated by the compiled network.

As the 2LP system reasons in a Prolog-like top-down way, it traverses to the lower level strata and finally gets to the minimal stratum. If the constraint, the atomic element in the minimal stratum which the system treats, is inconsistent with the current set of the constraint environment, then the APN propagates the negative token through the network, which means that the negative token propagates to the one level higher stratum than the minimal stratum. If the condition is met, i.e., a particular atomic element disappears from the stratum, the negative token propagates to the upper stratum. When the backtracking occurs, we undo what we have all done by propagating the positive tokens. The result of this propagation in fact is inserting the deleted atomic elements from each stratum back into each stratum.

5.3.3 Maintenance of the dynamic stratification

In the context of the interactive knowledge base maintenance system, the APN can only process atomic assertions which have already appeared in the network. The atomic assertions always preserve the stratification since they cannot generate a new loop among the condition elements. The APN can provide the logic interpreter with the dynamic stratification environment. The static and dynamic stratification is introduced in [Lassez, McAloon, Port] along with the concept of interactively building knowledge base.

Definition 5 *In the interactive case, the stratification is called static if the initial stratum is fixed and assertions can be made only on the lowest stratum. It is called dynamic if the initial stratum can be changed as a result of assertions.*

Proposition 2 *The APN algorithm can virtually maintain the concept of dynamic stratification by sharing the procedure table with the logic interpreter.*

Proof idea As the system dynamically generates the negative tokens, the assertion to the APN is made. Those assertions based on inconsistency checking on constraints can change the strata of the initial knowledge base. The APN behaves like the dynamic stratification by maintaining the list of available rules and the counter which represents the number of search branches for proving an atom from the goal list. The APN is considered as the dynamic stratification since APN can implicitly increase the number of strata by propagating the positive tokens and decrease the number of stratum by propagating the negative tokens through the networks.

For example, consider the following knowledge base.

KB4 = { (r1 (C3) → B) (r2 (C1 & B) → D) (r3 (C1 & C2) → D) (r4 (D) → F)}

where the minimal elements = {C1,C2,C3} and its three strata are KB41 = {r1}, KB42 = {r2, r3}, and KB43 = {r4}.

The procedure tables maintained in APN for the above knowledge base are of the form (B 1 (r1)), (D 2 (r2 r3)) and (F 1 (r4)). Moreover, these procedure tables are shared with the logic interpreter. As the logic interpreter finds the inconsistent constraint C1 with the current environment, the assertion <- C1> will be made by propagating the negative token through APN. As a result, the updated procedure tables are (B 1 (r1)), (D 0) and (F 0) where it has the single stratum KB41 = {r1} which is the rule to which the logic interpreter can refer for solving the goal list. If backtracking occurs at this point, then < + C1> is propagated and the knowledge base has three strata as before.

5.4 Efficiency considerations of APN

As described at the beginning of this chapter, there are two components of computation in APN which are consistency checking and the forward reasoning based on that. The major computing cost is constraint consistency checking with the current environment. Since every time the current environment is updated, the consistency checking with the rest of the constraints (QC) should be performed. It may not be efficient in small problems because of the overhead in the Simplex Method if we do not compute the consistency checking in parallel.

Without the parallelism, the following new implementation can be considered for the efficient implementation.

1. The disjunctive constraints can be detected at compilation time. If one of the constraints among the disjunctive constraints is enforced, the rest of the

disjunctive constraints are inconsistent.

2. The system can activate the consistency checking after the size of the *quick list* (constraints which never been used) reduced to some level.

3. The different ways of choosing the relevant constraints is the another possibility.

In considering fast propagation of inconsistency, the Stratified Table described in the section 5.3.2 is a good candidate to check. A similar research [Colomb, Chung] has been done for the fast implementation of the propositional expert systems. They showed that, with the special hardware support, the implementation based on the Stratified Table can be more efficient than the Rete-based implementation. The special hardware here is an inexpensive bit-serial content-addressable memory. It is our intuition that the real speed-up by applying the Stratified Table can be very limited unless we are dealing with the special hardware. However, we found that the compilation into the Rete-based network is more computationally intensive than the transformation of the rules into the Stratified Table in this case.

Chapter 6

Measurement on APN : Simulation result and analysis

This chapter presents the effectiveness of APN in 2LP system. Some of the issue in constraint solving are also discussed. The first section provides benchmark results in solving some puzzles which are traditionally considered as integer programming problems.

The second provides the benchmark results for some linear programming problems which can have integer or rational variables in the problem domains. In this benchmark, the number of node visited is counted cumulatively during the tree search process. The number of constraints checks is also counted cumulatively. We counted the number of constraints appearing in the right hand side of rule and added up cumulatively as the rule is rewritten. The consistency checking in APN is counted as one constraint check for each node visited. However, we regarded that the deterministic rewriting occurs as part of APN activity at a given node.

The No APN in this benchmark is the case which the APN is not active. Therefore, the basic search strategy is the same as the Prolog search strategy. In the the Full APN, we check the consistency with all the bottom-most leaves which have not been explored yet. The Partial APN is to check the consistency with the constraints set

Queens	*No APN*		*Partial APN*		*Full APN*	
	Node	Constraint	Node	Constraint	Node	Constraint
4-Queens	334	266	154	230	94	157
5-Queens	143	113	82	123	60	95
6-Queens	3,467	2,879	1,480	2,178	832	1,374
7-Queens	499	413	247	367	163	261
8-Queens	23,359	19,904	8,959	13,097	4,916	8,054

Table 6.1: N_Queens Problem

that can be derivable from the left most branch of search space. Z* is the obtained value for the given optimization function.

6.1 Puzzles

6.1.1 N-Queens Problems

Problem description N_Queens problems deal with placing N_Queens at non-attacking positions in N * N chessboard. Queens can attack each other if they are placed in a same columns, rows or the same diagonal.

Purpose Constraints appearing in N_Queens problem are all static and all relations appearing in constraints are disequations. Moreover, the variables are in finite discrete domains. In considering this particular situation, the purpose is to observe the characteristics of disequality constraints and the effectiveness of APN.

Benchmark result Table 6.1.

Remark The disequality constraints in Prolog are treated as the negation as failure. To be more specific, the disequation in Prolog can be defined as the following if we use the predicate "not_equal" instead of $\neq$.

```
not_equal(X,Y) :- X = Y, !, fail; true.
```

The consequence of this is that the variables, here X and Y, are grounded at the time of evaluation, which is quite primitive in considering the declarative power of constraint expression.

In CLP(R), this situation is improved by providing the delay mechanisms so that the variables are not necessarily grounded at the time of evaluation of constraints. However, the basic principle in CLP(R) is still the negation as failure.

In our current implementation of 2LP system, the disequations, $X \neq Y$, are converted into "$X \geq Y + 1$ or $Y \geq X + 1$". The delaying mechanism can be natural here and the pruning in the tree search space is also possible to some extent using APN. However, one disequative constraint becomes two inequality constraints. Moreover, the cost of resolving the logical operators "or" is not ignorable if there are many disequality constraints in the problem. This is actually converting static constraints into dynamic constraints in the CSP sense. Although we can simulate the Forward Checking in CHIP by implementing APN, it is impossible to do it in this case since we cannot load all constraints at once because of "or" operations.

On the other hand, in CSP, it is very natural to load all the constraints before the system explores the domain of all tree search space. With CSP, the system cannot only test disequality constraints, but it can also prune the search space *a priori*. In fact, N_Queens problem is one of the examples which is perfect for CSP. Different representations of N_Queens problem using CSP is explored in [Nadel 1].

In conclusion, the disequality constraints which cannot prune the search space effectively in the Simplex based method are pruned *a priori* using CSP very efficiently if we are dealing with discrete domains. Therefore, in discrete domains, the consistency checking which is dealing with disequality constraints in APN can be expected to perform efficiently using the concept of CSP in 2LP system.

	No APN	*Partial APN*	*Full APN*
Node	1,641	144	4
Constraint	1,509	243	12

Table 6.2: SEND + MORE = MONEY problem

6.1.2 Cryptarithmetic problems

Problem description "SEND + MORE = MONEY" problem is assigning different digit from {0, 1, 2, ..., 9} to each letter which satisfies the equation.

```
  S E N D
+ M O R E
- - - - -
M O N E Y.
```

Similarly, "GERALD + DONALD = ROBERT" problem can be also defined, which satisfies the equation.

```
  G E R A L D
+ D O N A L D
- - - - - - -
  R O B E R T
```

Purpose This example is to investigate the effectiveness of APN in some problems in which the linear equality and inequality constraints exist at the same time.

Benchmark result Table 6.2, Table 6.3.

Remark In this problem, as shown in table 6.2, the number of nodes visited and the number of constraint checks are reduced drastically. Moreover, "SEND + MORE = MONEY" problem runs much faster than execution without APN in our

	No APN	*Partial APN*	*Full APN*
Node	11,317	1,203	438
Constraint	10,472	1,798	739

Table 6.3: GERALD + DONALD = ROBERT problem

current version of implementation even without introducing parallelism in checking consistency. In this kind of problem, the APN can be very effective since 1) there are about 10 branching factors in each domain, and 2) the constraints appearing in this problem (equality and disequality) are strong enough to find many of the inconsistencies for APN.

6.1.3 Tennis puzzle

Problem description This problem is drawn from [Lauriere]. The problem is to find out who served in the first tennis game in the following conditions;

1. Frank and George play tennis.
2. Frank beats George 6 games to 3.
3. In 4 games the server loses.

In this problem, it is asked to find all possible ways in which the situation described in the problem could occur.

Benchmark result Table 6.4

Remark The variables appearing in this puzzle are all boolean. In this example, we tried to find all the solutions. The logical atoms which represent constraints are somewhat more shared here than in any other problems in these benchmarks. Therefore, once we find the inconsistency of atoms, propagating inconsistency is very effective for deciding deterministic goals and detecting the early failure.

	Z^*	6	6	6	6	6	6	6	6	6
NO APN	Node	199	412	881	1,088	1,301	1,407	1,586	1,806	2,012
	Constraint	286	592	1,286	1,584	1,890	2,041	2,306	2,620	2,918
Partial APN	Node	50	109	240	300	365	395	447	516	599
	Constraint	125	261	567	702	847	917	1,042	1,190	1,358
Full APN	Node	13	20	40	49	56	61	67	76	82
	Constraint	65	114	244	299	348	377	423	478	524

	Z^*	6	6	6	6	6	6	6	6
NO APN	Node	2,485	3,087	3,293	3,504	3,717	3,823	4,002	4,222
	Constraint	3,616	4,517	4,815	5,117	5,423	5,574	5,839	6,153
Partial APN	Node	760	964	1,049	1,118	1,184	1,211	1,253	1,322
	Constraint	1,708	2,161	2,331	2,475	2,617	2,679	2,786	2,933
Full APN	Node	104	132	138	149	156	161	167	176
	Constraint	660	842	888	949	998	1,027	1,073	1,128

	Z^*	6	6	6	6	6	6	6
NO APN	Node	4,428	4,534	4,705	4,890	5,114	5,320	5,530
	Constraint	6,451	6,602	6,859	7,128	7,446	7,744	8,046
Partial APN	Node	1,400	1,438	1,494	1,564	1,633	1,699	1,774
	Constraint	3,092	3,169	3,293	3,439	3,585	3,724	3,875
Full APN	Node	182	189	195	201	212	218	224
	Constraint	1,174	1,209	1,255	1,301	1,362	1,408	1,454

Table 6.4: Tennis Puzzle

6.2 Linear programming problems

All the problems described in the above section can be viewed as the integer programming problems. However, many problems in linear programming are not pure integer programming problems. Some examples of the mixed linear integer programming problems are drawn from [Williams] for evaluating the effectiveness of APN in 2LP system. These mixed integer linear programming problems are not amenable to the consistency technique in CHIP because of the presense of continuous variables.

6.2.1 Blending problem in food manufacture

Problem description A food is produced by refining nonvegetable and vegetable oils and blending them together. There are two vegetable oils, i.e., Veg1 and Veg2; and three nonvegetable oils, i.e., Oil1, Oil2 and Oil3, available for processing. Assume that the price of oils is predictable for each month and there is no loss of weight in the refining process. There is also a restriction on hardness of the final products which should range between 3 and 6. The hardness blends linearly in the final products and the hardness of each oils is known initially. The detailed description of the problem is explained in [Williams].

The problem is to maximize the profit by deciding what to buy and manufacture in each month for a six month period of time with the following additional constraints.

1. Mixing more than three oils in any given month is not allowed.

2. If an oil is used, at least 20 tons must be used.

3. Oil3 must be used if one of the vegetable oils is used in a month.

For simplicity, the two month problem is first implemented in this example. The

Z^*	*18,953.7*	*21,251.9*	*24,276.9*	*26,575*	*29,975*	*Final*
Node	12	17	22	41	54	66
Constraint	31	36	43	70	89	111

Table 6.5: Two months problem with NO APN

Z^*	*18,953.7*	*21,251.9*	*24,276.9*	*26,575*	*29,975*	*Final*
Node	12	17	22	37	49	58
Constraint	42	50	60	95	121	143

Table 6.6: Two month problem with Partial APN

six month problem is also implemented in order to observe the effectiveness of APN in the larger search space.

Purpose To deal with the optimization function, it is necessary to visit the tree search space exhaustively. This benchmark is performed to evaluate the effectiveness of APN in this situation.

Benchmark result

Two month problem in Table 6.5, Table 6.6, Table 6.7.

Six month problem in Table 6.8.

Remark In this problem, it is impossible to figure out whether the variables are integers or rational numbers before we actually solve them. In a blending problem, it is very reasonable to mix 1 1/2 gallons of vegetable oil with 2 1/3 gallons of non-vegetable oil to maximize the profit. Therefore, the concept of CSP cannot be applied to this kind of linear programming problem. However, with APN, it does not

Z^*	*18,953.7*	*21,251.9*	*24,276.9*	*26,575*	*29,975*	*Final*
Node	12	16	19	29	38	42
Constraint	51	58	66	91	115	126

Table 6.7: Two months problem with Full APN

	Z^*	94,879.6	100,227.8	100,927.8	101,089.3	101,484.3
NO APN	Node	32	39	41	53	64
	Constraint	91	98	101	116	131
Partial APN	Node	32	39	41	51	62
	Constraint	122	132	136	157	180
Full APN	Node	32	38	39	47	52
	Constraint	151	160	163	182	197

	Z^*	106,832.4	107,131.5	107,907.4	111,075	111,599.1
NO APN	Node	64	71	73	140	163
	Constraint	131	138	141	242	275
Partial APN	Node	69	71	109	121	123
	Constraint	190	194	283	308	312
Full APN	Node	54	55	89	100	101
	Constraint	203	206	299	325	328

	Z^*	112,290.7	112,691.7	112,691.7	112,778.7	112778.7
NO APN	Node	173	187	461	537	2,152
	Constraint	291	313	727	838	3,224
Partial APN	Node	129	140	256	296	840
	Constraint	329	358	626	711	1,935
Full APN	Node	107	117	232	280	722
	Constraint	346	377	680	799	1,924

	Z^*	112,778.7	Final
NO APN	Node	9,249	11,863
	Constraint	13,475	17,172
Partial APN	Node	3,044	3,991
	Constraint	6,747	8,770
Full APN	Node	2,091	2,632
	Constraint	5,344	6,693

Table 6.8: Six months blending problem

matter whether the domain is discrete or continuous, since we can use Simplex based consistency checking. We are required to use either none or more than 20 tons of an oil. This kind of "or" constraint can be expressed naturally by means of logic in the 2LP system.

As the searching is performed, the value of newly finds objective function is added into linear system as an another constraint. The efore, the system found more inconsistency as the more nodes are visited during the search. In the two months blending problem, the number of node visited is not reduced drastically even with the Full APN. Moreover, the Full APN needs more constraint checks than the constraint checks with No APN in the two months blending problem. However, in the six months blending problem, the number of nodes visited and the number of constraint checks are reduced drastically as shown in the table 6.8.

6.2.2 Mining problem

Problem description A mining company is considering four different locations for continuous operations. The company can operate at most three mines out of four mines. The company should pay royalties associated with each mine for keeping mines open. However, they can also close down mines permanently so that they do not need to pay royalties.

The mining company should also consider the other factors as the following;

1. The quality of ore which can be obtained from each mine.

2. The upper limit in the amount of ore which can be extracted from each mine.

3. The quality of blended ore which is obtained by mixing ores from the mines.

4. The revenue and expenditure for future years should be discounted at a rate of 10 percent per year.

With the above constraints given, the mining company wants to maximize profit for the next five years.

Purpose Although the number of nodes visited is reduced as the searching is performed with APN, the number of constraint checks with APN is still larger than that of the problem without APN in the two month blending problem. However, in the six month blending problem, the number of constraint checks and nodes visited with the APN is drastically reduced. To observe the effectiveness of APN with another larger searching problem, this benchmark is performed.

Benchmark result Table 6.5.

Remark The mining problem here is somewhat similar to the blending problem which is described at the above section. It is another blending problem which is dealing with ores instead of oils. However, the size of the search space in this problem is much larger than that of the two month blending problem. As shown in the table 5.5, the number of constraint checks and nodes visited are drastically reduced as the searching is progressing. APN in the 2LP system is especially useful if we want to find the existence of a solution with some lower bound on the optimizing function. For instance, if we want to find out whether there exists a solution with the lower bound 150 in this problem, APN can be very useful. According to my benchmark, the 2LP system without APN fails to find the solution after 17,542 nodes are visited and 24,670 constraints checks are performed. Whereas with APN, the system fails find the solution only after 5,413 nodes are visited and 12,673 constraint checks are performed. Intuitively, APN is working well in this case since the system can have

	Z^*	23.5	54.0	61.7	79.5	90.6	99.1	99.8	110.9	111.6
NO APN	Node	31	43	53	61	67	82	91	97	129
	Constraint	58	75	89	100	108	130	144	152	200
Partial APN	Node	28	38	46	52	56	65	71	75	90
	Constraint	79	100	117	130	139	164	181	190	231
Full APN	Node	28	38	46	52	56	63	69	72	81
	Constraint	97	121	140	154	163	183	200	208	235

	Z^*	122.8	129.1	129.1	131.3	131.8	131.8	136.6
NO APN	Node	135	153	585	2,242	3,250	5,307	12,434
	Constraint	208	235	877	3,327	4,822	7,843	18,154
Partial APN	Node	94	99	252	795	1,136	1,840	4,079
	Constraint	240	254	666	2,110	3,015	4,845	10,435
Full APN	Node	84	89	226	714	978	1,598	3,564
	Constraint	243	258	636	1,968	2,686	4,347	9,289

	Z^*	136.6	137.6	138.9	139.4	139.4
NO APN	Node	12,890	14,159	14,657	15,563	17,367
	Constraint	18,832	20,706	21,446	22,788	25,431
Partial APN	Node	4,238	4,653	4,811	5,108	5,715
	Constraint	10,870	11,972	12,403	13,195	14,782
Full APN	Node	3,705	4,039	4,173	4,400	4,912
	Constraint	9,683	10,588	10,959	11,576	12,942

	Z^*	141.2	Final
NO APN	Node	19,190	33,598
	Constraint	28,116	48,563
Partial APN	Node	6,252	10,736
	Constraint	16,205	26,923
Full APN	Node	5,379	9,753
	Constraint	14,188	24,722

Table 6.9: Mining Problem

great deal of inconsistency to find because of the bigger lower bound although the system needs to implicitly enumerate nodes in the problem space. In this kind of situation, APN can play an important role.

6.3 Conclusions

The APN in 2LP system can be very effective in solving mixed linear integer programming problems, especially if we are dealing with very large problems. Overall, regardless of problem domains, APN is a very powerful mechanism to prune the tree search space as shown in the examples in this chapter. However, the overhead in testing the consistency should be considered carefully. To minimize the overhead, there are three different methods to consider. One is checking the consistency in parallel. The second is that we can apply the concept of CSP for testing consistency in APN if it is known that the variables in the problems are finite discrete domains. The third is applying the CSP and parallelism at the same time which are described in the section 8.1.3.

Chapter 7

Related work

7.1 Consistency technique in CHIP

CHIP is designed to integrate the tree search strategy in Prolog with the efficiency of CSP. One of the disadvantage of Prolog-like search strategy is the passive use of constraints. In other words, the constraints in Prolog are just used for testing whether the current instantiation of a variable satisfies the constraints. Moreover, the constraints cannot be tested if any variables in the constraints have not been instantiated.

In the CHIP system, the designers remedied these problems by embedding the CSP concept in Prolog. They used the constraints in an active way. Conceptually, this means that once the constraints are loaded into the system with the given domain for the variables and one of the variables are instantiated, we can rule out some domains in the variables. For instance, in 4-Queens problem, once Q1 is bound to 1 we can rule out the possibility of that domain for other Queens, such as Q2 = 1, Q2 = 2, Q3 = 1, Q3 = 3, Q4 = 1 and Q4 = 4. By using CSP, this is implemented very efficiently by manipulating the binary relational matrices in CHIP.

However, if we want to implement this directly in the constraint solver in a linear programming problem, then there are some limitations in the consistency technique

in CHIP. The limitations are probably inherent to the CSP itself. One is that CSP can only deal with the static constraints which means that all the variables and constraints appeared in system should be used. Whereas the Simplex based solver in 2LP system or CLP(R) can dynamically introduce the variables and constraints naturally. Another is that CHIP cannot address the pruning of continuous domains or implicit domains as also mentioned in section 3.3. However, many problems in linear programming are mixed integer linear programming.

CHIP system is successfully implemented with the concept of CSP into the full power of Prolog with finite discrete domains. One the other hand, APN in 2LP system is dealing with the propositional logic with constraint solver. However, the consistency technique in CHIP is implemented at the propositional level. The predicate logic in Prolog is just used to generate the constraints and domains down to the propositional level during run time. To be more specific, let us consider the following example of a CHIP program which is considered as Generalized Forward Checking for 8-Queens problem drawn from [Van Hentenryck].

```
safe([]).
safe([F|T]) :- noattack(F, T), safe(T).
noattack(X, Xs) :- noattack(X, Xs, 1).
noattack(X, [], Nb).
noattack(X, [Y|Ys], Nb) :- X =\= Y, X =\= Y - Nb, X =\= Y + Nb,
                           Nb1 is Nb + 1, noattack(X, Ys, Nb1).
labeling([]).
labeling([X|Y]) :- indomain(X), labeling(Y).
domain eight-queens(1..8).
```

```
eight-queens([X1, X2, X3, X4, X5, X6, X7, X8]) :-
                    safe([X1, X2, X3, X4, X5, X6, X7, X8]),
                    labeling([X1, X2, X3, X4, X5, X6, X7, X8]).
```

The predicate "safe" here is used to generate all the constraints among 8-Queens before starting to solve the problem. With the declaration of domain to range over 1 to 8 and the constraints, we can apply the CSP technique to 8-Queens problem. This can be quite efficient. The domain declaration in CHIP, which is analogous to the variable declaration in conventional language such as Pascal, is somewhat awkward from the conventional logic programming point of view. The predicate "indomain(X)" which is built-in in the CHIP system allows programmers to generate any values in domain X. Therefore, X in this predicate should be a ground term or a domain variable. As the "indomain" predicate enforces the variable binding, the system can prune the search space using CSP technique described in section 2.1.2. These kinds of constraints with the domain concept can be easily expressed in 2LP system which basically deals with propositional logic with the constraints.

Both APN in 2LP and consistency technique in CHIP system attempted to avoid the fatal disadvantages of Prolog style searching mechanism which are labeled as "thrashing" behavior by Bobrow and Raphael. However, they are functionally different. In APN, the system try to remove the domain of each stratum by propagating inconsistency of constraints. However, the APN in 2LP system cannot prune the search space of unstratified rules. In the consistency technique in CHIP, the system tries to remove the domains of each variable in the problem. But, the system cannot prune the search space with consistency techniques if the domains of variables in the

problem are not explicit. Checking consistency in CHIP is more efficient than that of APN. On the other hand, in APN, checking consistency is somewhat separate from the propagation of the inconsistency. This can allow us to use any consistency checking mechanism, whether CSP or Simplex based solver, depending on the problem domain.

7.2 Forward checking through meta-interpretation and transformation

One of the disadvantages of the consistency technique in CHIP is that the forward checking should be performed during run time. Another way of achieving forward checking at compilation time is investigated in [Schreye, Bruynooghe]. In this paper, abstract meta-interpretation is used to enforce the simulation of the computation flow that the transformed program would have under a forward checking control rule. The transformation of a program is done by providing two library routines which are create-domain and select-from-domain. This converts a Prolog program into another Prolog program with domain generation.

In this approach, there are some advantages over the consistency technique in CHIP which are summarized as following.

1. The forward checking can be provided at compile time.

2. The forward checking is not a new computation rule, while on the other hand, the forward checking in CHIP is a new built-in inference rule in Prolog. The forward checking in Prolog can be more portable in this approach.

3. The specification of the computation rule can easily be automated.

There are also some disadvantages with this approach. One is that this is not as efficient as the forward checking in CHIP. The other is that the transformation of a program is not an easy task.

7.3 Problem solving techniques : OR versus AI

Empirical Researches [Van Hentenryck, Carillon], [Dhar, Ranganathan] were conducted to compare the two different approaches, OR and AI, of problem solving techniques. Dhar and Ranganathan compare Integer Programming with expert systems by performing a case study on generating a course schedule. Van Hentenryck and Carillon took a case study on a warehouse location problem to compare three different problem solving strategy, i.e. Integer Programming, a problem specific program based on A* algorithm and CHIP.

Of course, there is no panacea in problem solving. Each problem solving technique has its own advantages and disadvantages for a specific problem domain. Discussing which technique has an advantage over another is a very delicate issue. For instance, Van Hentenryck and Carillon stated in their paper that "As far as convenience of programming is considered, Integer Programming turns out to be the ideal solution". It may be true in their simplified warehouse location problem. However, it is not true if we consider a course scheduling problem in [Dhar, Ranganathan].

The features of each problem solving technique can be generally summarized as following;

Integer Programming (IP) : The formulation of a problem can be mathematically very declarative in some problems as in the simplified warehouse location problem. On the other hand, the modeling of IPs which are somewhat complicated, like a course scheduling problem, is a nontrivial problem. The IP program is also

difficult to read for the complicated problems. Moreover, as Dhar and Ranganathan pointed out, it is very difficult to modify the program once some underlying situation has been changed. In terms of speed, it is unlikely to be efficient if we consider the number of variables generated and the search space explored. Applying problem specific heuristics is not feasible in many cases. Finally, using IP, it is impossible to generate the partial solution especially if we want to get a solution which is close to feasible. IP spends lots of time and ends up with nothing in this case.

Expert system : [Dhar, Ranganathan] used an expert system with a truth maintenance system (TMS) to solve a course schedule problem. In contrast to IP, this is a symbolic way of solving problems which is a typical AI approach. Their research shows that an expert system usually finds a solution between 1 and 2 hours while the run time of IP is highly unpredictable, varying from a few minutes to few days. Expert system also generates a partial solution in the form of dependency network if it cannot find a solution. This is another good feature of expert system. By using expert system, the program can be much easier to revise than using IP if the underlying assumption is changed.

CHIP : The system can be more efficient than the other approaches to problem solving techniques mentioned above. However, it cannot generate a partial solution as the expert system did if the solution does not exist. Of course, we can trace out the activity of the program. But it is not feasible if we consider that most real problems tend to have huge search space. Moreover, we cannot address the mixed integer linear programming with the consistency technique in this system.

A specialized program tailored to a problem : [Van Hentenryck, Carillon] showed that this is the most efficient problem solving technique. However, it is difficult to modify the system after design. Moreover, it can take lots of program

development time. In their case study of simplified warehouse location problem, the program consists of 2,000 lines of Pascal code and took several months of development time.

With the development of the 2LP family of languages, we are not only adding another possibility of a problem solving technique, but we also can improve some of the difficulty described previously, which is addressed in section 8.2.

Chapter 8

Future researches and Conclusion

8.1 Future research

8.1.1 Parallel implementation of consistency checking

As described in the previous chapters, consistency checking using CSP is very efficient and economical in terms of memory. However, the Simplex Method is a more general and powerful constraint solver which has more overhead in computing. Therefore, the consistency checking in the Simplex-based method, while more general, is not as efficient as that of the CSP.

To improve the speed in computing, we can check the consistency with the current environment in parallel. Consistency checking is an independent enough task to perform on several different processors. More specifically, in our implementation of the 2LP system, the consistency checking routine is defined as the following:

```
(defun consistency-checking (quick-list)
   (dolist (cj quick-list)
      (when (disallow cj) (my-pop (list cj))) ))
```

The disallow function here is supposed to check the consistency of constraints with the

current environment. If the constraint is inconsistent, the inconsistency is propagated through APN by calling the function my-pop. By using a copy of the current Simplex configuration, the disallow function can be executed in parallel, at a degree of the number of constraints elements in *quick-list*, possibly in tens or hundreds of Unix workstations which are now widely spread throughout universities and industry.

8.1.2 Parallel implementation of inconsistency propagation

[Gupta] showed that the amount of speed-up available from the parallelism in production systems is about 10-fold with 32-64 high performance processors and special hardware support. The APN in 2LP system can be viewed as a propositional expert system where the working memories in APN represents the inconsistent constraints or logical atoms with the current environment. In our current implementation of APN, a variation of the Rete algorithm is used. However, as described in section 5.3.2, there is another possibility of implementation which is here referred as the Stratified Table.

A similar research is performed in [Colomb, Chung] to implement the propositional expert system such as COLOSSUS and the Garvan ES1, thyroid assay system, in an efficient manner using the decision table. The decision table is analogous to the Stratified Table in APN. A case study is performed by them to compare the Rete based implementation with the decision table based implementation in the propositional case. Although their result shows that the decision based implementation is much faster than the Rete based implementation, their comparison basically has two problems. One is that they compared the execution of Rete based implementation on a Microvax II with the execution of the decision table on a Sun 3/160 with a special hardware (a bit-serial content-addressable memory), which is like comparing an apple

with an orange. The other is that they translated an expert system, Garvan ES1, into OPS5 to execute on a Microvax II. However, OPS5 is not designed for a purely propositional expert system. Intuitively, it is acceptable that the decision based implementation can be faster than the Rete based implementation without knowing how much faster.

The OPS5 interpreter based on the Treat algorithm [Miranker 1] was designed at Colombia University to improve the disadvantages in the Rete algorithm especially in parallel implementation of production system. The Treat algorithm dynamically performs the join operation among working memories without saving the partial consistent bindings as opposed to explicitly saving them in Rete algorithm. The disadvantages of Rete algorithm over the Treat algorithm are 1) the overhead in removing the stored working memories 2) the size of the local memories in storing partial consistent binding can be explosive 3) sharing nodes in the Rete algorithm is not an advantage in parallel processing because of contention. The motivation behind the Treat algorithm is based on the conjecture in [McDermott,Newell,Moore], to wit, "It seems highly likely that for many production systems, the retesting cost will be less than the cost of maintaining the network of sufficient tests.".

What the Rete algorithm is to the Treat algorithm in predicate calculus, the Rete based propagation network in APN is to the Stratified Table in the propositional expert system. It is currently not clear which implementation technique is more efficient in consistency propagation in APN, just like it is a controversial issue [Miranker 2], [Nayak, Gupta, Rosenbloom] if we have to justify which implementation technique is more efficient, the Rete algorithm or the Treat algorithm. For future research, it will also be interesting to figure out empirically which implementation is faster in the propositional case, the Stratified Table or the Rete-based algorithm, with the real

applications. However, in considering the research mentioned in this section, it is highly likely that implementation based on stratified table will prove more efficient than the implementation based on the propositional Rete algorithm especially if we consider the parallel implementation.

8.1.3 The integration of the Simplex-based constraint solver with CSP

The APN in 2LP class language can integrate the efficiency of the CSP with the generality of the Simplex-based constraint solver. In other words, the Simplex based constraint solver can be used as main inference engine in constraint logic programming while the CSP is used as the consistency checking mechanisms for APN in the discrete domains. More specifically, the disequality constraints which cannot be solved efficiently using the Simplex-based method is the good candidate to investigate for integrating the efficiency of CSP with the Simplex based method in the 2LP class of language. For instance, in the cryptarithmetic problems, the equality constraints can be solved by the Simplex-based method, while the consistency checking for the disequality constraints is performed by using the efficiency of CSP in the 2LP system. The problems which have only the disequality constraints such as N-Queens problem can be solved using only the CSP technique.

By integrating the Simplex Method with the concept of CSP, the 2LP class of language is expected to be a new paradigm of problem solving technique which utilize the advantages of both AI and OR technique, namely generality and specificity [Van Hentenryck, Carillon]. This work is done for simulation purposes in the 2LP system. By integrating the Stratified Table for the inconsistency propagation, which is described in the section 5.3.2, with the concept of CSP, the basic operations in

pruning search spaces are all on the binary form which is potentially very efficient and economical in terms of speed and memory. Further research for the actual implementation is now under way.

8.2 Conclusions

The study of APN in 2LP system lead us to make the following conclusions.

1. The APN introduces Forward Checking mechanism for constraints over continous and discrete domain. It also provides very deep look ahead for the Simplex-based constraint solver. This property, Forward Checking mechanisms independent of domains, is highly desirable in linear programming environments if we consider that many linear programming problems are mixed integer linear programming. We have shown that the tree search space can be pruned dramatically using the APN as described in the Chapter Six.

2. By using the APN, we can exploit the OR-parallelism, i.e., parallelism in consistency checking, in constraint optimization problems effectively in the 2LP system [McAloon Tretkoff]. As we have pointed out in the section 6.2, the APN is very useful to solve the optimization problems. Especially if we want to find a optimization solution which is a near-failure, the APN can reduce the tree search space drastically as shown in the section 6.2.

3. As described in the section 8.1.3, the APN, which is a domain independent forward checking mechanism, opens a possibility of integrating the Simplex-based constraint solver with CSP.

4. As described in section 7.3, the expert system can generate the partial solutions if the system cannot find a solution. This feature is considered as the one which any other problem solving technique cannot have naturally. The APN can address the partial solution naturally since the APN can be viewed as the propositional expert system and the current working memories in APN are the inconsistent constraints found so far with the current environment.

Appendix A

Code for the network interpreter

```
; The main routine of the network interpreter.
(defun node-interpreter (token i)
   ; fetches the node type for the next node from the memory.
   (let ((node (get-node-type i)))
      (cond
         ; Another Branch ?
         ((equal node 'fork) (fork token i))
         ; Join to two input node ?
         ((equal node 'join) (my-join token i))
         ; Terminal node for the rule ?
         ((equal node 'update) (update token i))
         ; Testing for constant element.
         ((equal node 'eq) (eq token i))
         ; And operation for two input node ?
         ((equal node 'and) (my-and token i))
         ; Else it must be compiler error.
         (t 'network-compiler-error))) )

; Save the address of the branch and pass the control to the
; next node.
(defun fork (token i)
   ; push the another node into stack for later processing.
   (push (aref *dna* i 2) *br-stack*)
   (succ token)) ; Go to the next node.

;Join to the right-memory of the two-input node.
(defun my-join (token i)
   ; Affect the right-memory if this  node meet two-input
   ; node.
   (setq *direction* 'right)
   ; Branch to the node which the merge node point to.
   (setq *ctr* (gethash (aref *dna* i 2) *indx*))
   ; interpret the node.
   (node-interpreter token *ctr*))

;Update the associated procedure table.
```

```
(defun update (token i)
   ;Update the procedure table since we get to the terminal
   ;node for a rule.
   (update-bucket token (aref *dna* i 2) (aref *dna* i 3))
   ; Get to the unprocessed nodes.
   (my-fail token))

; One-input node to test the constant.
(defun teqa (token i)
   ; Assign the constant of the one-input node.
   (let ((node (aref *dna* i 2)))
     (cond
        ; If the token has the same symbol as
        ; the one-input node, ...
        ((equal (car token) node) (succ token))
        ; If it is a negative token and it match, ...
        ((and (equal (car token) '-) (equal (cadr token) node))
                  (succ token))
        ; Else, failing  the propagation.
        (t (my-fail token)) )))

; Two-input node to perform the join.  If the  token  reaches
; this node, the node need to update accordingly depending on
; the  tag.  The AND operation performed here for the left and
; right-memory.
(defun my-and (token i)
    ; Before we set anything into memory, save them.
    (let   ((t-left (aref *dna* i 2))
            (t-right (aref *dna* i 3)))
        (cond
          ; If it is the negative token which means the
          ; deletion, ..
          ((equal (car token) '-)
              (cond
                ; Check if the node is from the left ...
                ((equal *direction* 'left)
                    ; If it is, delete the left memory.
                     (setf (aref *dna* i 2) nil))
                ; If it is from the right node, ...
                (t (setq *direction* 'left)
                   ; Delete the right memory.
                   (setf (aref *dna* i 3) nil)))
              (cond
                ; If the previous memories, both left and right,
                ; are set to true, this deletion should affect to
                ; the next nodes.
                ((and t-left t-right) (succ token))
                (t (my-fail token))))
                ; If the token is positive, then ...
                ; If the node is from the left parent, then ...
          (t (cond ((equal *direction* 'left)
                       ; Set it to true.
```

```
                        (setf (aref *dna* i 2) t))
                        ; If the node is from the right parent, ...
                    (t (setq *direction* 'left)
                        (setq (aref *dna* i 3) t)))
              ; If the and operation is successful because of the new
              ; token, then the token propagation will be continue.
              (cond ((and (excl-or t-left t-right)
                          (and (aref *dna* i 2) (aref *dna* i 3)))
                          (succ token))
                    ; Otherwise, it fails.
                    (t (my-fail token))))) ))

; Increment the node counter to reach the next node.
(defun succ (token)
   (cond
      (setq *ctr* (+ *ctr* 1))
      (node-interpreter token *ctr*))

; If the  failure occurs during the matching, then get a  node
; from the stack which has the unprocessed nodes.
(defun my-fail (token)
   (cond
     ; If the stack is empty, halt matching.
     ((null *br-stack*) nil)
     ; Get the unprocessed node, and continue ...
     (t (setq *ctr* (gethash (pop *br-stack*) *indx*))
        (node-interpreter token *ctr*)) ))

; In this node, the operation for updating the procedure tables
; will be performed and the selection for the further propagation
; will  also be performed.
(defun update-bucket (token r-num head)
   ; Assign the procedure table to the corresponding head  for
   ; the terminal node.
   (let* ((act (gethash head procedure-table))
          (cnt (car act)))
      (cond
         ; If the token is negative, we have to decrement
         ; the counter.
         ((equal (car token) '-)
            (cond
               ((= cnt 2) (setq *det* (cons head *det*)))
               ; If the counter becomes zero, save the head for the
               ; further propagation.
               ((= cnt 1) (setq *selection-set*
                                (cons (list '- head) *selection-set*))))
                          (setq *dead* (cons head *dead*))))
            ; If the counter is greater than 1, decrement the counter
            ; and remove the corresponding rule.
            (cond
               ((> cnt 0)
                 (setf (gethash head procedure-table)
```

```
                        (remove r-num act)))))
            ; If the token is positive, ...
            (t (cond
               ; If the counter is zero before, save the head for
               ; the further propagation into the stack.
                 ((= cnt 0)
                    (setq *selection-set*
                          (cons (list head) *selection-set*))
                    (setq *det* (cons head *det*))))
               ; If not, check if the counter for the head is not
               ; exceeding the total number of the head.  In this way
               ; we can prevent the loop.
               (cond ((< cnt (gethash head *c-max*))
                         ; Add the rule and increment the counter.
                         (setf (gethash head procedure-table)
                               (right-insert r-num act)))))
       ))))

(defun right-insert (i llist)
    (cond ((endp llist) (list i))
          ((< i (first llist)) (cons i llist))
          (T (cons (first llist) (right-insert i (rest llist))))))
```

Appendix B

Code for the stratification

```
;Stratification algorithm
(defun build-st-table ()
   (copy-pro-table)
   (do* ((rule-no 1 (+ rule-no 1))
 (test (gethash rule-no *head*) (gethash rule-no *head*)))
     ;loop termination condition.
     ((null test) (setq *no-of-rule* rule-no))
   (let*
     ((goal (right-hand-side-goal-list
            (gethash rule-no right-hand-side-table)))
      (procedures (gethash test *pro-table*)))
     (cond
       ((null goal)
          (setf (gethash test *pro-table*) (remove rule-no procedures))
   (setf *mark-rule* (nconc*mark-rule* (list rule-no))))
       (t (setf (gethash rule-no *strat-table*) goal)
          (setq *unmark-rule* (nconc *unmark-rule* (list rule-no)))))
      )))

(defun copy-pro-table ()
  (dolist (procedure *all-head*)
    (setf (gethash procedure *pro-table*)
          (gethash procedure procedure-table)) ))

(defun new-mark (head unmarks)
   (dolist (mark unmarks) (mark-arule head mark)))

(defun mark-arule (head unmark)
   (let ((unmark-body (gethash unmark *strat-table*))
         (unmark-head (gethash unmark *head*)))
  (cond
    ((member head unmark-body)
       (let ((unmarked (remove head unmark-body)))
         (cond
           ((null unmarked)
             (cond
               ((null (gethash unmark-head *pro-table*))
```

```
                    (setq *unmark-rule*
                          (remove unmark *unmark-rule*))
                    (setq *mark-rule* (nconc *mark-rule*  (list unmark)))
                    (setq *new-mark* (nconc  *new-mark*  (list unmark))))
                  (t (setq *unmark-rule*
                           (remove unmark *unmark-rule*))
                     (setq *mark-rule* (nconc *mark-rule* (list unmark)))
                     (setf (gethash unmark-head *pro-table*
                           (remove unmark
                                   (gethash unmark-head *pro-table*))))
                     (cond
                       ((null (gethash unmark-head *pro-table*))
                           (setq *new-mark*
                                 (nconc  *new-mark* (list unmark))))
                       ) ))))
          (setf (gethash unmark *strat-table*) unmarked)) ))))

(defun stratification ()
   (compiler-initial-value)
   (build-st-table)
   (keep-marking *mark-rule* *unmark-rule*))

(defun keep-marking (mark-rules unmark-rules)
   (marking mark-rules unmark-rules)
   (let ((new *new-mark*))
       (setq *new-mark* nil)
       (cond
          ((null new) (sort *mark-rule* #'<))
          (t (keep-marking new *unmark-rule*)))))

(defun marking (mark-rules unmark-rules)
   (mapcar #'(lambda (mark) (new-mark mark unmark-rules))
                              (list-of-head mark-rules)))

(defun list-of-head (rules)
   (do ((heads nil)
        (rules rules (cdr rules)))
    ((null rules) heads)
    (setq heads (adjoin (gethash (car rules) *head*) heads)) ))
```

Bibliography

[Aho, Ullman] Alfred V. Aho, Jeffrey D. Ullman, *Principles of compiler design*, Addison Wesley, Massachusetts, 1979.

[Apt, Blair, Walker] Krzysztof R. Apt, Howard A. Blair, Adrian Walker, Towards a theory of declarative knowledge, *Foundations of deductive databases and logic programming*, Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp 89-148.

[Borning] Alan Borning, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, *ACM Transactions on Programming Language and Systems*, Vol 3, No. 4, Oct., 1981, pp 353-387.

[Borning, Duisberg] Alan Borning, Robert Duisberg, Constraint-Based Tools for Building User Interfaces, *ACM Transactions on Graphics*, Vol 5. No. 4, October 1986, pp 345-374.

[Brownston et al] Lee Brownston, Robert Farrell, Elaine Kant, Nancy Martin, Programming Expert Systems in OPS5, An introduction to Rule-Based Programming, Addison Wesley, Massachusetts, 1986.

[Chandra Lewis Makowsky] A. Chandra, H. Lewis, and J. Makowsky, Embedded Implicational Dependencies and their Inference Problem, *STOC*, 1981,pp 342-354.

[Chvatal] Vasek Chvatal, *Linear programming*. W. H. Freeman and Company, New York, 1983.

[Cohen] Jacques Cohen, Constraint Logic Programming Languages, *Communications of the ACM*, July 1990, Vol.33 No.7, pp 52-68.

[Colmerauer 87] A. Colmerauer, Opening the Prolog III universe: a new generation of Prolog promises some powerful capabilities,*BYTE*, Aug. 1987, pp 177 - 182.

[Colmerauer 90] Alain Colmerauer, An introduction to Prolog III, *Communications of the ACM*, July 1990, Vol.33 No.7,pp 69-90.

[Colomb, Chung] Robert M. Colomb and Charles Y. C. Chung, Very fast decision table execution of propositional Expert System, AAAI-90, *Proceedings eighth national conference on artificial intelligence*, The MIT Press, 1990, pp 671-684.

[Cox, McAloon, Tretkoff] Jim Cox, Ken McAloon and Carol Tretkoff, Computational Complexity and Constraint Logic Programming Languages, Brooklyn College Computer Science Technical Report, No.90-4.

[Davis] Ernest Davis, Constraint propagation with interval labels, *Artificial Intelligence* 32, 1987,pp 281-331.

[Dechter, Pearl] Rina Dechter and Judea Pearl, Network-based Heuristics for constraint-satisfaction problems, *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, New York,1988.

[Dhar, Ranganathan] Vasant Dhar and Nicky Ranganathan, Integer programming vs. Expert Systems: An experimental comparison. *Communications of the ACM*, March 1990,pp 323-336.

[Dincbas et al 1] M. Dincbas, Pascal Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of International Conference on Fifth Generation Computing Systems*, 1988.

[Dincbas et al 2] Mehmet Dincbas, Helmut Simonis and Pascal Van Hentenryck, Solving large combinatorial problems in Logic Programming, *The journal of logic programming*, 1990:8:75-93.

[Eastman] Charles M. Eastman, Automated Space Planning, *Artificial Intelligence 4* 1973, pp 41-64.

[Forgy 79] Charles Forgy, On the efficient implementation of Production systems, *Ph. D. Thesis*, Carnegie-Mellon University, 1979.

[Forgy 82] Charles Forgy, Rete : A fast algorithm for the many pattern/many object pattern match problem,*Artificial Intelligence* 19, Sept. 1982,pp 17-37.

[Fox] Mark S. Fox, *Constraint-directed search: A case study of job-shop scheduling*, Morgan Kaufman Publishers, Inc.,Los Altos, California,1987.

[Freeman-Benson, Maloney, Borning] Bjorn N. Freeman-Benson, John Maloney and Alan Borning, An Incremental Constraint Solver, *Communications of the ACM*, Vol. 33, pp 54-63, January 1990.

[Freuder 78] Eugene C. Freuder, Synthesizing Constraint Expressions, *Commucations of the ACM*, vol. 21, pp 958-966, November 1978.

[Freuder 90] Eugene C. Freuder, Complexity of K-Tree structured constraint satisfaction problems, *Proc. Eight Nat. Conf. on Artificial Intelligence (AAAI-90)*, Boston, MA, August 1990,pp 4-9.

[Gorlick et al] Michael M. Gorlick, Carl F. Kesselman, Daniel A. Marotta and D. Stott Parker, Mockingbird: A logical methodology for testing, *The journal of logic programming*, 1990:8:95-119.

[Gupta] Anoop Gupta, *Parallelism in production systems*, Morgan Kaufman, Inc., Los Altos, California, 1988.

[Gusgen] Hans Werner Gusgen,*CONSAT: A system for constraint satisfaction*, Morgan Kaufmann Publishers, Inc, Los Altos, California,1989.

[Haralick, Elliott] Robert M. Haralick and Gordon L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence* 14, 1980,pp 263-313.

[Heintze, Michaylov, Stuckey] Nevin Heintze, Spiro Michaylov and Peter Stuckey, CLP(R) and some Electrical Engineering Problems, *Fourth IEEE Symposium on Logic Programming*, San Francisco, Aug., 1987.

[Hennessey] Wade L. Hennessey,*Common LISP*, McGraw-Hill Book Company, New York, 1989.

[Huynh, Lassez 89] Tien Huynh and Catherine Lassez, An expert decision-support system for option-based investment strategies, *IBM technical report*, March, 1989.

[Huynh, Lassez 88] Tien Huynh and Catherine Lassez, A CLP(R) Option Analysis System, *Proceedings 1988 Logic Programming Symposium*,pp 59-69.

[Jaffar Lassez 86] J. Jaffar and J. L. Lassez, Constraint Logic Programming, Monash University Technical Report 1986.

[Jaffar Lassez 87] J. Jaffar and J. L. Lassez, Constraint Logic Programming, *Proceedings of POPL*, 1987, Munich.

[Jaffar, Michaylov] J. Jaffar and S. Michaylov, Methodology and implementation of a constraint logic programming system, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, M.I.T press, 1987, pp 196 - 218.

[Jones Laaser] N. Jones and W. Laaser, Complete problems for deterministic polynomial time, *Theoretical Computer Science* 3 (1977) pp 107-117.

[Kowalski] Robert Kowalski, Algorithm = Logic + Control, *Communications of the ACM*, July 1979, Vol. 22, Number 7, pp 424-436.

[Lassez McAloon] J-L. Lassez and K. McAloon, Applications of a canonical form for generalized linear constraints, *Proceedings of the 1988 FGCS Conference*, Tokyo, pp 703-710.

[Lassez, McAloon, Port] Catherine Lassez, Ken McAloon and Graeme Port, Stratificationand Knowledge Base Management, *J. Symbolic Computation* July 1989, pp 509-522.

[Lassez, McAloon, Yap] Catherine Lassez, Ken McAloon and Roland Yap, Constraint Logic Programming and Option Trading, *IEEE Expert*, Fall, 1987, pp 42-50.

[Lauriere] Jean-Louis Lauriere, *Problem solving and artificial intelligence*, Prentice Hall,New York,1990.

[Leler] Wm Leler, *Constraint Programming Languages, Their specification and generation*, Addison-Wesley, Massachusetts,1988.

[Lloyd] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag,New York, 1984

[Mackworth] Alan K. Mackworth, Consistency in Networks of Relations, *Artificial Intelligence* 8, 1977, pp 99-118.

[Mackworth, Freuder] Alan K. Mackworth and Eugene C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Aritificial Intelligence* 25, 1985, pp 65-74.

[Maher] M. J. Maher, Logic Semantics for a Class of Committed-Choice Programs, *Proceedings of the 1987 Logic Programming Conference*, Melbourne, MIT Press, pp 858 - 876.

[Maier Warren] D. Maier and D. S. Warren, *Computing with Logic*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1988.

[McAloon Tretkoff] K. McAloon and C. Tretkoff, *2LP*: A Logic Programming and Linear Programming System, Brooklyn College Computer Science Technical Report No. 1989-21.

[McDermott,Newell,Moore] J. McDermott, A. Newell and J. Moore. *The efficiency of certain production system implementations, In Pattern-directed Inference System*, Academic Press, New York, 1978.

[Mittal et al 1986] Sanjay Mittal, Clive L. Dym and Mahesh Morjaria, PRIDE: An expert system for the design of paper handling systems, *Computer*, vol. 19, pp 102 - 114, July, 1986.

[Mittal, Falkenhainer] Sanjay Mittal and Brian Falkenhainer, Dynamic Constraint Satisfaction Problems, *AAAI-90 Proceedings*, American Association for Artificial Intelligence, The MIT Press,July 1990, pp 25-32.

[Mittal, Nadel] Sanjay Mittal and Bernard Nadel, Constraint Reasoning: Theory and Applications, Tutorial, *AAAI-90 Proceedings*, July 1990.

[Miranker 1] Daniel P. Miranker, TREAT: A better match algorithm for AI production systems, *AAAI-87 Proceedings*, American Association for Artificial Intelligence, July, 1987, pp 42-47.

[Miranker 2] Daniel P. Miranker, *Treat: A new and efficient match algorithm for AI production systems*, Morgan Kaufmann Publishers, Inc.,San Mateo, California, 1990

[Montanari, Rossi] Ugo Montanari and Francesca Rossi, Fundamental Properties of networks of constraints: A new formulation, *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag,New York, 1980

[Mohr, Henderson] Roger Mohr and Thomas C. Henderson, Arc and Path Consistency Revisited, *Artificial Intelligence* 8, 1986,pp 225-233.

[Nadel 1] Bernald A. Nadel, Representation selection for constaint satisfaction: A case study using N-Queens, *IEEE Expert*, June 1990, pp 16-23.

[Nadel 2] Bernald A. Nadel, Tree search and arc consistency in constraint satisfaction algorithms, *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, New York, 1988.

[Nadel 3] Bernald A. Nadel, The complexity of constraint satisfaction in Prolog,*AAAI-90, Proceedings of eight national conference on Artificial Intelligence*, The MIT Press, 1990, pp 33-39.

[Nayak, Gupta, Rosenbloom] Pandurang Nayak, Anoop Gupta, Paul Rosenbloom, Comparison of the Rete and Treat Production Matchers for Soar (A summary). *AAAI-88 Proceedings*, Aug., 1988, Minnesota, pp 693-698.

[Nilsson] Nils J. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag, New York, 1980.

[Rossi] Francesca Rossi, Constraint Satisfaction Problems in Logic Programming, *SIGART Newsletter*, Number 106, ACM press, New York, October 1988.

[Rothenberg] Ronald I. Rothenberg,*Linear Programming*, North-Holland, New York, 1979.

[Sakai Aiba] K. Sakai and A. Aiba, *CAL*: Theoretical Background of Constraint Logic Programming and its Applications, *Journal of Symbolic Computation*, 8 No. 6 (1989), pp 589-604.

[Schor, Daly, Lee, Tibbitts] Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, Beth R. Tibbitts, Advances in Rete pattern matching. *AAAI-86 Proceedings*, American Association for Artificial Intelligence.

[Schreye, Bruynooghe] Danny De Schreye and Maurice Bruynooghe, The compilation of forward checking regimes through meta-interpretation and transformation, *Meta-programming in Logic Programming* edited by Harvey Abramson and M. H. Rogers, The MIT Press, 1989.

[Shapiro] E. Shapiro, Alternation and the computational complexity of logic programs, *Journal of Logic Programming*, 1984 Vol. 1 (1984) pp 19-33.

[Shepherdson] John C. Shepherdson, Negation in Logic Programming, *Foundation of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988.

[Steele 1] Guy L. Steele,*The definition and implementation of a computer programming language based on constraints*, Ph. D. Thesis, MIT, Aug. 1980.

[Steele 2] Guy L. Steele, *Common Lisp: The language.* Digital Press, Massachusetts,1984, 1990.

[Sussman Steele] Gerald Jay Sussman and Guy Lewis Steele, CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions, *Artificial Intelligence* **14**, North-Holland Publishing Company, 1980, pp 1-39.

[Stokey] Richard J. Stokey, AI factory scheduling: Mutiple problem formulations, *SIGART Newsletter*, Number 110, October 1989, ACM Press.

[Tatar] Deborah G. Tatar, *A programmer's guide to common Lisp*, Digital Press, Massachusetts, 1987.

[Van Hentenryck] Pascal Van Hentenryck,*Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Massachusetts 1989.

[Van Hentenryck, Carillon] Pascal Van Hentenryck and Jean-Philippe Carillon, Generality versus Specificity: An Experience with AI and OR technique, *AAAI 1988 Proceedings*, Minnesota, pp 660-664.

[Warren] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Report 309, SRI International, 1983.

[Williams] H. P. Williams, *Model building in mathematical programming*,John Wiley & Sons,New York, 1985.

[Winston] Patrick Henry Winston, *Artificial Intelligence*, Addison-Wesley Publishing Co., Second Edition, Massachusetts,1984.

[Winston, Horn] Patrick Henry Winston and Berthold Klaus Paul Horn, *LISP*. Addison-Wesley Publishing Company,Massachusetts, 1984.