AN EMPIRICAL STUDY OF DIFFERENT BRANCHING STRATEGIES FOR CONSTRAINT SATISFACTION PROBLEMS

by

Vincent Park

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, January 2004

Abstract

Many real life problems can be formulated as constraint satisfaction problems *(CSPs)*. Backtracking search algorithms are usually employed to solve *CSPs* and in backtracking search the choice of branching strategies can be critical since they specify how a search algorithm can instantiate a variable and how a problem can be reduced into subproblems; that is, they define a search tree. In spite of the apparent importance of the branching strategy, there have been only a few empirical studies about different branching strategies and they all have been tested exclusively for numerical constraints. In this thesis, we employ the three most commonly used branching strategies in solving finite domain *CSPs*. These branching strategies are described as follows: first, a branching strategy with strong commitment assigns its variables in the early stage of the search as in k-Way branching; second, 2-Way branching guides a search by branching one side with assigning a variable and the other with eliminating the assigned value; third, the domain splitting strategy, based on the least commitment principle, branches by dividing a variable's domain rather than by assigning a single value to a variable. In our experiments, we compared the efficiency of different branching strategies in terms of their execution times and the number of choice points in solving finite domain *CSPs*. Interestingly, our experiments provide evidence that the choice of branching strategy for finite domain problems does not matter much in most cases—provided we are using an effective variable ordering heuristic—as domain splitting and 2-Way branching end up simulating k-Way branching. However, for an optimization problem with large domain size, the branching strategy with the least commitment principle can be more efficient than the other strategies. This empirical study will hopefully interest other practitioners to take different branching schemes into consideration in designing heuristics.

Contents

1 Introduction **1**

2 Background **7**

2.1 Basic Definitions 7

2.2 Constraint Propagation 10

2.3 Backtracking Search *(BT)* 11

3 Different Branching Strategies in a Search **21**

3.1 Branching Strategies 21

3.2 Heuristics: Branch Ordering Heuristics 30

3.3 Related Work 32

4 Experimental Setup **35**

4.1 Problems 35

4.1.1 The N-queens Problem 35

4.1.2 The Golomb Ruler Problem 36

4.1.3 The Car Sequencing Problem 37

4.1.4 The Truck Scheduling Problem *(TSP)* 38

4.1.5 The Instruction Scheduling Problem 39

4.1.6 The Uniform Binary Random Problem 40

4.2 Implementation 41

5 Experimental Results **48**

5.1 Results for the N-queens Problems 48

5.2 Results for the Golomb Ruler Problems 51
5.3 Results for the Instruction Scheduling Problems 54
5.4 Results for the Car Sequencing Problems 56
5.5 Results for the Truck Scheduling Problems (*TSP*) 58
5.6 Results for the Uniform Binary Random Problems 59

6 Discussion **65**

7 Conclusion **68**

List of Tables

4.1 The example of configurations in the car sequencing problem (taken from [6]) 38

4.2 Table of different branching ordering strategies implemented in ILOG . . . 47

5.1 The execution time of different branching strategies for the 20-queens problem 49

5.2 The running time of solving the N-queens problems with MinSize 50

5.3 The running time of solving the N-queens problems with MaxMin 50

5.4 The choice points of solving the N-queens problems with MinSize 50

5.5 The execution time of different branching strategies for the OGR with 8 marks 51

5.6 The running time of different branching strategies for the Golomb ruler problems with lexicogr

5.7 The running time of different branching strategies for the Golomb ruler problems with minimu

5.8 The running time of different branching strategies for the Golomb ruler problems with least mi

5.9 The running time of different branching strategies for the Golomb ruler problems with least ma

5.10 The choice points of different branching methods for the Golomb ruler problems with lexicogra

5.11 The execution time of different branching strategies in scheduling 394 instructions 54

5.12 The choice points of different branching strategies in scheduling 69 instructions 54

5.13 The execution time of different branching strategies in the instruction scheduling problems 55

5.14 The running time of different branching strategies in sequencing 20 cars . . 56

5.15 The running time of different branching strategies with Lex in sequencing cars 57

5.16 The running time of different branching strategies with MinSize in sequencing cars 57

5.17 The running time of different branching strategies with MaxMin in sequencing cars 57

5.18 The running time of different branching strategies with MinMax in sequencing cars 58

5.19 The choice points of different branching strategies with Lex in sequencing cars 58

5.20 The execution time of different branching strategies in *TSP* with 7 customers 59

5.21 The choice points of different branching strategies in *TSP* with 7 customers 59

5.22 The execution time of different branching strategies in *TSPs* 60
5.23 The running time for different branching strategies combined with MaxSize in solving the rand
5.24 The number of choice points for different branching strategies combined with MinSize in solvin
5.25 The running time of different branching strategies combined with Lex in solving the random pr

List of Figures

1.1 An underlying search tree of k-Way branching that is exclusively used in backtracking 3
1.2 An underlying search tree of 2-Way branching strategy with using equality 4
1.3 An underlying search tree of the domain splitting strategy with pivoting in the middle of doma

3.1 A search trace of chronological backtracking with k-Way branching in solving 4-queens problem
3.2 A search trace of chronological backtracking with 2-Way branching in solving 4-queens problem
3.3 A search trace of chronological backtracking with domain splitting in solving 4-queens problem
3.4 A search trace of backtracking with forward checking for k-Way branching in solving 4-queens
3.5 A search trace of backtracking with forward checking for 2-Way branching in solving 4-queens
3.6 A search trace of backtracking with forward checking for domain splitting in solving 4-queens
3.7 A part of a search tree for k-Way Branching in solving the 4-queens problem 29
3.8 A part of a search tree for 2-Way branching that simulates k-Way branching 29
3.9 A part of a search tree for domain splitting that simulates 2-Way branching 30

4.1 ILOG code for solving N-queens problems from [6] 41
4.2 ILOG code segment for the objective function 43
4.3 ILOG code for 2-Way branching 44
4.4 ILOG code for domain splitting 46
4.5 ILOG code for an enumeration algorithm called Generate 46

5.1 The running time of different branching strategies in solving the random problem with 32 varia

List of Algorithms

1 Chronological Backtracking Algorithm 12
2 Generic Backtracking Algorithm from [12] 14
3 Subroutine for chronological backtracking 15
4 Subroutine for BT with forward checking from [12] 16
5 Pseudo code of Update used in forward checking 16
6 Subroutine for Maintaining Arc Consistency *(MAC)* from [12] 18
7 Pseudo code for Arc Consistency (AC) algorithm in BT with AC 19
8 Pseudo code for Node Consistency (NC) algorithm in AC 20
9 Pseudo code of Revise-Domain in BT with AC 20

Chapter 1

Introduction

Many real life problems can be formulated as constraint satisfaction problems *(CSPs)* [8] that are defined by three components, namely the variables, V, representing the entities in a problem, the domains of the variables, D, signifying the possible combinations of answers, and lastly the constraints, C, specifying the relations that hold in a solution and mapping a problem into a solution. Among different *CSPs*, finite domain problems are particularly interesting to us since they make up the largest class of real life problems tackled by constraint programming *(CP)* systems [9]. *CP* languages providing constraints over finite domains have proved to be ideal for solving such problems and have had the most industrial impact [9]. In [12], Tsang also suggests that efficient algorithms can exploit the finiteness in the number of variables and the size of domain. Henceforth, we only deal with finite domain *CSPs* such as N-queens problems, optimal Golomb ruler problems, car sequencing problems, truck scheduling problems, instruction scheduling problems, and random problems—we will discuss these problems in more depth in Section 4.1.

To better understand modeling a *CSP*, we briefly define the 4-queens problem as a *CSP*. The 4-queens problem is to find the legal arrangement of four queens on the four by four chess board without imposing a threat to each other—for people who are not familiar with chess, the problem is basically to occupy four squares in the four by four grids without facing each other on the same column, on the same row or diagonally. As a *CSP*, we can define a column of the board as a variable and a row of the board as a value. Then, the assignment of a value to a variable will represent the location of a queen. Also, it is possible to model a row as a variable and a column as a value since a grid of a chess board is symmetrical. Formally, $V = \{x_1, x_2, x_3, x_4\}$ and $D = \{d_1, d_2, d_3, d_4\}$. Then, arithmetic constraints are used to find the legal position of four queens as follows:

- When $1 \leq i, j \leq 4$, $x_i \neq x_j$
- $x_i + i \neq x_j + j$
- $x_i - i \neq x_j - j$

In the ILOG environment [6][1] that aids us in carrying out the experiments, the primitive constraint called `alldiff` is employed to ensure the inequality since this primitive constraint is more efficient than arithmetic constraints. The model with `alldiff` is introduced in Section 2.1. We will use this 4-queens problem as an example to illustrate different concepts throughout this thesis.

In solving a *CSP*, usually a backtracking search is employed to find a solution. During the search, each variable has to be assigned a legal value in order to be a solution; legal means that the assignment does not violate the constraints imposed on a variable or variables. Labeling is used to guarantee that each variable is instantiated [9]. In labeling, we have a choice of how to branch and assign. The concept of branching comes from the Operations Research *(OR)* and SAT communities where it defines a search tree. However, branching in labeling from the Constraint Programming *(CP)* community has a slightly different meaning in that it could mean either all of the backtracking search process including a variable ordering heuristic [9] or only the instantiation a variable as in [12]. In this thesis, the notion of branching from *OR* and the definition from [12] are employed. Furthermore, the backtracking search with full arc consistency (also refers to the backtracking search with domain consistency) is exclusively used because it prunes more search spaces than the other backtracking searches.

A variety of branching strategies are proposed to reduce the search space by taking advantage of the structure of a particular problem, as a backtracking search, the most expensive operator in a search, uses labeling to try different values for the variables [9]. We

[1]ILOG is a commercially available software package that provides a constraint programming library written in **C++** [6]. We have employed version 4.2 of ILOG in this thesis.

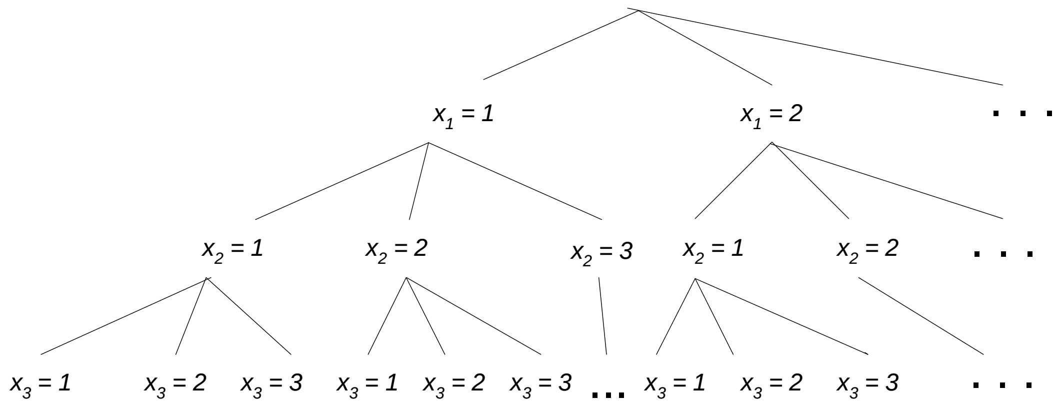

Figure 1.1: An underlying search tree of k-Way branching that is exclusively used in backtracking

mainly focus on three different branching strategies in labeling, analyzing and comparing them. The names of the different branching strategies, k-Way and 2-Way branching, appear in [10] and the domain splitting strategy is introduced in [2].

The first strategy, k-Way branching, assigns a value to a constrained variable in the order of variables that is determined by a variable ordering heuristic, and this strategy is used almost exclusively in *CSP* (as shown in Figure 1.1[2]). Here, the branching factor becomes equal to the maximum domain size since it has to instantiate each variable with all the possible values in the domain in the predefined or dynamically arranged order. Assuming that the size of domains are equal to d and there are n number of variables, then the branching factor of a search tree, b, becomes equal to d, the depth of the tree is $O\left(n\right)$ and the number of nodes in the search tree is $O\left(d^n\right)$ or $O\left(b^n\right)$.

The second strategy, 2-Way branching, splits a domain of a variable into two sub-domains so that one sub-domain contains a value that is assigned to a variable and the other has the rest of the values in the domain excluding the assigned value. Solely based on the membership, it splits a domain in two ways. This binary tree has depth larger

[2]Figures 1.1, 1.2, and 1.3 are the so called underlying search trees which are implicitly defined by the branch ordering heuristic and the branching strategy in comparison with Figures 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6, which are actually the traces of a search algorithm.

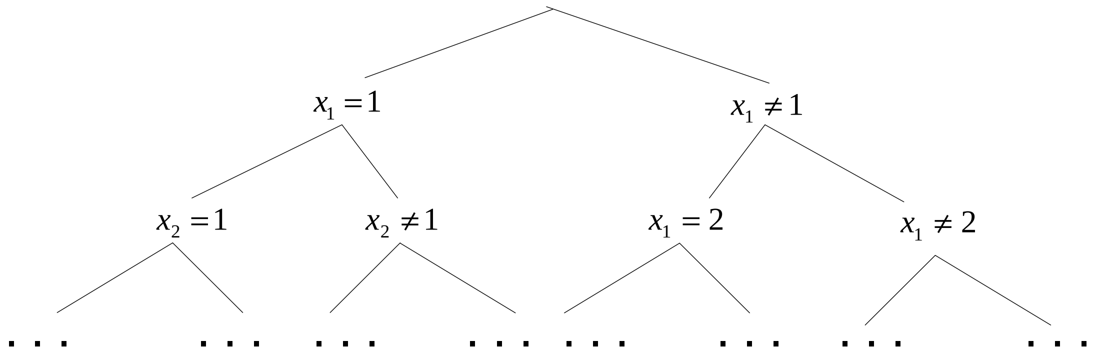

Figure 1.2: An underlying search tree of 2-Way branching strategy with using equality

than n; more precisely, it would be $O(d * n)$. The number of nodes generated by 2-Way branching is $O\left(2^{d*n}\right)$, where the branching factor is two. It is also worth mentioning that 2-Way branching can simulate k-Way branching with increasing depth, but k-Way branching cannot simulate 2-Way branching [10]. The graphical illustration of 2-Way branching can be found in Figure 1.2.

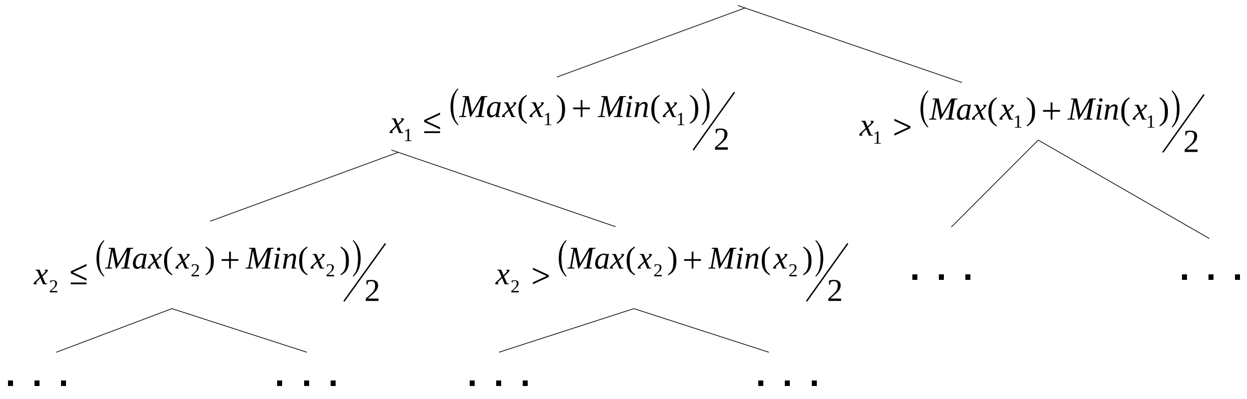
$x_1 \leq (Max(x_1) + Min(x_1))/2$
$x_1 > (Max(x_1) + Min(x_1))/2$
$x_2 \leq (Max(x_2) + Min(x_2))/2$
$x_2 > (Max(x_2) + Min(x_2))/2$

Figure 1.3: An underlying search tree of the domain splitting strategy with pivoting in the middle of domain

The last strategy, called "domain splitting" (some literature also refers to it as dichotomic search [2]), repeatedly reduces the domain of each variable by splitting the current domain into subdomains that contain only some part of the original domain. Each subdomain is mutually exclusive, meaning that there are no duplicate values in subdomains. The most obvious example can be achieved by splitting at the middle point of the domain with resulting subdomains as the lower and upper halves [9]. Domain splitting employs the same idea as 2-Way branching; however, it is different since it splits in the pivot that the user chooses instead of being based on the membership of a single assigned value and it may most likely contain more than one value in subdomains. In this strategy, there are parameters to be set to maximize the performance; for example, the size of subdomains, the number of subdomains and the pivot need to be set in order to define how deep a search tree is, how large a search tree is, and how well a search tree guides to a solution. The pictorial description of domain splitting is illustrated in Figure 1.3. With the assumption that the search tree is binary, the depth of this tree becomes $O\left(n * \log d\right)$, and the upper bound of nodes in the tree is $O\left(2^{n * \log d}\right)$. About the power of expressiveness, domain splitting can simulate both 2-Way and k-Way branching, but 2-Way and k-Way branching are not able to map domain splitting; therefore, it is safe to conclude that the domain splitting strategy is the most expressive among three strategies and then followed by 2-Way branching and k-Way branching in order.

Usually in labeling, branching strategies are often combined with variable and value ordering heuristics. Variable ordering defines the order of branches to be explored in a search tree and value ordering specifies the order of values to be assigned. From here on, we adopt the notion of a branch ordering heuristic which signifies the variable ordering heuristics that we have employed to test with branching strategies.

To the author's knowledge, the comparison among different branching strategies in solving finite domain *CSPs* has not been made so far. This thesis attempts to compare the efficiency of different branching techniques with combinations of branch ordering heuristics and, furthermore, to classify the types of problems that are beneficial in using particular branching techniques. After extensive empirical studies, we observe that there is some promising evidence supporting the fact that domain splitting seems to work better than k-Way and 2-Way branching for an optimization problem with large domain; however, in most cases, the choice of branching strategies becomes trivial when they are combined with the most efficient variable ordering heuristics since their running times behave very alike as domain splitting and 2-Way branching end up simulating k-Way branching. These results show us that we need to design specialized heuristics to exploit the different branching strategies so that the different branching strategies can be better used.

The remainder of the thesis is organized in the following manner. In Chapter 2, we elucidate the basic definitions of *CSP*, constraint propagation, and backtracking search. In Chapter 3, we discuss branching strategies in more depth and also present some of available branch ordering heuristics. In addition, we review some of current work related to our research. In Chapter 4, we explain how we set up the experiments. We describe the problems used in this experiment and show how ILOG Solver [6] is used in the implementations. In Chapter 5, the results of the experiments are reported and in Chapter 6, we analyze the empirical data and evaluate our contributions. Lastly in Chapter 7, we conclude with a glimpse of future research directions.

Chapter 2

Background

In this chapter, we present some necessary background information to understand this thesis. In Section 2.1, we discuss some basic definitions and clarify some terminology that we use throughout the thesis. Some methods to prune the search space are presented in Section 2.2. Lastly, in Section 2.3, we introduce some basic techniques and algorithms commonly used in solving constraint satisfaction problems.

2.1 Basic Definitions

A *constraint satisfaction problem*(CSP) consists of a set of n *variables*, $X = \{x_1, \ldots, x_n\}$; a set of d *values*, $D = \{v_1, \ldots, v_d\}$, where each variable $x_i \in X$ has an associated finite domain $dom(x_i) \subseteq D$ of possible values; and a collection of m *constraints*, $\{C_1, \ldots, C_m\}$. Each constraint C_i is a constraint over some set of variables, denoted by $vars(C_i)$. C_{x_i,x_j} also signifies the constraint imposed between the variables i and j, meaning that $vars(C_{x_i,x_j}) = \{x_i, x_j\}$.

The following definitions in this section are adopted from [12].

Definition 1 (Label) A label is simply a variable-value pair that represents the assignment of the value to the variable. $x = v$ denotes the label of assigning the value v to the variable x, and it is only meaningful if v is in the domain of x (i.e. $v \in dom(x)$).

Moreover, labeling means the process of assigning a value to a variable.

Definition 2 (Compound Label) A **compound label** is the simultaneous assignment of values to a (possibly empty) set of variables.

Hereafter, we employ the notation $\{x_1 = v_1, x_2 = v_2, \ldots, x_n = v_n\}$ to denote the compound label of assigning $v_1, v_2, \ldots, v_n$ to $x_1, x_2, \ldots, x_n$ respectively.

Definition 3 (Satisfies) If the variables of the compound label L are the same as those variables of the elements of the compound labels in constraint C, then L **satisfies** C if and only if $L \in C$. In other words, $\{x_1 = v_1, x_2 = v_2, \ldots, x_k = v_k\}$ satisfies $C_{x_1,x_2,\ldots,x_k}$ if and only if $\{x_1 = v_1, x_2 = v_2, \ldots, x_k = v_k\} \in C_{x_1,x_2,\ldots,x_k}$.

The task in a *CSP* is to assign a value to each variable such that all the constraints are satisfied simultaneously. A *CSP* is satisfiable if a solution tuple exists. A solution tuple of a CSP means a compound label for all those variables which satisfy all constraints.

For instance, with the 4-queens problem as in the earlier example, the variables are the four different columns, $n = 4, X = \{x_1, x_2, x_3, x_4\}$, and x_i represents the i^{th} column. The domains of these variables are initially set to be the values of four different rows, $d = 4, D = 1, 2, 3, 4$. The constraints in this particular problem can also be seen as follows:

- all the variables x_i are pair-wise distinct.
- all the variables $x_i + i$ are pair-wise distinct.
- all the variables $x_i - i$ are pair-wise distinct.

A constraint such as `alldiff` can be employed to ensure the uniqueness of all the variables. For example, `alldiff(`x_1, x_2, x_3, x_4`)` will make sure that all the variables are pair-wise distinct—i.e. $x_1 \neq x_2$, $x_1 \neq x_3$, $x_1 \neq x_4$, $x_2 \neq x_3$, $x_2 \neq x_4$, and $x_3 \neq x_4$.

For 4-queens problem, there is more than one solution for the problem, and the compound label, $\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$, can be one of solutions. This compound label specifies to place four queens at the following locations: the first queen in the cell at the first column and the second row; the second queen in the cell at the second column and the fourth row; the third queen in the cell at the third column and the first row; the fourth queen in the cell at the fourth row and the third row.

As shown in the 4-queens example, a *CSP* problem can have multiple solutions, and in that case, sometimes, we are not only interested in the satisfiability of a constraint but also wish to find the best solution. Depending on the number of solutions required, CSPs can be classified into the three categories as follows [12]:

- CSPs in which one has to find any solution tuple.
- CSPs in which one has to find all solution tuples.
- CSPs in which one has to find optimal solutions, where optimality is defined according to some domain knowledge.

Finding the best solution of a *CSP* is called an optimization problem. This requires some way of specifying which solutions are better than others. The usual way of doing this by giving an objective function that maps each solution to a real value. By convention, we will assume that the aim is to minimize the objective function **f**. However, we can easily change the function to find the maximum value as **f** is changed into $-\mathbf{f}$. An example of an optimization problem is an Optimal Golomb Ruler problem of which the goal is to minimize the overall length in the ruler. The objective function for this problem can be simply to minimize the last scale, say x_n. So the objective function looks like $\mathbf{f}(x_n) = min(x_n)$. This problem will be dealt with in more detail later in the problems section.

In this thesis, we have used six different finite domain problems to test the effectiveness of the branching strategies. The problems are namely, the N-queens problems, the Golomb ruler problems, the instruction scheduling problem, the car sequencing problem, the truck scheduling problem, and random problems. We have chosen all finite problems since, among different types of *CSPs*, finite domain problems make up the largest class of real life problems tackled by Constraint Programming *(CP)* systems [9].

We can also distinguish *CSPs* into different problem classes based on the type of constraints. A binary *CSP*, or binary constraint problem, is a *CSP* with unary and binary constraints only. A *CSP* with constraints not limited to unary and binary will be referred to as a general *CSP*.

Therefore, on the merit of what types of constraints a problem employs and what kind of solutions a problem has, we can categorize our test cases in the following way:

- the N-queens and the car sequencing problems are general satisfiable problems,
- the random problems are binary satisfiable problems,
- the Golomb ruler, instructions scheduling, and truck scheduling problems are general optimization problems.

2.2 Constraint Propagation

The goal of propagation is to efficiently look for the potential *false*[1] or *valuation*[2] domains in order to prune an unproductive search space. The following definitions are based on the presentation in [9].

To find value and false domains efficiently, various techniques have been proposed. In here, we closely look at domain consistency that we have extensively used in our experiments since it propagates more tightly than the others. Due to its tightness, it also requires more computation time, but we overlook this aspect because for the large problem size that we have tested, the efficiency of domain consistency seems to out weigh its overhead cost.

Definition 4 (Domain consistency) A constraint C is **domain consistent** with domain D if for each variable $x \in vars(C)$ and domain assignment $v \in dom(x)$, there is an assignment to the remaining variables in C, say $x_1, x_3, \ldots, x_k$, such that $v_j \in dom(x_j)$ for $1 \leq j \leq k$ and $\{x = v, x_1 = v_1, x_2 = v_2, \ldots x_k = v_k\}$ is a solution of C. A *CSP* with constraint $C_1 \wedge \ldots \wedge C_m$ and domain D is domain consistent if each primitive constraint C_i is domain consistent with D for $1 \leq i \leq m$.

To elucidate better how the propagator works, the previous 4-queens example is used. The propagator will proceed with the following steps to remove 1 from $dom(x_1)$ that violates domain consistency:

- $x_1 = 1$ and $x_2 = 1$ violate $x_1 \neq x_2$, and

[1]A domain is a **false** domain if some variable in the domain becomes empty and it indicates that the original problem is unsatisfiable.

[2]A domain is a **valuation** domain if every variable in the domain has a single value left in their domain and it indicates that the resulting assignment is a solution for the original problem.

- $x_1 = 1$ and $x_2 = 2$ violate $x_1 - 1 \neq x_2 - 2$, but
- $x_1 = 1$ and $x_2 = 3$ satisfy C_{x_1,x_2}.
- Similarly $x_1 = 2$, $x_1 = 3$ and $x_1 = 4$ are tested.
- Then, the values for x_2 are examined.
- Since all the values for x_1 and x_2 can have a satisfying assignment for the constraint imposed between them, x_1 and x_2 indeed are domain consistent for C_{x_1,x_2}.
- However, further investigations with other constraints in C reveal that $x_1 = 1$ cannot have a successful assignment for C_{x_3,x_4}.
- Therefore, $1 \notin dom(x_1)$ in order to maintain domain consistency for the problem.

For other values in the domain, they are also iteratively and exhaustively examined with every constraint in the problem, $C = \{C_{x_1,x_2}, C_{x_1,x_3}, C_{x_1,x_4}, C_{x_2,x_3}, C_{x_2,x_4}, C_{x_3,x_4}\}$ and those values that make a constraint domain inconsistent are removed from the domain. At the end, the propagator ensures that all the constraints are domain consistent with the newly updated domain. After propagating the domains, if a domain of a variable becomes a valuation domain, it becomes a label for the variable in a solution, and if a domain of a variable is a false domain, it means that the problem is unsatisfiable.

2.3 Backtracking Search *(BT)*

In this section, we discuss search strategies commonly used in *CSP*. We have solely employed with the backtracking search algorithm with full arc consistency since it has the most pruning effect. For pedagogical reasons, we start by explaining chronological backtracking search *(BT)*, then *(BT)* with forward checking and with full arc consistency as a consistency based search. The material in this section is based on the presentation in [12].

Algorithm 1: Chronological Backtracking Algorithm

```
Algorithm: Chronological Backtracking($V, D, C, Branch$)
begin
    Backtrack($V, \{\}, D, C, Branch$) ;
end
```

One of the simplest techniques for determining satisfiability of a *CSP* is chronological backtracking. In backtracking, k-Way branching is to determine satisfiability of a CSP by choosing one variable at a time, assigning one value for it at a time, and ensuring that the newly selected label is compatible with the already assigned compound label. If the current assignment violates any constraints imposed on the problem, then an alternative value in its domain is tried. If at any stage, no value can be assigned to a variable without violating any constraints, the label which was last selected is revised, meaning that an alternative value is tried for the variable in the label. This process repeats itself until either a solution is found or all the values in the domains of variables are exhaustively tried.

The 2-Way branching strategy backtracks similarly as k-Way does; however, it requires more branches for a variable to be labeled than k-Way branching since after assigning a value, v, to a variable, x, 2-Way branching adds a new constraint $x \neq v$ to the constraint, C, rather than trying different values (refer to Algorithm 2).

Unlike the other two branching strategies, domain splitting does not instantiate a variable unless the domain of a variable is a valuation domain. Rather, it splits a domain into the two sub-domains on the merit of the predefined pivot point[3]. As a result, the backtracking algorithm for domain splitting requires more steps to separate the domains into small subdomains until either a valuation or a false domain is obtained. As shown in Algorithm 2, the techniques for domain splitting to backtrack in case of a false domain are very similar to the other branching strategies. The reason is that even in domain splitting, a variable first needs to be assigned and tested for the satisfiability against constraints before a search backtracks to find an alternative value.

[3]In this thesis, we employed the pivot point to be a middle of a domain; however, we are also aware that there can be many other ways to divide a domain that exploit the domain information in order to prune effectively.

The pseudo code for the chronological backtracking algorithm is found in Algorithm 1. Algorithm 1 calls the helper procedure, `Backtrack` in Algorithm 2. Algorithms 1 and 2 are also employed in consistency based searches with the minor changes made in the subroutine area in Algorithm 2. For chronological backtracking, the subroutine in Algorithm 3 can be placed inside Algorithm 2.

Algorithm 2: Generic Backtracking Algorithm from [12]

```
Algorithm: Backtrack (UNLABELLED, COMPOUND-LABEL, $D$, $C$, BRANCH)
UNLABELLED is a set of variables to be labeled
COMPOUND-LABEL is a set of labels already committed to
$D$ is a set of domains
$C$ is a set of constraints
BRANCH specifies a branching strategy
begin
    if UNLABELLED $= \{\}$ then
        return COMPOUND-LABEL ;
    else
        begin
            Pick one variable $x$ from UNLABELLED ;
            repeat
                if BRANCH = k-Way branching then
                    Pick one value $v \in dom(x)$;
                    Delete $v$ from $dom(x)$;
                    if COMPOUND-LABEL $\cup \{x = v\}$ violates no constraints then
                        /* Here for Subroutine */
                if BRANCH = 2-Way branching then
                    Pick one value $v \in dom(x)$;
                    $C = C \cup x \neq v$;
                    if COMPOUND-LABEL $\cup \{x = v\}$ violates no constraints then
                        /* Here for Subroutine */
                if BRANCH = domain splitting then
                    if $dom(x)$ is a valuation then
                        Pick one value $v \in dom(x)$;
                        Delete $v$ from $dom(x)$;
                        if COMPOUND-LABEL $\cup \{x = v\}$ violates no constraints then
                            /* Here for Subroutine */
                    else
                        divide $dom(x)$ into two subdomains, $dom_1(x)$ and $dom_2(x)$;
                        $D' = \{D - dom(x)\} \cup dom_1(x)$;
                        Backtrack (UNLABELLED, COMPOUND-LABEL, $D'$, $C$, BRANCH);
                        $D' = \{D - dom(x)\} \cup dom_2(x)$;
                        Backtrack (UNLABELLED, COMPOUND-LABEL, $D'$, $C$, BRANCH);
            until $D_x = \{\}$;
            return $NIL$;
            /* signifying no solution */
        end
end
```

Algorithm 3: Subroutine for chronological backtracking

```
begin
    RESULT ← Backtrack (UNLABELLED−{$x$}, COMPOUND-LABEL∪{$x = v$}, $D$, $C$, BRANCH);
    if RESULT ≠ $NIL$ then
        return RESULT ;
end
```

Chronological backtracking is sometimes referred to as a naive backtracking because it is quite costly to use; it has the time complexity of $O\left(m * d^n\right)$, and the space complexity is $O\left(n * d\right)$. To improve the efficiency, *CSPs* are usually solved by interleaving backtracking search with constraint propagation, such as forward checking and maintaining arc consistency. These constraint propagations use the information of a current label to prune the search space of future variables. With the help of an effective propagator, the potential search space can be pruned; consequently, the resulting time complexity of the algorithm can also dramatically be improved. We call these type of algorithms Lookahead algorithms. The basic steps of Lookahead algorithms are as follows:

1. The search algorithm commits one label at a time,
2. simplifies the problem at each step in order to reduce the search space,
3. and detects unsatisifiability.

One of BT with Lookahead algorithms is Forward Checking (FC) that does exactly the same thing as backtracking except that it maintains the invariance that for every unlabeled variable there exists at least one value in its domain which is compatible with the labels that have been committed [12]. To ensure that this is true, every time a label **L** is committed, the propagator for FC will remove values from the domains of the unlabeled variables which are incompatible with **L**. If the domain of any of the unlabeled variables is reduced to a false domain, then **L** will be rejected. Otherwise, FC would try to label the unlabeled variable, until all the variables have been labeled. In case all the labels of the current variable have been rejected, FC will backtrack to the previous variable as Backtracking does. If there is no variable to backtrack to, then the problem is insoluble [12].

Algorithm 4: Subroutine for BT with forward checking from [12]

```
begin
    $D' \leftarrow$ Update (UNLABELLED $- \{x\}, D, C, \{x = v\}$);
    if $no\ domain\ in\ D'\ is\ empty$ then
        begin
            RESULT $\leftarrow$ Backtrack (UNLABELLED $- \{x\}$, COMPOUND-LABEL $\cup \{x = v\}$), $D'$,
            $C$);
            if RESULT $\neq NIL$ then
                return RESULT ;
        end
end
```

In the pseudo code for *BT* with *FC*, Algorithm 2 employs Algorithm 4 as a subroutine. In Algorithm 4, the `Update` procedure is first used to make sure that the domains are consistent after the propagation, and the detail of this algorithm can be found in Algorithm 5.

Algorithm 5: Pseudo code of Update used in forward checking

```
Algorithm: Update ($W, D, C$, LABEL)
/* Update only considers binary constraints */
begin
    $D' \leftarrow D$;
    foreach $variable\ y \in W$ do
        foreach $value\ v \in D'_y$ do
            if $y = v$ is incompatible with LABEL with respect to the constraints in $C$ then
                $D'_y \leftarrow D'_y - \{v\}$;
    return $D'$;
end
```

To delineate more clearly about `Update`, the following example shows how `Update` updates the domains of the 4-queens problem upon labeling $x_1 = 1$:

- 1 is removed from $dom(x_2)$ since $x_2 = 1$ violates $x_1 \neq x_2$,

- 2 is removed from $dom(x_2)$ since $x_2 = 2$ violates $x_1 - 1 \neq x_2 - 2$,
- 1 is removed from $dom(x_3)$ since $x_3 = 1$ violates $x_1 \neq x_3$,
- 3 is removed from $dom(x_3)$ since $x_3 = 3$ violates $x_1 - 1 \neq x_3 - 3$,
- 1 is removed from $dom(x_4)$ since $x_4 = 1$ violates $x_1 \neq x_4$, and
- 4 is removed from $dom(x_4)$ since $x_4 = 4$ violates $x_1 - 1 \neq x_4 - 4$.

In comparing forward checking with chronological backtracking, we find the main difference is that *FC* calls `Update` every time after a label is committed to (as shown in Algorithm '4'). If the frequent updates of domains of the unlabeled variables cause any domain to be a false domain (empty domain), *FC* will reject the current label without wasting computation time to explore the unproductive search space.

The other Lookahead algorithm is a backtracking search with the aid of full arc consistency; this algorithm is usually referred to as the Maintaining Arc Consistency *(MAC)* backtracking algorithm. Arc consistency *(AC)* is a special type of domain consistency when the arity[4] of constraints is less than or equal to 2. In this strategy, when a label is committed, the values of unlabeled variables which are incompatible with the committed label are removed from its domain. Furthermore, BT with AC ensures that unlabeled variables are arc consistent with each other. In other words, it makes sure that there exists a pair of compatible labels between every pair of unlabeled variables.

By spending more computational effort in problem reduction, the arc-consistency Lookahead algorithm makes it possible to reject more redundant labels than forward checking. In forwarding checking, the domains of the unlabeled variables are only checked against the committed labels. It is possible to reduce the problem further by maintaining arc consistency in each step after a label has been committed to.

For a better illustration, let's consider the same example as in forward checking. Label $\{x_1 = 1\}$ will prune $\{x_2 = 1, x_2 = 2, x_3 = 1, x_3 = 3, x_4 = 1, x_4 = 4\}$ in forward checking, but BT with AC will prune more search space than FC as follows:

[4]The arity of a constraint is the number of variables that the constraint restricts. For example, the arity of C_{x_1,x_2} is 2.

- First, $\{x_2 = 1, x_2 = 2, x_3 = 1, x_3 = 3, x_4 = 1, x_4 = 4\}$is eliminated as in *FC*.
- Then, only $x_2 = 3$, and $x_2 = 4$ are possible to be a label for x_2, but BT with FC further prunes $x_2 = 3$ because $x_2 = 3$ causes a conflict with labeling in x_3, specifically $x_3 = 2$ and $x_3 = 4$.
- Furthermore, as the label $x_2 = 4$ is not arc consistent with x_3 and x_4, $x_2 = 4$ is also pruned and this makes $dom(x_2)$ a false domain; therefore, the algorithm backtracks to $x_1 = 1$ in order to try a different label for x_1.

As shown in the above example, BT with AC prunes a search space very effectively. To understand better about this algorithm, the pseudo code of the subroutine for BT with AC is presented in algorithm 6.

Algorithm 6: Subroutine for Maintaining Arc Consistency *(MAC)* from [12]

```
begin
    $D' \leftarrow$ Update (UNLABELLED $-\{x\}, D, C, x = v$);
    /* Update is the same Update procedure used in Forward-Checking */
    (UNLABELLED $-\{x\}$, $D$", $C$) $\leftarrow$ AC (UNLABELLED $-\{x\}$, $D'$, $C$);
    if no domain $\in D$" is empty then
        begin
            RESULT $\leftarrow$ Backtrack (UNLABELLED$-\{x\}$, COMPOUND-LABEL$\cup\{x = v\}$, $D$", $C$);
            if RESULT $\neq NIL$ then return RESULT ;
        end
end
```

Algorithm 7: Pseudo code for Arc Consistency (AC) algorithm in BT with AC

```
Algorithm: AC $(V, D, C,)$
begin
    /* check first node consistency of variables */
    NC $(V, D, C)$;
    $Q \leftarrow \{x = y \mid C_{x,y} \in C\}$;
    repeat
        CHANGED $\leftarrow$ FALSE;
        foreach $x = y \in Q$ do
            CHANGED $\leftarrow$ (Revise-Domain $(x = y, (Z, D, C))$ or CHANGED);
    until $\neg$CHANGED;
    return $(V, D, C)$;
end
```

The pseudo code of the procedure `AC` used in Algorithm 6 is presented in Algorithm 7 and `Update` in Algorithm 6 is the same as the `Update` in forward checking. Basically, Algorithm 6 filters domain values as forward checking and prune more inconsistent values in order to maintain arc consistency.

As mentioned earlier, arc-consistency is a special type of domain consistency when the arity of a constraint is less than or equal to 2. `AC` in Algorithm 7 is employed to maintain domain consistency. First, `AC` calls the procedure, `NC`, as in Algorithm 8 to enforce domain consistency for constraints with arity, 1[5]. Then, `AC` can maintain domain consistency for the constraints of arity 2 with the aid of `Revised-Domain` in Algorithm 9 that removes any values that do not satisfy the given constraints and extensively tries all the value pairs of the variables for the constraints.

[5]In some literature, node consistency is defined as domain consistency for a constraint of arity 1

Algorithm 8: Pseudo code for Node Consistency (NC) algorithm in AC

```
Algorithm: NC $(V, D, C)$
begin
    foreach variables $x \in V$ do
        foreach values $v \in D_x$ do
            if $\neg$ SATISFIES $(x = v, C_x)$ then
                $D_x \leftarrow D_x - \{v\}$;
    /* certain $D_x$ may be updated */
    return $(V, D, C)$;
end
```

Algorithm 9: Pseudo code of Revise-Domain in BT with AC

```
Algorithm: Revise-Domain $(V, D, C)$
/* If there is no support for the given label, Revise-Domain removes the label from the domain
and returns true */
begin
    DELETED $\leftarrow$ FALSE ;
    foreach values $a \in D_x$ do
        if there exists no $b \in D_y$ such that SATISFIES $(\{x = a, y = b\}, C_{x,y})$ then
            begin
                $D_x \leftarrow D_x - \{a\}$;
                DELETED $\leftarrow$ TRUE;
            end
    return DELETED ;
end
```

Chapter 3

Different Branching Strategies in a Search

This chapter consists of three sections, ‘Branching Strategies’, ‘Heuristics’ and ‘Related Work’. First, we start with explaining the general concept of what “branching strategy” means in *CSP*. In the second section, we discuss the details of how a variable can be selected in a search tree, and we also define this process as branch ordering. Lastly, we investigate some research in which branching strategies are employed.

3.1 Branching Strategies

In a backtracking search, many different ways to branch in a search tree have been proposed to reduce the search space by taking advantage of the structure of a particular problem. Here, we discuss only three of them; however, any combination of these three will also be possible provided that they can take advantage of the structure of a problem and lead into more effective pruning with the help of a propagator and a branching ordering heuristic.

First, k-Way branching instantiates each variable with all the possible values in the domain according to the statically or dynamically arranged order. Most search trees that appear in *CSP* are based on this strategy. The branching factor of a search tree with k-Way branching, b[1], is equal to d, the depth of the tree is $O(n)$ and the total number of

[1]Throughout this thesis, we follow the notation introduced in the Introduction: b is for the branching

nodes in the search tree becomes $O\left(d^n\right)$ or $O\left(b^n\right)$.

In order to explain how a backtrack search actually uses k-Way branching, Figure 3.1[2] is prepared in solving the 4-queens example with the aid of chronological backtracking[3] and the minimum domain size heuristic which is a branch ordering heuristic to pick a variable with the minimum number of values in its domain. This heuristic and other branching ordering heuristics will be discussed in detail in the next section. The number of nodes and the depth in this figure will not be exactly the same as the calculated numbers for a search tree since the figure is a search trace of the algorithm which terminates once the first solution is found. However, a search trace is employed here to give an overview of how a search algorithm can use different branching strategies.

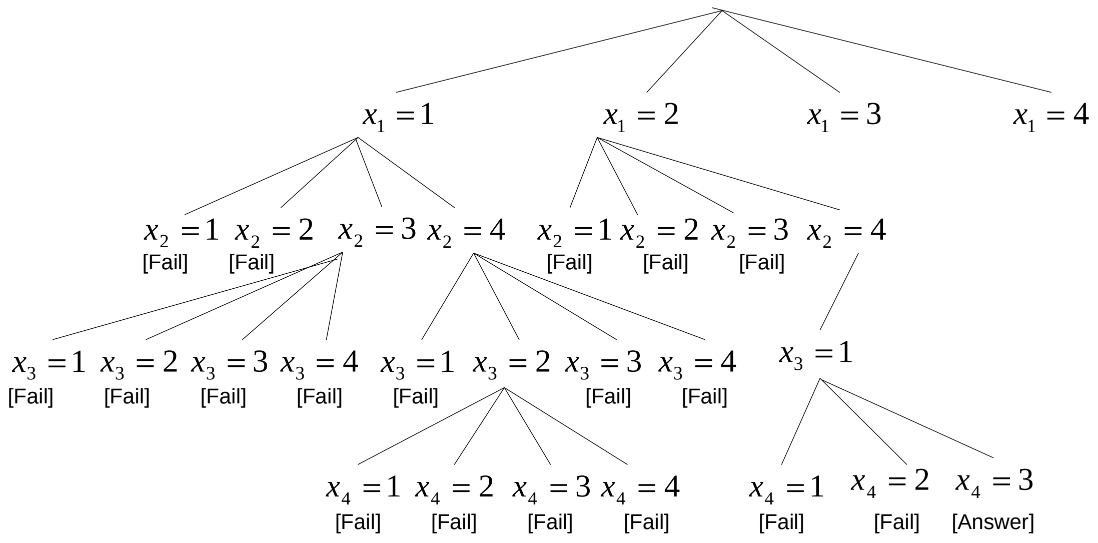

Figure 3.1: A search trace of chronological backtracking with k-Way branching in solving 4-queens problem

The search trace in Figure 3.1 illustrates that a chronological backtracking search with k-Way inspects 26 nodes in 4 levels to find the first solution for the 4-queens problem.

factor, n for the number of variables, and d for the maximum size of domains

[2]Figure 3.2 and 3.3 are also prepared with the same way as Figure 3.1.

[3]Chronological backtracking is described in the Background section

Furthermore, this figure also suggests that chronological backtracking with k-Way also generates a wider search tree than the backtracking with other branching strategies (shown in Figure 3.2 and 3.3) as the branching factor for k-Way is 4 and the others are only 2.

Unlike k-Way, the 2-Way branching strategy splits a domain of a variable into two sub-domains so that one sub-domain has an assigned value and the other takes the rest of values in the domain without the assigned value. The resulting search tree is a binary tree that has the depth of $O\left(d * n\right)$, and the number of nodes in the tree is $O\left(2^{d*n}\right)$, where a branching factor is 2. The overall size of the search tree for 2-Way is deeper and narrower than the search tree for k-Way.

In Figure 3.2, we present how a chronological backtracking search with 2-Way traces in solving 4-queens. 2-Way branching needs to examine 44 nodes in 10 levels. In comparison with the search trace for k-Way branching in Figure 3.1, 2-Way creates a deeper and narrower search trace.

Lastly, let us consider domain splitting, sometimes referred to as dichotomic search [2], that repeatedly splits the domains of variables into subdomains containing only some part of the original domain until a valuation domain is obtained. Each subdomain at the same level of a search tree is mutually exclusive, meaning there is no duplication of values in subdomains.

In domain splitting, we need to specify how a domain is split into subdomains. The most obvious example of a pivot point can be a middle point that separates the domain into two subdomains, the one with the lower half of the initial domain and the other with upper half [9]. Using a middle point as a pivot, the depth of the search tree for domain splitting becomes $O\left(n * \log d\right)$, and the upper bound of nodes in the tree is $O\left(2^{n*\log d}\right)$. This binary search tree for domain splitting also has a deeper and wider tree than k-way, but in comparison with 2-Way branching, domain splitting creates fewer nodes in less depth if the pivot is in the middle. There can also be many different ways to split a domain. The way to split domains can be like making subsets so that the ways to create subdomains can be as many as the number of subsets of a domain. As a matter of fact, 2-Way is a special form of domain splitting because 2-Way splits a domain in a predefined way where one branch contains a valuation domain and the other has a subdomain with the rest of the values.

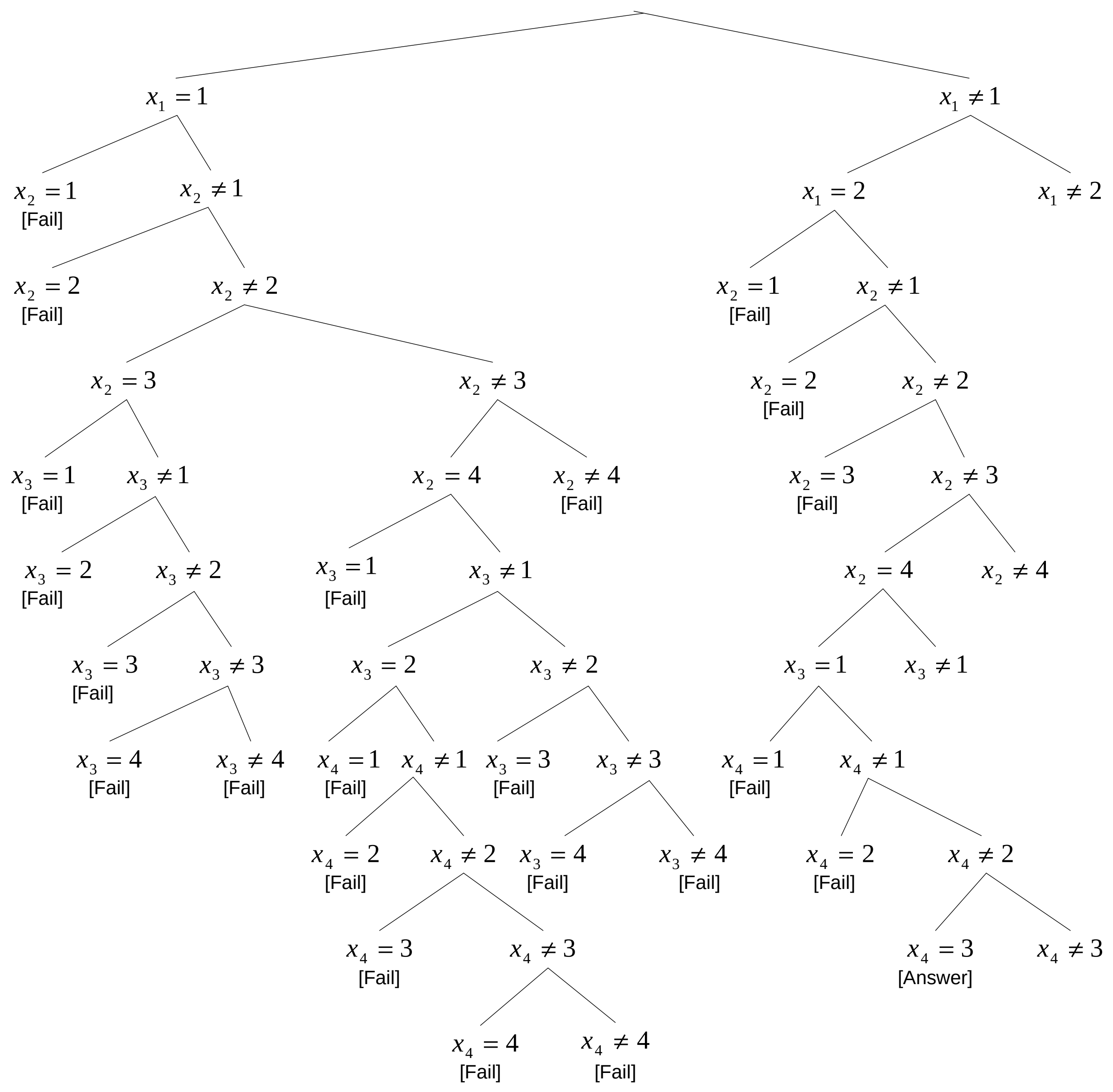

Figure 3.2: A search trace of chronological backtracking with 2-Way branching in solving 4-queens problem

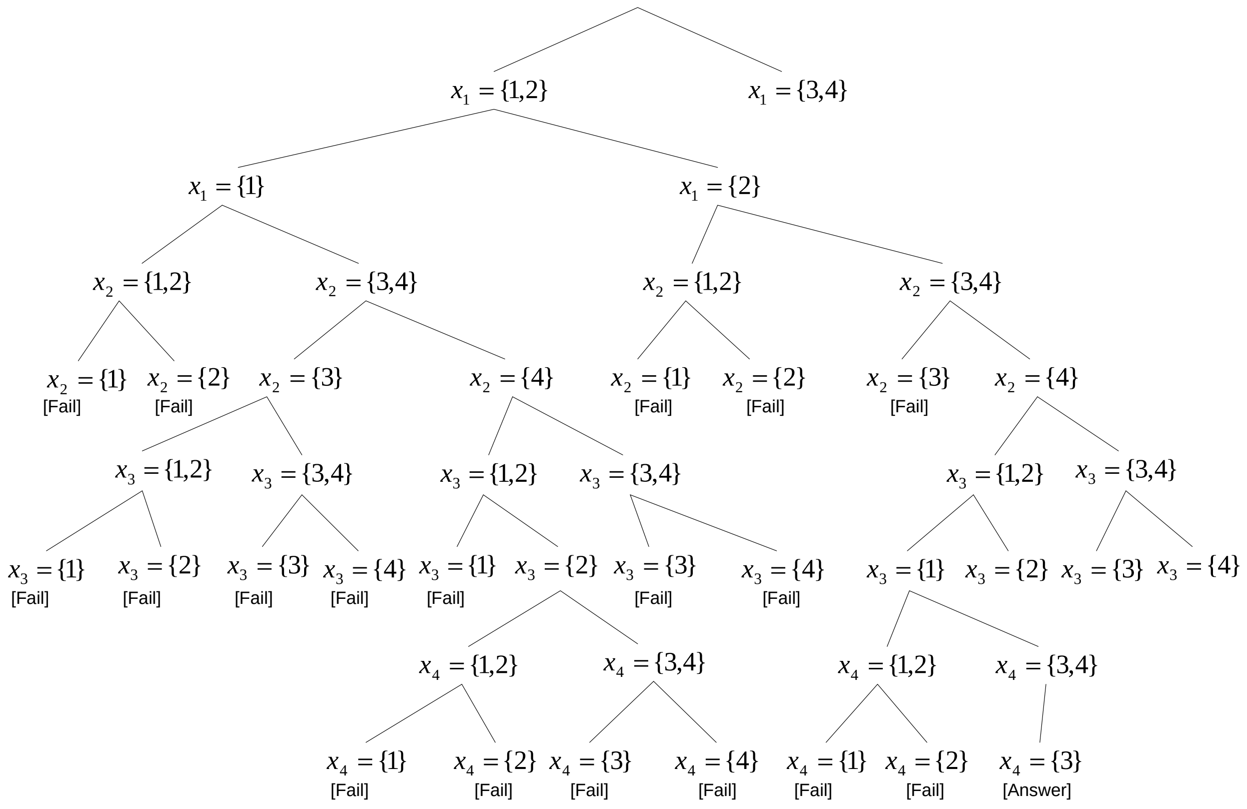

Figure 3.3: A search trace of chronological backtracking with domain splitting in solving 4-queens problem

The pictorial description of domain splitting that solves the 4-queens problem is illustrated in Figure 3.3. The chronological backtracking with domain splitting inspects 40 nodes in depth of 8 in order to find a solution, and this search trace is smaller than the one with 2-Way.

Even with the larger sizes of a search tree and a search trace, the usage of 2-Way branching and domain splitting methods can be justified by the principle of least commitment. This states that when making a choice for some variable we should make the choice which commits us as little as possible. The advantage is that, if we detect unsatisfiability after making a weak commitment, more of the search space is removed than if we detect unsatisfiability after a strong commitment. Moreover, according to [9], the branching strategies with least commitment can be more beneficial to use in an optimization problem since they can utilize more information to guide a search to obtain an optimal solution. In our test cases, there has been an instance that supports this conjecture.

As in k-Way branching, assigning a variable to a single value is too restrictive to contain much information. Domain splitting is less restrictive than k-Way and 2-Way branching strategies, and 2-Way is less restrictive than k-Way. Domain splitting only removes half of the remaining values in the variable's domain rather than all but one. 2-Way also carries more information in one side of branch than k-Way since one of the branches in 2-Way contains a subdomain rather than a single value. Therefore, the principle of least commitment tells us to prefer domain splitting over the other branching strategies and 2-Way over k-Way.

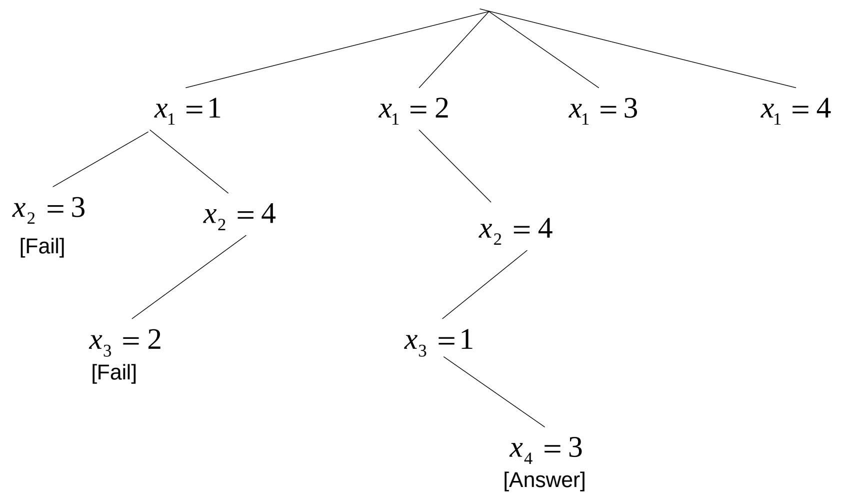

Figure 3.4: A search trace of backtracking with forward checking for k-Way branching in solving 4-queens problem

One of the apparent drawbacks of the least commitment principle is that less commitment may result in too much information for the incomplete solver to determine satisfiability or unsatisfiability. If a variable is least committed, this can be disadvantageous in propagation because a strong propagation like forward checking requires that a variable is committed to a value prior to propagation; 2-Way and domain splitting may not prune as much of a search space as k-Way branching does in the early stage of a search. However, they can have more freedom to choose the next variables and next values, which means they have more choice points[4] in a search tree. Moreover, they can utilize this information to reduce the amount of backtracking and to prune more effectively in the later stage of a search.

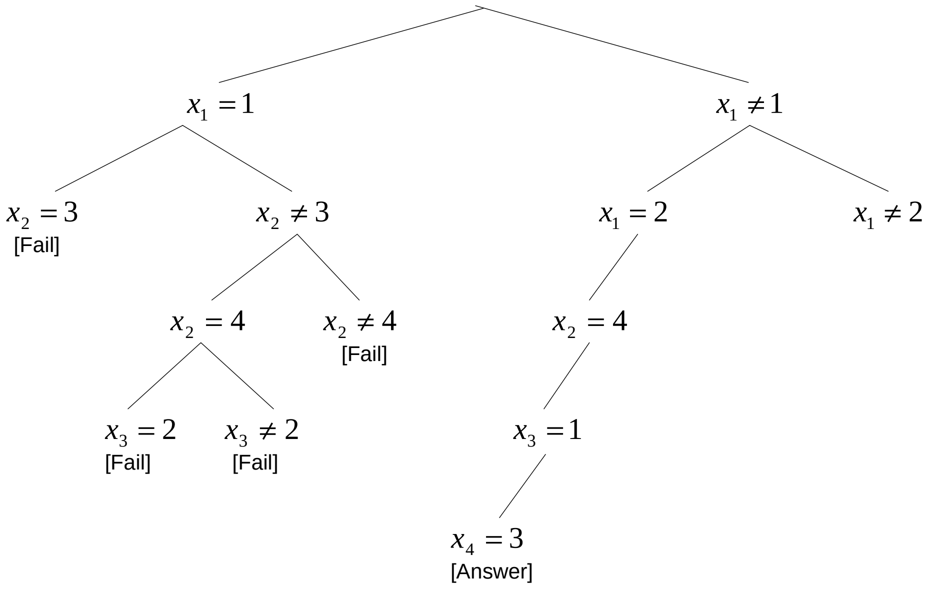

Figure 3.5: A search trace of backtracking with forward checking for 2-Way branching in solving 4-queens problem

In Figures 3.4, 3.5, and 3.6, we delineate how a search space can be pruned when different branching strategies are employed in a search algorithm interleaved with a pruning strategy, like a backtracking search with forward checking. The discrepancies among the search traces of different branching strategies can be reduced as shown in Figures 3.4, 3.5, and 3.6. The effectiveness of the filtering strategy on different branching methods can be easily observed when we compare Figures 3.4, 3.5, and 3.6 with Figures 3.1, 3.2, and 3.3. The sizes of the search traces generated by backtracking with forward checking

[4]We employ the definition of choice point given in [9]: a state that has two or more children.

that is employed with the three different branching methods become very similar to each other albeit there are 4 more nodes in 2-Way branching than in k-Way and 4 more than in domain splitting—k-Way has 8 nodes and a depth of 4, domain splitting has 9 nodes and a depth of 5, and 2-Way has 12 nodes and a depth of 5.

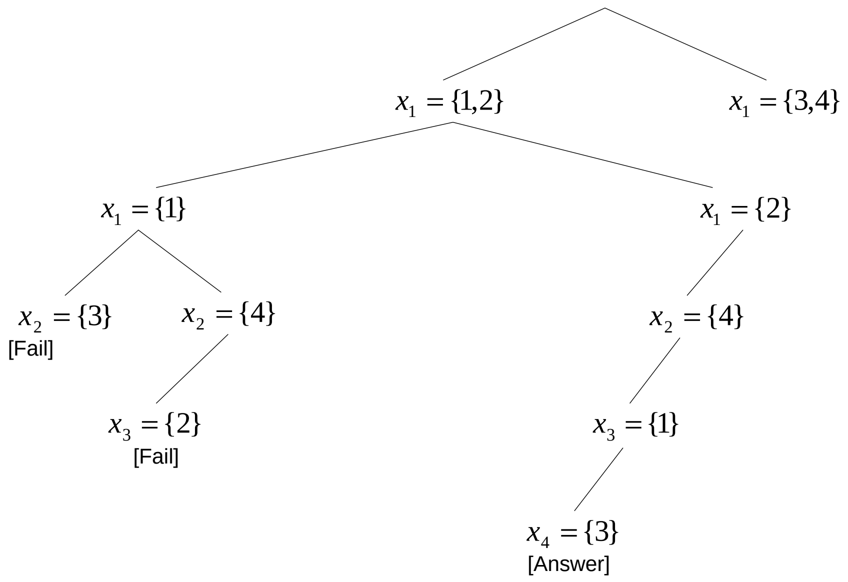

Figure 3.6: A search trace of backtracking with forward checking for domain splitting in solving 4-queens problem

The branching strategies with least commitment can be more efficient than the one with strong commitment provided that the constraints involving the unlabeled variables can provide substantial consistency information from the already split domain, despite the fact that domain splitting and 2-Way branching produce a larger search tree than k-Way branching. The reason behind this is that domain splitting may be able to eliminate half of a variable's possible domain values at once by adding a single constraint. This is an ideal case; however, unfortunately, we have not been able to observe this in our experiments.

About the power of expressiveness among different branching strategies, Mitchell shows that 2-Way branching can simulate k-Way branching but k-Way branching cannot express 2-Way [10]. Graphically, we show that a search tree of 2-Way branching in Figure 3.8

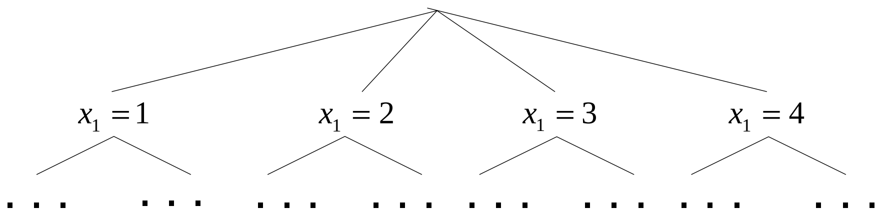

Figure 3.7: A part of a search tree for k-Way Branching in solving the 4-queens problem

simulates k-Way branching as presented in Figure 3.7 when solving the 4-queens problem.

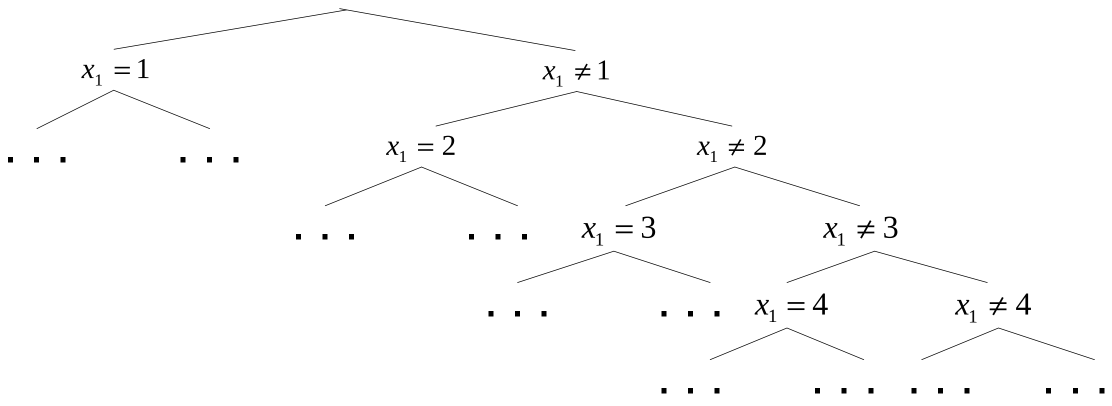

Figure 3.8: A part of a search tree for 2-Way branching that simulates k-Way branching

Since 2-Way branching strategy is a special form of domain splitting, we can induce from Mitchell's claim that domain splitting can also simulate 2-Way branching and therefore, k-Way as well. We give a pictorial description of how domain splitting can simulate 2-Way branching in Figure 3.9. Consequently, it is safe to conclude that the domain splitting strategy is the most expressive among three strategies, followed by 2-Way branching and, lastly, k-Way branching.

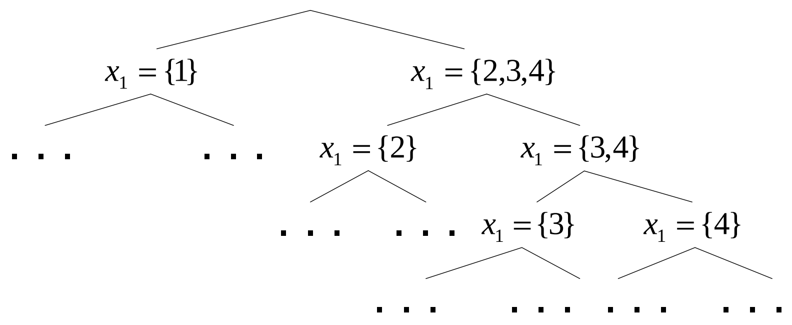

Figure 3.9: A part of a search tree for domain splitting that simulates 2-Way branching

3.2 Heuristics: Branch Ordering Heuristics

There are two choices to be made when a search algorithm assigns a value to a variable: the order in which variables are assigned and the order in which the values for a particular variable are explored [9]. In this thesis, we only focus on variable ordering heuristics. Although a value ordering heuristic specifies how to explore the search tree, altering the order in which variables are instantiated can have more dramatic effects on the size of a search tree [9] as some choices of variables can lead into pruning a search space more effectively than the other choices in propagation.

One of the most widely used branch ordering heuristics is based on the principle called **Fail First** (*FF*). According to this principle saying, "to succeed, try first where you are most likely to fail [4]", it guides a search to choose a variable with fewer choices. Therefore, it is advisable that the variables with many possible values are instantiated after those with fewer values.

Based on *FF*, the minimum domain size ordering heuristic is proposed [9]. This heuristic claims that choosing the variable with the smallest number of values in its domain allows the potential failure of the entire branch to be determined quickly, thus guiding the search to more profitable areas. A further motivation for this strategy is that, with luck, once these variables are assigned a value, propagation will be able to trim the domains of the remaining variables [9]. In much of the literature, there seems to be a consensus that this heuristic seems to be very efficient, especially when the size of a problem is large.

Unlike the minimum domain size branch ordering, the maximum domain size branch ordering heuristic prefers the variable with the maximum domain size; this works in the exactly opposite way to the minimum domain size ordering. This heuristic is to choose a variable with the most choices; therefore, a search is more likely to find a solution with fewer fails. However, empirical studies show that the maximum domain size ordering works worse in practice than the minimum domain size branching ordering in most cases [9].

Both the minimum domain size and maximum domain size ordering heuristics can choose a variable either dynamically and statically[5]. When a branching ordering is static, the order of variables is predefined before a search starts. In dynamic branch ordering, a search algorithm updates the domains of variables during the search and a variable is chosen based on the current domain information.

There can be other dynamic branch ordering heuristics like the greatest minimal bound, greatest maximal bound, least minimal bound, and least maximal bound branch ordering strategies. These heuristics work as follows:

- Greatest minimal bound variable ordering picks the variable with the largest value in the lower bound of domain

- Least minimal bound variable ordering chooses the variable with the smallest value in the lower bound of domain

- Greatest maximal bound variable ordering prefers the variable with the largest value in the upper bound of domain

- Least maximal bound variable ordering first assigns the variable with the smallest value in the upper bound of domain

It is also an important task to decide the order in which values are tried [14]. Ordering a value in a domain will only change the order in which solutions are found. In other words, a value ordering specifies the order in which a search explores a search tree. In order to find a solution for a *CSP* more quickly, it seems a good heuristic to first assign the least constraining value to a variable. This heuristic is based on the assumption that the value which least reduces the domains of other variables is more likely to be a part

[5]In some literature, non-deterministically means dynamically and deterministically refers to statically.

of a solution. Also, there have been other value ordering heuristics proposed to exploit the problem-specific knowledge in hope of picking a value that is more likely to lead to a solution so that the computational time for a search to find a (first) solution improves. Ordering of domain values can be also useful in evaluating optimization goals. In this case, the values in domain are arranged so that the optimum solution can be found in the earlier stage of a search.

3.3 Related Work

When solving numeric *CSPs*[6], researchers seem to prefer domain splitting over the order branching strategies [7]. Jussien and Lhomme claim that using a search called dynamic domain splitting they were able to dramatically improve the search time for numeric *CSPs*.

Dynamic domain splitting is based on dynamic backtracking. Dynamic backtracking relies on reducing backtracking [7] by utilizing the information of no-goods in case of failure and unassigning the invalid assignment without modifying any other assignments. We can think of dynamic backtracking as backtracking in a search space rather than in a search tree [7]. Domain splitting is employed to add a constraint to eliminate the failed assignment rather than to unassign a variable when a failure occurs.

It is interesting to see that Jussien and Lhomme have been able to improve the running time with the aid of domain splitting on a continuous domain although they warn the reader that the result is not comprehensive and in order to find out the effectiveness of domain splitting, experiments with a larger scale need to be done [7].

As for finite domain problems, k-Way branching seems to be almost exclusively used. However, Milano and van Hoeve introduce a search strategy called Decomposition Based Search *(DBS)* [15], where a domain splitting strategy is employed to break down a finite domain problem into subproblems. Their research is also particularly of interest to us since they tested their algorithm with the aid of the ILOG Solver as we have done in our experiments.

The unique features in their research are that *DBS* is based on two steps, subproblem generation and subproblem solution, and domain splitting is only employed in subproblem

[6]Numeric *CSPs* are the constraint satisfaction problems that have a non-finite domain size.

generation [15]. Domain splitting is employed to decompose a problem into subproblems with smaller subdomains, and the values in a domain are grouped into subdomains according to a value ordering heuristic. The heuristic needs to rank a value with two levels of accuracy: first, it should measure accurately how successful a value is; second, it is required to discriminate among values with the same rank [15].

In a search, after a certain level that is defined by a user in a pre-search stage, domain splitting is not employed to separate domains, but a search selector is instead used to instantiate a variable. Although they mention that Depth-First Search that we have used in our experiments can be a good choice as a search selector because it is usually much faster than any specialized search strategies when subproblems are equally likely to be successful, they employ Limited Discrepancy Search (*LDS*) to exploit the ranks that the value ordering heuristic assigns to a value in the subproblem generation stage [15].

As test cases, they have tested *DBS* on two finite domain *CSPs*: the traveling salesman problem as an optimization problem and the partial Latin square completion problem as a satisfiable problem. They have analyzed test data of *DBS* by comparing with the ones for a search without domain splitting, *LDS*, and they have reported that in both problems, *DBS* is faster than *LDS* on average [15]. Furthermore, they also have made an interesting observation saying that with the help of the pruning effect of a primitive constraint like `alldiff`, the efficiency of domain splitting drops so that *LDS* becomes faster than *DBS* [15]. This observation is in agreement with our finding as we exploit the pruning power of the addiff constraint in our experiments.

Albeit no one has compared the effectiveness of different branching strategies in the finite domain problem, Mitchell describes the expressive power of different branching strategies in [10]. He uses resolution-like proof systems for finite-domain constraint satisfaction problems in order to prove that 2-Way branching can simulate k-Way branching, but the converse does not hold.

The proof systems employed in [10] are ***NG-RES*** that is suggested in de Kleer's study and ***C-RES*** that is introduced by Baker. With these proof systems, Mitchell explains the expressiveness of k-Way and 2-Way branching. First he shows that 2-Way and k-Way branching strategies can be expressed as C-RES and NG-RES respectively, and then he claims that 2-Way branching can simulate k-Way since the proof system C-RES can simulate NG-RES. In addition, he also points out that k-Way branching cannot simulate 2-Way because NG-RES cannot be translated into C-RES.

In our empirical studies, we have also gathered some data that give the evidence hat 2-Way branching indeed simulates k-Way branching; furthermore, with close observation, we find out that 2-Way branching can be considered as a special form of domain splitting that splits a domain into two in a way that one branch has a single value and the other has the rest of the values.

Chapter 4

Experimental Setup

In this chapter, we explain how to model and implement different *CSPs*. In Section 4.1, we classify the problems used in this experiment and elucidate the modeling process before explaining the implementation. In Section 4.2, we show how the problems are solved in the ILOG environment.

4.1 Problems

In order to compare the effectiveness of different branching strategies in solving *CSPs*, we carefully chose the following problems: the N-queens, the car sequencing, and uniform random binary problems as satisfiable problems and the Golomb ruler, the instruction scheduling, and the truck routing problems as optimization problems.

4.1.1 The N-queens Problem

The N-queens problem is chosen because it is a well-known finite domain problem as well as an arithmetic CSP. The N-queens problem involves placing N queens on an $N \times N$ chessboard in such a way that no queen can capture any other queens using the conventional chess moves allowed to a queen. In other words, the problem is to select N squares on a chessboard so that any pair of selected squares is never aligned vertically, horizontally, nor diagonally [6].

One way of modeling this problem is to recognize that in any solution there is exactly one queen in every column. For example, the variable x_i represents the column number of the queen in the i^{th} column. These $x_i, 1 \leq i \leq N$, correspond to the column variables. Next, each column variable is given an initial domain, D, containing integers between 1 and N which signify the row numbers; i.e., $dom(x_i) = \{1, \ldots, N\}$. Then constraints, C, are specified to ensure that no queen falls on the same row or diagonal as any other queen. Recognizing the constraints of the problem can be stated in the following way. For every pair of variables, (x_i, x_j), where i is different from j, a constraint $x_i \neq x_j$ guarantees that each column have a distinct row value; and constraints, $x_i + i \neq x_j + j$ and $x_i - i \neq x_j - j$, together make sure that the diagonals are distinct. In the ILOG setting as shown in Figure 4.1, the primitive constraint such as `alldiff` can be used to ensure the inequality among variables. For example, `alldiff(x)` ensures that every x is different as in $x_i \neq x_j$. Similarly `alldiff(`$x_i - i$`)` makes sure that the inequality, $x_i - i \neq x_j - j$, holds so does `alldiff(`$x_i + i$`)` satisfy $x_i + i \neq x_j + j$. When we compare this modeling with the definition given in the 'Background' section, we can easily notice that the modeling specifies how a *CSP* is defined.

4.1.2 The Golomb Ruler Problem

For similar reasons as the N-queens problem, the Golomb ruler problem is selected. A Golomb ruler is a set of non-negative integers such that the differences of any two distinct pairs of the numbers from the set are not the same. This can be conceptually viewed as a ruler constructed in a way that any two distinct marks in the ruler do not measure the same distance. The Golomb ruler problem is the optimization problem of finding the shortest Golomb ruler possible for a given number of marks.

With the same manner as how the N-queens are modeled, the Golomb ruler for N marks is modeled as a *CSP*. The variable, $x_i, 1 \leq i \leq N$, represents the location of the i^{th} mark in a ruler. The variables take a value from the domain, $D = \{d | 1 \leq d \leq N^2\}$. The constraints, C, ensure the difference in length between any pair of marks in a ruler to be distinct as follows:

- All the variables are arranged in ascending order,$x_1 < x_2 < \ldots < x_N$.

- All the differences of any combinations of two variables in the set are distinct, $x_k - x_l \neq x_m - x_p, 1 \leq k, l, m, p \leq N$.

One of the differences between the N-queens and the Golomb ruler problem is that the Golomb ruler problem is required to find the optimal solution while N-queens is solved once a solution is found. In *CSPs*, an objective function is introduced to evaluate solutions and find the optimum. Usually, a minimization function is employed to select the optimal solution that has the minimum value of the function. For instance, the Golomb ruler uses the function of the last mark, x_N , to evaluate a solution, meaning that the smaller the value of x_N, the shorter the ruler becomes; therefore, the objective function for the Golomb ruler can be written as $f(x_n) = x_n$.

4.1.3 The Car Sequencing Problem

Since the car sequencing problem is a real-life problem that frequently arises on assembly lines in factories in the automotive industry, this problem is chosen[6]. [6] describes the problem in a follow way: An assembly line makes it possible to build many different types of cars, where the types correspond to a basic model with selected options. In that context, one type of vehicle can be seen as a particular configuration of options. It is also possible to put multiple options on the same vehicle while it is on the assembly line so any combination of options is possible to be manufactured on the assembly line. However, because of the physical limitations such as the amount of time needed to install certain options, a particular option cannot be installed on every vehicle on the line. This constraint is defined by what we call the “capacity” of an option. The capacity of an option is usually represented as a ratio p/q where for any sequence of q cars on the line, at most p of them will have that option. The problem in car sequencing then consists of determining in which order cars corresponding to each configuration should be assembled, while keeping in mind that we must build a certain number of cars with the desired number of configurations.

For example, the problem is specified by 10 cars to build, 5 options available for installation, and 6 configurations required. Table 4.1.3 indicates which options belong to which configuration: if $\clubsuit$ is found in the cell of option i and configuration j, it indicates that configuration j requires option i: a blank means configuration j does not require option i.

option	**capacity**	**configurations**					
		0	1	2	3	4	5
0	1/2	♣				♣	♣
1	2/3			♣	♣		♣
2	1/3	♣				♣	
3	2/5	♣	♣		♣		
4	1/5			♣			
number of cars		1	1	2	2	2	2

Table 4.1: The example of configurations in the car sequencing problem (taken from [6])

The table also shows the capacity of each option as well as the number of cars to build for each configuration.

For example, the chart in the table indicates that option 1 can be put on at most two cars for any sequence of three cars. Option 1 is required by configurations 2, 3, and 5.

4.1.4 The Truck Scheduling Problem *(TSP)*

The truck scheduling problem can be seen as the traveling salesman problem, and because it is a typical example of optimization problems, we consider this problem. For the truck scheduling problem, we employ the problem specification given in [6] to model and implement. Suppose a truck driver has n customers to visit to deliver goods. The driver knows where each customer is located and thus the distance to travel from one to another. We further assume that the driver can start with any customer, but the tour has to end at the location where the tour started. The driver may visit a customer in any order but cannot visit any customer more than once. In other words, we want to find a tour (a cycle) visiting all the customers. The goal of this problem is to minimize the total distance traveled to complete this tour.

The problem is modeled in the following manner [6]:

- The starting point of the tour can be chosen arbitrarily since the driver has to visit all the locations anyway; the starting point also becomes the ending point and $n-1$

nodes represent the customers that neither start nor end the tour.

- Two arrays are introduced to measure the distance traveled, `Distance` and `Length`; `Distance[i][j]` gives the distance between customer i and customer j and `Length[i]` keeps the total distance traveled from start to customer i.
- The array called `Next` is used to keep neighborhood relations; `Next[i]` stores the neighbors of customer i.

In other words to describe the problem, a problem is to find a shortest path via all the nodes representing customers in a graph. Then, the nodes in the graph have the following characteristics: first, the *end* node that represents the last customer has the value n in a graph and it is used to index the above arrays; second, the *start* node where the tour begins and ends is indexed as $n+1$; third, the other nodes i are indexed as i. Then the goal of the problem is to minimize `Length[end]`.

4.1.5 The Instruction Scheduling Problem

Because instruction scheduling is one of the most important steps for improving the performance of object code produced by a compiler [13], we include this problem in our experiment. The local instruction scheduling problem is to find a minimum length instruction schedule for a basic block subject to precedence, latency (delay), and resource constraints [13]. We follow the experimental design set in [13], where local instruction scheduling for single-issue processors with arbitrary latencies are implemented.

[13] defines the local instruction scheduling problem as follows: The standard labeled directed acyclic graph *(DAG)* is used to represent a basic-block, where each node corresponds to an instruction. $l\,(i,j)$ indicates the latency needed between the instructions i and j. For example, if an instruction j must not be executed until i has executed for $l\,(i,j)$ cycles, then there is an edge from i to j labeled with a positive integer $l\,(i,j)$.

Given a labeled dependency DAG $G=(N,E)$ for a basic-block, where N is a set of nodes and E is a set of edges connecting nodes. We also need an array S to specify the start times for the instructions. If we define $S\,[i]$ for the start time of the instruction i, we can derive the following constraints for the problem:

- No two instructions are issued simultaneously, $S[i] \neq S[j], i, j \in N, i \neq j$.
- The start time of an instruction depends upon the start times and latencies of its predecessors, $S[j] \geq S[i] + l(i,j), (i,j) \in E$

The goal of the local instruction scheduling problem is to minimize the total execution time of the instructions and the objective function is to minimize $max\{S[i] | i \in N\}$

4.1.6 The Uniform Binary Random Problem

Many *CSP* researchers use random uniform instances to evaluate their constraint satisfaction algorithms [1]. In spite of the fact that it is generally agreed that it is more important to test the performance of their algorithms on real-world problems [1], Random problems offer the following advantages for empirically evaluating the performance of CSP algorithms [1]:

1. easy to study quantitatively,
2. easy to modify the parameters of problems,
3. easy to find the phase transition point,
4. easy to reproduce the data, and
5. easy to share with other researchers.

A random generator is employed in generating a random problem to produce uniform and binary instances. It uses extensionally represented constraints, meaning that it enumerates invalid assignments between a pair of variables as a constraint. The randomness is exploited in generating constraints, and `rand2()` is employed to guarantee the randomness [1]. Hence, each variable is equally likely to be selected in `rand2()`; so, the constraints are uniformly distributed among variables.

4.2 Implementation

This section explains how a software package like the ILOG solver can be used in implementing the search algorithm and the different heuristics to guide a search in solving *CSPs* after modeling. Henceforth, we follow the specification of ILOG codes given in [6, 5].

For instance, the N-queens problem can easily be translated with the aid of built-in ILOG commands embedded in the *C++* language. To illustrate better, the code in Figure 4.1 is used as an example:

```
void main(int argc, char** argv) {
IlcManager m(IlcEdit);
IlcInt nqueen = (argc > 1) ? atoi(argv[1]) : 1000;
IlcIntVarArray x(m, nqueen, 0, nqueen-1),
           x1(m, nqueen), x2(m, nqueen);
for (i=0; i<nqueen; i++) {
x1[i] = x[i] + i;
x2[i] = x[i] - i;
}
m.add(IlcAllDiff(x));
m.add(IlcAllDiff(x1));
m.add(IlcAllDiff(x2));
m.add(IlcGenerate(x, IlcChooseMinSize));
if (m.nextSolution()) {
              for ( i=0; i < nqueen ; i++)
                 m.out()<<x[i].getValue() << " ";
            }
            m.end();
            return 0;
}
```

Figure 4.1: ILOG code for solving N-queens problems from [6]

In ILOG version 4.2, the constructor of the `IlcManager` class (as in Figure 4.1) has to be called before defining constraints and generating a solution. When an instance of `IlcManager` is created, it initializes internal data for the solver. That instance, known as a manager, then handles input and output, memory allocation, and other general services for all the constrained variables constructed for that instance and for all the constraints and goals added to that instance. `IlcManager` can take one of two different arguments, namely `IlcEdit` and `IlcNoEdit`. In the `IlcNoEdit` mode, `IlcManager::add` immediately posts and propagates constraints; however, the `IlcEdit` mode delays the propagation after `IlcManager::nextSolution` is called. Since we want to postpone the propagation until all the constraints are gathered, we prefer the `IlcEdit` mode and all the experiments are carried out in this mode.

ILOG also enables users to define variables for the solver [11]. `IlcInt` type is declared for the `nqueen` variable (as defined in Figure 4.1) that takes an integer value for N in N-queens problem. Programmers need to declare a variable as `IlcIntVar` if the variable takes `IlcInt` values. `IlcIntVarArray` can also be used to declare the array of constrained integer variables that represents a list of N distinct column variables ranging between 0 and $N-1$.

Using built-in primitive `IlcAllDiff` for the `alldiff` constraint (as illustrated in Figure 4.1), `IlcManager::add` adds constraints through the arrays of the column variables from 1 to N in order to make sure that each queen does not fall on the same row or diagonal as the remaining queens in the list. The reason for using primitive constraints rather than specifying individually the relation among column variables is that it is more efficient since a solver usually provides special propagation rules for its own primitives [9].

After setting up constraints, the labeling command, `IlcGenerate` (as in Figure 4.1), is invoked to make sure that each variable is assigned to a value before looking for a solution. `IlcGenerate` specifies the type of branching strategies; moreover, `IlcGenerate` (as implemented in Figure 4.1) is employed for k-Way branching since the default type for `IlcGenerate` in ILOG 4.2 is k-Way branching. Later in Figure 4.3, 4.4, and 4.5, we also show the implementations of the other branching strategies.

To generate a solution, `IlcManager::nextSolution` needs to be called as in Figure 4.1. The first time `nextSolution` is used; it iteratively pops one goal from the goal stack and executes it [5]. The execution of the goal can add other goals to the stack and can set choice points. The execution of `nextSolution` terminates in two cases. First, when the goal stack becomes empty, the function returns `IlcTrue`. Second, if a failure occurs and no choice point with untried subgoals and correct labels exists, the function restores the state of the invoking manager and returns `IlcFalse`.

At the end of a program, `IlcManager::end` is called as in Figure 4.1 to clean up the Solver memory allocations associated with the invoking manager.

For the optimization problems, the objective function can be implemented with the built-in predicate called `setObjMin` to find an optimal solution. `IlcManager::setObjMin` (refer to Figure 4.2) takes a variable as its arguments [5]. This variable becomes the objective to be minimized. This member function changes the behavior of `nextSolution` (as in Figure 4.2) so that each time `nextSolution` is called, the new solution will yield a better value for the cost variable. The last solution found will be the optimal solution.

```
//ILOG objective function built-in primitive
//Toggling the positive or negative sign enables users to solve
//the minimization or maximization problems without great effort
m.setObjMin(x[n-1]);
while(m.nextSolution()) {
...
//Keep enumerating the solutions as long as the current objective
//function value is better than the previous one, and
//the objective function value is updated to the current one.
//
}
```

Figure 4.2: ILOG code segment for the objective function

ILOG also provides several built-in predicates to implement different branching strategies. `IlcGoal IlcGenerate (const IlcIntVarArray, IlcChooseIntIndex)` is used as a default predicate to instantiate a value for a variable. `IlcGenerate` simulates the backtracking strategy, k-Way branching, and this predicate repeatedly chooses a variable, assigns a value to the variable, and checks the consistency of the current assignment until every variable is instantiated.

The ILOG manual explains the execution of `IlcGenerate` as follows [6]: `IlcGenerate` calls `IlcInstantiate` to create and return a goal in the algorithms searching for solutions. The `IlcInstantiate` assigns a value to a constrained variable. It uses choice points so that if a failure occurs as a result of that reversible assignment, another value will be assigned to the constrained variable so that the search can continue. If a variable has already been bound, meaning it has an evaluation domain, the `IlcInstantiate` does nothing and succeeds. Otherwise, `IlcInstantiate` sets a choice point, and assigns a value to the constrained variable. In case of failure, the “tried-and-failed” value is removed from the domain of the constrained variable, and another value not yet used is tried until a value assignment succeeds or the domain is exhausted. In that latter case, the domain becomes empty, and the member function IlcManager::fail is called. Different branch ordering heuristics can be employed by specifying `IlcChooseIndex` parameter. If `IlcChooseIndex` is not set, variables are tried by default in lexicographical order.

The other branching strategies can readily be implemented in the ILOG setting. The primitive called `GOAL ILCGOALn` (n stands for the number of variables) is used to define 2-Way branching and domain splitting. In the program shown in Figure 4.3, `ILCGOAL1` defines the procedure called `Instantiate` and takes the other variable type `IlcIntVar` which specifies the type of argument that `Instantiate` can take (refer to Figure 4.3). First, it checks whether a variable is bound or not before committing a variable to a value, and if the variable has more than one value in its domain, the procedure will assign the minimum value in the domain. `IlcOr` is a logical ILOG operator, meaning “or,”and so does `IlcAnd` mean “and.” The logical connectives are used to iterate `Instantiate` until all the values for variables are exhaustively tried. The following code illustrates how ILOG implements 2-Way branching.

```
ILCGOAL1(Instantiate, IlcIntVar, var) {
  if (var.isBound())
     return 0;  //if variable is bound, do nothing
  //Otherwise
  else {
    IlcInt val = var.getMin();
    return IlcOr(var == val, IlcAnd(var != val, this));
  }
}
```

Figure 4.3: ILOG code for 2-Way branching

As for domain splitting, it dynamically splits the domains of variables into two subdomains by pivoting a middle point. In a similar manner as how 2-Way branching is implemented, domain splitting can be defined using `ILCGOAL1`. Figure 4.4 shows the ILOG implementation of domain splitting. It specifies the procedure named `Dichotomize` (as in Figure 4.4) and it repeatedly splits the domain into two halves rather than assigning with the minimum value as in 2-Way branching.

Later upon calling the procedure `ILCGOAL2`, users can specify their branching strategy to be either 2-Way branching or domain splitting. `ILCGOAL2` takes three arguments as elucidated in Figure 4.5: first, the name of the procedure that it defines, `Generate`; second, the type of argument for the procedure, `IlcIntVarArray`; third, the parameter for variable ordering heuristics, `IlcChooseIntIndex`. `Generate` is an enumeration algorithm that guarantees to exhaustively try all values in the variables's domains and find the legal assignment if there is one. In order to bind each constrained variable, `Generate` can employ different branching strategies that a user specifies. If it is 2-Way branching, we need to return `IlcAnd(Instantiate(getManager(),vars[index]),this)`, and as for domain splitting, we instead use `return IlcAnd(dichotomize(getManage(),vars[index],this)` as a return statement.

`Generate` also provides the parameter that controls the order in which variables an values are tried during the search for a solution. This parameter for controlling the choice of variables is set to direct the search toward productive branches of the tree for a solution and to eliminate useless branches as early as possible in the search process. ILOG supplies different built-in predicates for `IlcChooseIndex` to specify the variable ordering heuristic. Some examples of `IlcChooseIndex` are shown in Table 4.2.

```
ILCGOAL1(Dichotomize, IlcIntVar, var) {
  if (var.isBound())
     return 0;  //if variable is bound, do nothing
  //Otherwise
  else {
    IlcInt Val = (var.getMin() + var.getMax())/2;
    return IlcOr(var <= val, IlcAnd(var > val, this));
  }
}
```

Figure 4.4: ILOG code for domain splitting

```
ILCGOAL2(Generate,
         IlcIntVarArray, vars,
         IlcChooseIntIndex, chooseIndex) {
  IlcInt index = chooseIndex(vars);
  if (index == -1)
     return 0; //If there is not a variable left, do nothing
  //Otherwise
  else {
     //Specify the branch strategy
     return IlcAnd(Instantiate(getManager(),vars[index]),this);
     //return IlcAnd(dichotomize(getManage(),vars[index],this);
  }
}
```

Figure 4.5: ILOG code for an enumeration algorithm called Generate

Branch Ordering Heuristics	Description
Lex (Lexicographical Ordering)	It chooses the first unbound variable in the lexicographical order. This is used as a default.
MinSize (Minimum Domain Size Ordering)	It chooses the variable with the domain that has the least cardinality. That is to select the variable with the smallest domain based on the Fail First principle.
MaxSize (Maximum Domain Size Ordering)	It chooses the variable with the domain that has the greatest cardinality. That is to select the variable with the largest domain.
MinMin (Least Minimal Bound Ordering)	It chooses the variable with the least minimal bound. That is to select the variable with the smallest value of the lower bound in the domains.
MaxMin (Greatest Minimal Bound Ordering)	It chooses the variable with the greatest minimal bound. That is to select the variable with the largest value of the lower bound in the domains.
MinMax (Least Maximal Bound Ordering)	It chooses the variable with the least maximal bound. That is to select the variable with the smallest value of the upper bound in the domains.
MaxMax (Greatest Maximal Bound Ordering)	It chooses the variable with the greatest maximal bound. That is to select the variable with the largest value of the upper bound in the domains.

Table 4.2: Table of different branching ordering strategies implemented in ILOG

Chapter 5

Experimental Results

In this chapter, we discuss the test results from the experiments designed and carried out to compare the efficiency of different branching strategies in combination with currently most used branch ordering heuristics when solving finite domain problems. The N-queens problems are presented in Section 5.1, the Golomb ruler problems in Section 5.2, the instruction scheduling problems in Section 5.3, the car sequencing problems in Section 5.4, the truck scheduling problems in Section 5.5, and the uniform binary random problems in Section 5.6. All the data were prepared using Visual C++ version 6.0 compiler and ILOG constraint solver library version 4.2 on a system equipped with an Intel Pentium 4 processor and 512MB of memory. In presenting the experimental results, we present data to only one decimal point, because the system timing routines are not accurate to more digits of precision. Since we are interested in how fast branching strategies are in solving different types of problems, we put an emphasis on the analysis of the execution time; however, the number of choice points is used to elucidate some behavior of different branching strategies if it is necessary.

5.1 Results for the N-queens Problems

First, different branch ordering heuristics are tested in placing 20 queens on the chessboard. The results are shown in Table 5.1. By carefully observing the table, one can find minimum domain size (MinSize) and greatest minimal bound (MaxMin) orderings are effective in solving 20-queens problems. On the order hand, maximum domain size (MaxSize) and greatest maximal bound (MaxMax) ordering heuristics are very ineffective especially when they are combined with the domain splitting strategy; as a matter of fact, they were incomplete in the sense that the program did not halt within a reasonable amount of time. We set the time limit as 100 seconds for the 20-queens problem and in both occasions, they were not able to find a solution within 100 seconds.

The Execution Time (in seconds)	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	1.1	0.0	0.3	0.1	0.0	1.5	2.0
2-Way Branching	1.5	0.0	0.3	0.1	0.0	2.3	1.9
Domain Splitting	1.5	0.0	>100	1.2	0.0	1.6	>100

Table 5.1: The execution time of different branching strategies for the 20-queens problem

With the aid of the minimum domain size and greatest minimal bound orderings, we carried out the tests for larger N-queens problems up to $N = 1500$. Tables 5.2 and 5.3 show the results respectively. In Table 5.2, we report the behavior of the running time of the program with MinSize as the size of the problem increases. However, the experiments with MaxMin, as shown in Table 5.3, does not reveal any useful information since the program did not find the answer within 100 seconds when N is larger than 50. Among different branch ordering heuristics that we have tested, the minimum domain size ordering heuristic seems to be the most beneficial in solving larger N-queens.

For the N-queens problems, k-Way branching and 2-Way branching outperform domain splitting and the difference between them increases as the number of queens increases. When N is 1500, k-Way and 2-Way become two times faster than domain splitting (refer to Table 5.2).

On top of this, there is another interesting point that is worthwhile to make here. The values of k-Way branching and 2-Way branching seem to be identical and this leads us to suspect that 2-Way branching might simulate k-Way branch when solving the N-queens problems. As discussed in Mitchell's paper [10], 2-Way branching can simulate k-Way branching; however, k-Way cannot simulate 2-Way. Also when we compare the choice points of k-Way and 2-Way branching, there is another empirical evidence that 2-Way ends up simulating k-Way branching because the traces of choice points set by these two branching strategies are exactly the same in Table 5.4. Besides the running time of a program, the number of choice points can also be a good means to decide the efficiency of a heuristic. This is because for the same problem, the size of the derivation tree is roughly proportional to the number of choices in the tree [9].

The Execution Time (in seconds)	$N = 10$	$N = 50$	$N = 100$	$N = 1000$	$N = 1500$
k-Way Branching	0	0	0.0	3.0	8.3
2-Way Branching	0	0	0.0	3.0	8.3
Domain Splitting	0.0	0.0	0.0	3.1	15.412

Table 5.2: The running time of solving the N-queens problems with MinSize

The Execution Time (in seconds)	$N = 10$	$N = 50$	$N = 100$	$N = 1000$	$N = 1500$
k-Way Branching	0	> 100	> 100	> 100	> 100
2-Way Branching	0	> 100	> 100	> 100	> 100
Domain Splitting	0	> 100	> 100	> 100	> 100

Table 5.3: The running time of solving the N-queens problems with MaxMin

The Number of Choice Points	$N = 10$	$N = 50$	$N = 100$	$N = 1000$	$N = 1500$
k-Way Branching	11	50	95	996	1492
2-Way Branching	11	50	95	996	1492
Domain Splitting	22	652	392	6315	13329

Table 5.4: The choice points of solving the N-queens problems with MinSize

In combination with the best currently available branch ordering heuristics, namely minimum domain size variable ordering heuristic, domain splitting is not a good choice for a branching strategy in solving large N-queens problems and the choice between 2-Way and k-Way branching would not matter much since 2-Way simulates k-Way branching.

5.2 Results for the Golomb Ruler Problems

Unlike N-queens, it is a little difficult to pick the best branch ordering heuristic for the Golomb ruler problems. When there are fewer than 8 marks, the problems become too trivial to solve, meaning that the running times are very close to zero and it is very difficult to determine which branch orderings are better. In hope to collect more distinctive data, we have experimented with larger marks; however, the manifest difference did not show until the number of marks is more than 9 and when there are more than 9 marks, the problem becomes very hard to solve in combination with certain branch ordering heuristics, namely maximum domain size, greatest minimal bound, and greatest maximal bound heuristics. Table 5.5 is prepared by solving the Golomb ruler with 8 marks and the table suggests that lexicographical (Lex), minimum domain size (MinSize), least minimal bound (MinMin) and least maximal bound (MinMax) heuristics might be more effective than maximum domain size (MaxSize), greatest minimal bound (MaxMin), and greatest maximal bound (MaxMax) in solving the Golomb ruler problems. Hence, for larger problem sizes, different branching strategies are tested only with lexicographical, minimum domain size, least minimal bound, and least maximal bound branch ordering heuristics.

The Execution Time (in seconds)	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	0.4	0.4	0.6	0.4	1.1	0.4	1.1
2-Way Branching	0.4	0.4	0.6	0.4	1.1	0.4	1.1
Domain Splitting	0.4	0.4	9.2	0.4	1.1	0.4	1.1

Table 5.5: The execution time of different branching strategies for the OGR with 8 marks

Since the execution time exceeds 1000 seconds when the number of marks is greater than 10, the experiments were carried up to the Golomb rulers with 10 marks. The following Tables 5.6, 5.7, 5.8 and 5.9 show the results from these experiments. Irrespective of different branch orderings, one can observe from these tables that domain splitting works better than 2-Way and k-Way branchings and the difference becomes more prominent as the problem size increases.

In spite of the fact that domain splitting creates more choice points (as shown in Table

The Execution Time (in seconds)	$N = 3$	$N = 5$	$N = 7$	$N = 9$	$N = 10$
k-Way Branching	0	0.0	0.0	4.6	52.8
2-Way Branching	0	0.0	0.0	4.7	52.4
Domain Splitting	0	0.0	0.0	4.5	50.4

Table 5.6: The running time of different branching strategies for the Golomb ruler problems with lexicographical ordering

The Execution Time (in seconds)	$N = 3$	$N = 5$	$N = 7$	$N = 9$	$N = 10$
k-Way Branching	0	0.0	0.0	5.0	56.2
2-Way Branching	0	0	0.0	4.9	55.9
Domain Splitting	0	0	0.0	4.7	52.8

Table 5.7: The running time of different branching strategies for the Golomb ruler problems with minimum domain size ordering

The Execution Time (in seconds)	$N = 3$	$N = 5$	$N = 7$	$N = 9$	$N = 10$
k-Way Branching	0	0.0	0.0	4.6	52.7
2-Way Branching	0	0.0	0.0	4.6	52.8
Domain Splitting	0	0	0.0	4.5	50.6

Table 5.8: The running time of different branching strategies for the Golomb ruler problems with least minimal bound ordering

5.10), the running time of domain splitting seems to be a little faster than k-Way and 2-Way branching methods. For instance, when there are 10 marks in the ruler and lexicographical ordering is employed (refer to Table 5.6), domain splitting improves about 2 seconds; domain splitting takes 50.4 seconds to find the optimal solution while k-Way branching does for 52.8 seconds and 2-Way for 52.4 seconds. The improvement of 2 seconds sounds insignificant; however, if we take the problem size into consideration, we can suggest that domain splitting works better than k-Way and 2-Way branching in the Golomb

The Execution Time (in seconds)	$N = 3$	$N = 5$	$N = 7$	$N = 9$	$N = 10$
k-Way Branching	0	0.0	0.0	4.6	52.6
2-Way Branching	0	0	0.0	4.7	52.3
Domain Splitting	0	0.0	0.0	4.5	50.5

Table 5.9: The running time of different branching strategies for the Golomb ruler problems with least maximal bound ordering

ruler problems. Marriott and Stuckey [9] claim that domain splitting is beneficial for the optimization problems since it employs the optimal partitioning principle. The detailed description about the optimal partitioning is given in the Background section. Since the Golomb ruler problem is an optimization problem and it has the domain size of $O\left(N^2\right)$ if there are N marks, this empirical evidence supports the fact that domain splitting can be a better branching strategy for optimization problems with large domain size.

Also one might point out that the values for 2-Way and k-Way branching in the tables are almost alike within 0.4 seconds, and this suggests that 2-Way branching simulates k-Way branching as in the N-queens problems. Even when comparing the number of choice points, we are more confident to say that 2-Way and k-Way behave equally in solving the Golomb ruler problems. Table 5.10 is introduced as an example to consolidate this claim. It is prepared with the help of lexicographical ordering and the data in the table supports that 2-Way branching simulates k-Way branching since there is no difference in the number of choice points set by both branching strategies. This observation coincides with what we have observed in the N-queens problems. For other branch ordering heuristics, the same observation can be made, and we haven't included them in order to avoid redundancy.

In comparison with N-queens, a wide range of branch ordering heuristics are beneficial to use in the Golomb ruler problems, and in combination with them, domain splitting seems to be a better choice for the large Golomb ruler problems. As for k-Way and 2-Way, they behave in a similar manner irrespective of branch ordering heuristics. By inspecting their identical running times and choice points, we are more confident to suggest that 2-Way simulates k-Way in solving the Golomb ruler problems as in N-queens.

The Number of Choice Points	$N = 3$	$N = 5$	$N = 7$	$N = 9$	$N = 10$
k-Way Branching	0	13	713	41959	317620
2-Way Branching	0	13	713	41959	317620
Domain Splitting	0	16	726	42546	322280

Table 5.10: The choice points of different branching methods for the Golomb ruler problems with lexicographical ordering

5.3 Results for the Instruction Scheduling Problems

In order to find out the best branch ordering heuristic or heuristics for instruction problems, we chose the problem with 394 instructions as a benchmark. Table 5.11 shows the running time of different branching strategies in 394 instructions. In the table, the greatest minimal bound and greatest maximal bound heuristics seem to be more efficient than any other heuristics.

The Execution Time (in seconds)	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	0.7	0.5	> 100	0.6	0.2	0.6	0.2
2-Way Branching	0.6	0.5	> 100	0.6	0.2	0.6	0.2
Domain Splitting	4.9	1.3	> 100	5.1	0.2	4.7	0.2

Table 5.11: The execution time of different branching strategies in scheduling 394 instructions

Choice Points	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	87	43	139	87	18	74	10
2-Way Branching	87	43	139	87	18	74	10
Domain Splitting	449	128	551	476	28	474	29

Table 5.12: The choice points of different branching strategies in scheduling 69 instructions

The test cases are designed to test the efficiency of different branching strategies in a following manner: the problems with 69, 111, 216, 381, 690, and 1006 instructions are to find the optimal sequence of instructions in order to minimize the compilation time. With the aid of minimum domain size, greatest minimal and greatest maximal bound heuristics, we present the execution time of three branching strategies in Figure 5.13.

Branch Ordering	Branching Strategies	$N = 69$	$N = 111$	$N = 216$	$N = 381$	$N = 690$	$N = 1006$
MinSize	k-Way Branching	0	0.0	0.1	0.1	0.4	1.8
MinSize	2-Way Branching	0.0	0.0	0.1	0.1	0.4	1.8
MinSize	Domain Splitting	0.0	0.0	0.1	0.3	0.9	4.7
MaxMin	k-Way Branching	0	0.0	0.0	0.1	0.5	>100
MaxMin	2-Way Branching	0	0.0	0.0	0.1	0.5	>100
MaxMin	Domain Splitting	0.0	0.0	0.0	0.1	0.6	>100
MaxMax	k-Way Branching	0.0	0.0	0.0	0.1	0.7	2.0
MaxMax	2-Way Branching	0.0	0.0	0.0	0.1	0.7	1.9
MaxMax	Domain Splitting	0.0	0.0	0.1	0.1	0.6	16.1

Table 5.13: The execution time of different branching strategies in the instruction scheduling problems

The interesting point can be observed in Table 5.13; that is, the entries for 2-Way branching are almost the same as k-Way branching. With the empirical evidence as shown in choice points of Table 5.12 that is prepared to schedule 69 instructions, we cautiously presume that 2-Way branching also simulates k-Way branching in the instruction scheduling problem. As for different branch ordering heuristics, the minimum domain size variable ordering seems to be more efficient than the greatest maximal and the greatest minimal bound as the problem size increases. In the greatest minimal bound ordering, when there are 1006 instructions, it cannot even find the optimal solution within 100 seconds. Since we set 100 seconds as the time limit, >100 means that it is incomplete in Table 5.13.

Even though this problem is an optimization problem, domain splitting does not show any promising result as it did in OGR. Its running time gets very slow as the number of instructions increases, and it eventually become less effective than k-Way and 2-Way branching. The reason for this can be speculated in the algorithm employed from [13], and in this paper, the algorithm is already optimal and fast. Therefore, it might be difficult to improve on something already optimal.

5.4 Results for the Car Sequencing Problems

For the car sequencing problem, different branch ordering heuristics are tested and compared in solving 20 cars with 5 options and 6 different configurations. The time limit is set to 600 seconds. Table 5.14 shows that programs using lexicographical, minimum domain size, greatest minimal bound and least maximal bound clearly outperform programs with maximum domain size, least minimal bound and greatest maximal bound heuristics. Furthermore, maximum domain size variable ordering and domain splitting are found to be the worst combination in the car sequencing problem since the running time exceeds the time limit, 600 seconds.

The Execution Time (in seconds)	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	0.3	0.4	3.2	1.6	0.3	0.3	0.9
2-Way Branching	0.3	0.4	3.5	1.6	0.3	0.3	0.8
Domain Splitting	0.3	0.4	> 600	7.7	0.3	0.3	438.5

Table 5.14: The running time of different branching strategies in sequencing 20 cars

For the larger problems, Tables 5.15, 5.16, 5.17, and 5.18 present the data for sequencing cars up to 50 cars with Lex, MinSize, MaxMin, and MinMax correspondingly. In all four tables, k-Way, 2-Way branching and domain splitting seem to perform in a similar manner although the actual difference between domain splitting and k-Way branching is at most 4 seconds and the difference between k-Way branching and 2-Way branching is at most 2 seconds—in most cases, k-Way and 2-Way have an identical value. We can notice the pattern in which domain splitting becomes more efficient than 2-Way and k-Way branchings. For example, when there are 50 cars, domain splitting becomes faster than any other branching in all the tables. Hence we cautiously claim that domain splitting becomes more favorable irrespective of the choice of branch ordering as the problem size increases. When we compare the number of choice points set by three different branching strategies in Table 5.19, domain splitting surprising generates less choice points than k-Way and 2-Way. This observation also supports that domain splitting is more beneficial to use in the car sequencing problems.

The Execution Time (in seconds)	10 cars	20 cars	30 cars	40 cars	50 cars
k-Way Branching	0.0	0.3	2.6	20.6	152.8
2-Way Branching	0.0	0.3	2.7	20.4	154.4
Domain Splitting	0.0	0.3	2.7	20.1	148.9

Table 5.15: The running time of different branching strategies with Lex in sequencing cars

The Execution Time (in seconds)	10 cars	20 cars	30 cars	40 cars	50 cars
k-Way Branching	0.0	0.4	2.8	21.2	153.0
2-Way Branching	0.0	0.4	2.6	20.5	153.7
Domain Splitting	0.0	0.4	2.7	20.1	150.7

Table 5.16: The running time of different branching strategies with MinSize in sequencing cars

The Execution Time (in seconds)	10 cars	20 cars	30 cars	40 cars	50 cars
k-Way Branching	0.0	0.3	2.9	20.8	160.2
2-Way Branching	0.0	0.3	2.7	21.2	159.7
Domain Splitting	0.0	0.3	2.8	21.2	156.9

Table 5.17: The running time of different branching strategies with MaxMin in sequencing cars

As for k-Way and 2-Way branchings, we once again presume that 2-Way branching ends up simulating k-Way branching in ILOG when solving car sequence problems because there is empirical evidence that k-Way is closely related to 2-Way branching. For instance,

The Execution Time (in seconds)	10 cars	20 cars	30 cars	40 cars	50 cars
k-Way Branching	0.0	0.3	2.4	20.2	149.9
2-Way Branching	0.0	0.3	2.7	20.4	151.0
Domain Splitting	0.0	0.3	2.7	20.2	147.8

Table 5.18: The running time of different branching strategies with MinMax in sequencing cars

Table 5.19 shows the number of choice points set by the three different branching strategies in combination with the lexicographical branch ordering heuristic. In the table, the values for k-Way and 2-Way are exactly the same and it suggests that 2-Way behaves in the same way as k-Way.

Choice Points	10 cars	20 cars	30 cars	40 cars	50 cars
k-Way Branching	3	769	5519	40156	289824
2-Way Branching	3	769	5519	40156	289824
Domain Splitting	6	733	5290	38492	277842

Table 5.19: The choice points of different branching strategies with Lex in sequencing cars

5.5 Results for the Truck Scheduling Problems (*TSP*)

In order to find a more efficient branch ordering heuristic for the truck scheduling problem (*TSP*), we tested 7 different branch ordering upon solving *TSP* with 7 customers, and the table 5.20 is obtained.

Solely based on the running time, it is extremely difficult to select efficient branch ordering heuristics because the running times in table 5.20 are very alike within ± 0.02 seconds; therefore, we compare the number of choice points among different branch orderings as shown in table 5.21. Taking choice points into account, we select lexicographical, maximum domain size, greatest minimal bound, least maximal bound, greatest maximal bound branch ordering heuristics to examine branching strategies in larger problems.

The Execution Time (in seconds)	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	0.5	0.6	0.5	0.5	0.6	0.6	0.6
2-Way Branching	0.5	0.6	0.5	0.5	0.5	0.6	0.5
Domain Splitting	0.6	0.6	0.4	0.5	0.5	0.5	0.5

Table 5.20: The execution time of different branching strategies in *TSP* with 7 customers

The Number of Choice Points	Lex	MinSize	MaxSize	MinMin	MaxMin	MinMax	MaxMax
k-Way Branching	37508	71434	33307	46744	37394	37497	37511
2-Way Branching	37508	71434	33307	46744	37394	37497	37511
Domain Splitting	37442	71382	36367	45183	37333	37411	52307

Table 5.21: The choice points of different branching strategies in *TSP* with 7 customers

The truck scheduling problem (*TSP*) is also an optimization problem as the OGR, and as expected, domain splitting works better than 2-way and K-way branching if we employ the least maximal bound ordering; when there are twelve customers, domain splitting can find the shortest route for the truck in 246 seconds which is 12 seconds faster than k-Way branching and 21 seconds faster than 2-Way branching. With other branch orderings, k-Way branching seems to be fastest. Especially with maximum domain size ordering, k-Way and 2-Way branchings are about 41 times faster than domain splitting. The closeness of k-Way and 2-Way regardless of branch ordering heuristics is well elucidated in Table 5.22.

5.6 Results for the Uniform Binary Random Problems

Before testing, some parameters for the uniform binary random problems need to be set. These are such as the number of variables, the number of values in the domain, the number of constraints, the number of invalid assignments (no-goods), the random seed to generate the random sequence, and the number of instances. Firstly, based on the personal preference, we chose the random problem with 32 variables, 10 values, 50 constraints, 100 for

Branch Ordering	Branching Strategies	3 customers	5 customers	7 customers	9 customers	11 customers	12 customers
Lex	k-Way Branching	0.4	0.4	0.5	1.9	39.5	307
Lex	2-Way Branching	0.4	0.4	0.5	1.8	41.9	321
Lex	Domain Splitting	0.4	0.3	0.6	1.7	39.5	306
MaxSize	k-Way Branching	0.5	0.4	0.5	0.9	2.0	7.3
MaxSize	2-Way Branching	0.5	0.4	0.5	0.8	2.1	7.6
MaxSize	Domain Splitting	0.4	0.3	0.4	1.6	39	315
MaxMin	k-Way Branching	0.4	0.4	0.6	1.6	23.8	221
MaxMin	2-Way Branching	0.4	0.4	0.5	1.7	25.5	228
MaxMin	Domain Splitting	0.4	0.4	0.5	1.6	24.3	224
MinMax	k-Way Branching	0.5	0.4	0.6	1.8	38.1	258
MinMax	2-Way Branching	0.4	0.4	0.6	1.9	40.2	267
MinMax	Domain Splitting	0.4	0.4	0.5	1.8	36.7	246
MaxMax	k-Way Branching	0.4	0.4	0.6	1.8	40.2	309
MaxMax	2-Way Branching	0.4	0.4	0.5	1.8	42.2	324
MaxMax	Domain Splitting	0.3	0.3	0.5	2.1	38.1	344

Table 5.22: The execution time of different branching strategies in *TSPs*

the random seed, and 200 repetitions for each problem, and the number of invalid varies from 1 to 100 as the small size problem. With these parameters, we ran tests to measure the running time of different branching strategies also employing seven different branch ordering heuristics. In this experiment, we found that some branch ordering heuristics are too efficient to compare the effectiveness of different branching strategies, and some entries in the data are very close to zero. The results are graphically shown in Figure 5.1

In order to analyze more effectively the data, we use a table to represent the data for MaxSize that has the longest running time in Table 5.23. In order to emphasize the phase transition that occurs when a problem becomes extremely difficult to solve, the number of no-goods in Table 5.23 only ranges from 50 to 86 with the increments of 3 since the phase transition for this particular instance is calculated as 77.

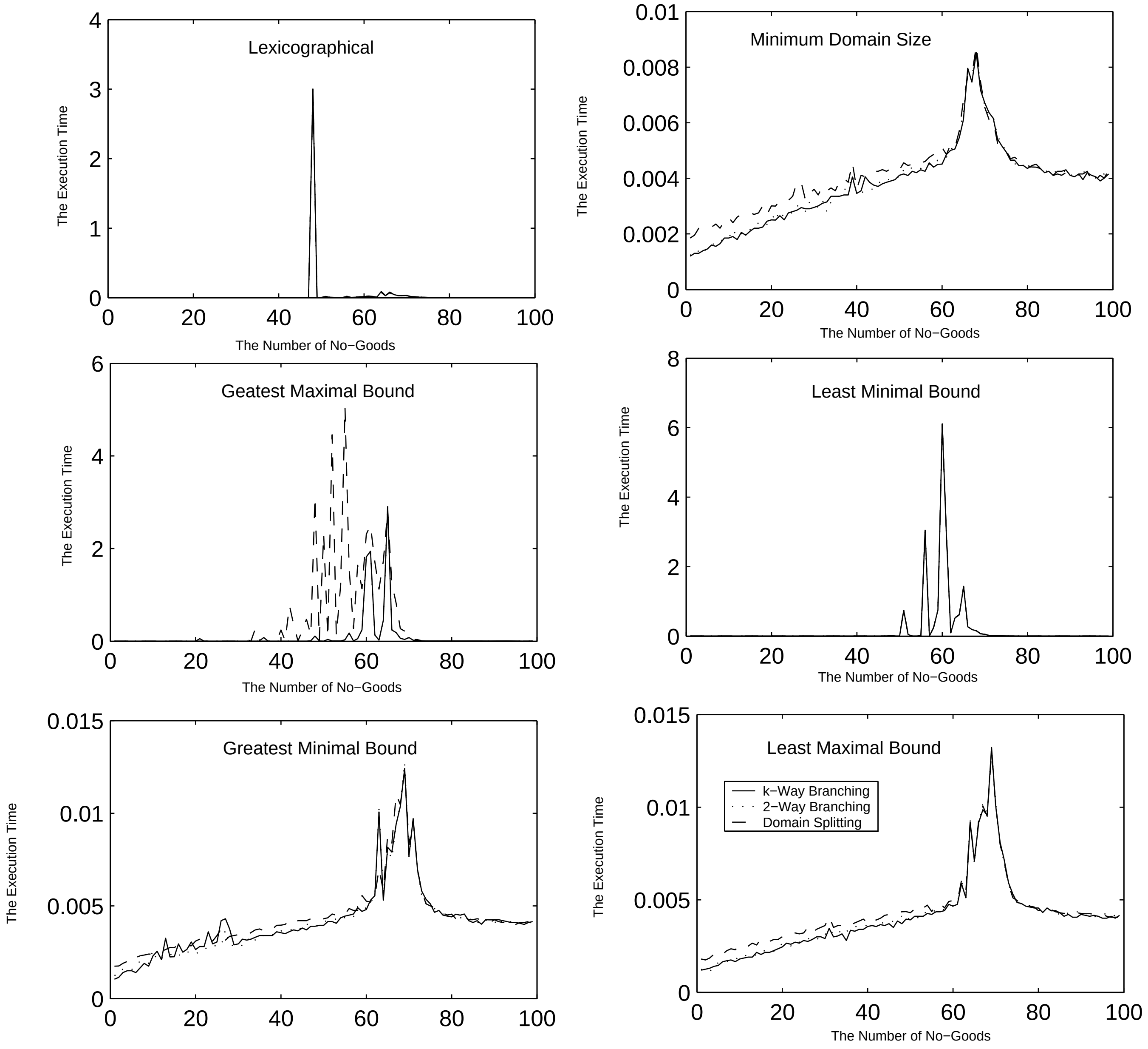

Figure 5.1: The running time of different branching strategies in solving the random problem with 32 variables

The number of no-goods	k-Way branching	2-Way branching	Domain Splitting
50	0.0	0.0	0.2
53	0.5	0.5	0.4
56	3.9	3.9	0.8
59	6.7	7.8	10.0
62	19.7	18.6	19.2
65	34.9	33.9	31.8
68	16.1	17.5	16.5
71	3.4	3.4	1.3
74	0.1	0.1	0.0
77	0.0	0.0	0.0
80	0.0	0.0	0.0
83	0.0	0.0	0.0
86	0.0	0.0	0.0

Table 5.23: The running time for different branching strategies combined with MaxSize in solving the random problem with 32 variables (in seconds)

With the aid of the constrainedness , a phase transition point is calculated to make the constrainedness equal to 1, and the constrainedness for the random problem is defined in [3] as follows: $K = \frac{n-1}{2} * p_1 * \frac{\log \frac{1}{1-p_2}}{\log m}$ where n is the number of variable, m is the number of values, p_1 is a constraint density, and p_2 is a tightness. If we insert all the parameters $(n = 32, m = 10, K = 1, p_1 = 0.1)$ and solve for p_2, we can approximate the value of p_2 as 0.77, meaning that there are approximately 77 no-goods.

This approximately coincides with the graphs in Figure 5.1; however, with lexicographical ordering, the phase transition happens around 50 no-goods and the one for MaxSize happens to be around 65 in Table 5.23. Comparing the running times among the figures in Figure 5.1 and Table 5.23, one can notice that minimum domain size branch ordering has the most efficient execution time around the phase transition point. Another interesting point about the random problem can be made—if the problem becomes very hard, around the phase transition area, domain splitting and 2-Way branching ends up simulating k-Way branching.

The number of no-goods	k-Way branching	2-Way branching	Domain Splitting
50	29	29	54
53	27	27	49
56	25	25	44
59	23	25	39
62	22	23	35
65	24	24	35
68	25	25	31
71	8	8	9
74	1	1	1
77	0	0	0
80	0	0	0
83	0	0	0
86	0	0	0

Table 5.24: The number of choice points for different branching strategies combined with MinSize in solving the random problem with 32 variables

When we compare the choice points for the random problem, we feel more confident to claim that 2-Way branching simulates k-Way branching. We prepared Table 5.24 in the same way as Table 5.23 except with the more efficient branch ordering heuristic, MinSize, and the entries for k-Way and 2-Way branching seems to be very similar to each other. If we take into consideration that each of the entries in Table 5.24 is the average of 200 different instances, the table shows some evidence that 2-Way branching simulates k-Way in solving the random problem. Furthermore, Table 5.24 also suggests that domain splitting is distinguishably different from the other two. As observed in the earlier experiments, domain splitting generates more choice points than k-Way and 2-Way.

When the problem becomes harder to solve, the running times of different branching methods tend to merge together at the phase transition state regardless of the choice of branch ordering heuristics. This phenomenon can still be observed even in the larger problems. For example, the data in Table 5.25 exhibits the same pattern. The table is prepared by solving a random problem with 50 variables, 10 values, 122 constraints and -100 random seed with the help of Lex, and truncated around the phase transition point.

The number of no-goods	k-Way branching	2-Way branching	Domain Splitting
50	0.9	0.9	0.9
52	1.2	1.8	1.7
54	7.6	7.6	7.1
56	11.6	11.5	11.1
58	2.0	2.0	1.9
60	0.8	0.8	0.7
62	0.1	0.1	0.1
64	0.0	0.0	0.0
66	0.0	0.0	0.0
68	0.0	0.0	0.0
70	0.0	0.0	0.0
72	0.0	0.0	0.0

Table 5.25: The running time of different branching strategies combined with Lex in solving the random problem with 50 variables (in seconds)

Chapter 6

Discussion

In this section, we discuss the meaning of our experiments by comparing our test results with the predictions made in some of the literature. Furthermore, we attempt to explain some of the behaviors observed in the experiments.

Based on [9], the domain splitting strategy should work better than the others since it follows the least commitment principle. This principle states that when making a choice for some variable we should make the choice which commits us as little as possible [9]. The advantage of this is that, if we detect unsatisfiability after making a weak commitment, more of the search space is removed than if we detect unsatisfiability after a strong commitment. In k-Way and 2-Way branchings, setting a variable to a value commits the variable to a single value, which is very restrictive. Domain splitting is less restrictive [9] as it only removes half of the remaining values in the variable's domain rather than all but one. The principle of least commitment, therefore, tells us to prefer domain splitting since the other labeling approaches lead to stronger commitment, and so does it prefer 2-Way over k-Way branching because 2-Way commits less than k-Way [9].

Marriott and Stuckey [9] also point out that optimization problems can be good candidates to use domain splitting. The search for an optimal solution involves computing a long sequence of answers, each slightly better than the last, until the optimal solution is eventually reached. In a sense, the default minimization routine performs a linear search through the solution space in which solutions are ordered by the best value of the objective function. However, domain splitting reduces the number of solutions considered by performing a binary search through the solution space. Marriott and Stuckey refer to this process as optimal partitioning and explain it in the following way. We first search for a solution in the lower half of the range. If this is found, we then look for a better solution in the lower half of the range. If there is no solution in the lower half of the range, we search in the upper half of the range. At each step in the search we keep track of the minimum and maximum values that the objective function can take. Our experiments provides empirical evidence that domain splitting can be beneficial to solve optimization problems like optimal Golomb ruler and truck scheduling problems. Although domain splitting only marginally improves the execution time by at most 12 seconds, we can observe the pattern that the amount of improvement becomes larger as the problem size increases; hence, it is reasonable to conclude that domain splitting can be more beneficial when larger optimization problems are solved.

In other finite domain problems, N-queens and instruction scheduling show that domain splitting is less efficient than k-Way and 2-Way branchings. The ineffectiveness of domain splitting can be explained with the following statement: the fact that domain splitting is not always advantageous as less commitment may also mean less information for the incomplete solver to determine satisfiability or unsatisfiability [9]. For example, propagation like forward checking becomes inefficient when a variable is not assigned. The reason is that, in order for the propagator to check the consistency between the future variables and the current assignment, the current variable has to be committed, meaning that a variable is assigned to a value. Because domain splitting delays strong commitment, the information that the propagator uses to prune the search space is less available in domain splitting than in the other branching techniques. This was the case in the above satisfiable problems, which favors the labeling strategies with strong commitment.

The other possible explanation for the inefficiency of domain splitting on the N-queens problem can be found in [12], which states that benchmarks on different algorithms produced using the N-queens problem must be interpreted with caution. This is because the N-queens problem has very specific features: first, it is a binary constraint problem; second, every variable is constrained by every other variable, which need not be the case in other problems [12]. More importantly, in the N-queens problem, each label for every variable conflicts with at most three values of each other variable, regardless of the number of variables in the problem [12]. For example, the label, $x_1 = 2$ meaning that the first column variable is assigned to the value 2 (i.e. a queen is placed in the first column and the second row) has conflict with other labels $x_2 = 1$, $x_2 = 2$, and $x_2 = 3$. In the 8-queens problem, for example, when 2 is assigned to Queen 1, there are 5 out of 8 values that Queen 2 can take. But in the 1,000,000-queens problem, there are 999,997 out of 1,000,000 values that Queen 2 can take after $x_1 = 2$ has been committed to. Therefore, constraints get looser as N grows larger [12]. For large N, it is well known that solutions to the N-queens problems are more likely to be found by starting in the middle of the domain [9]. The uniqueness of the N-queens problem contributes to the slowness of the program using domain splitting.

Another interesting finding of this experiment is that the trace of 2-Way is exactly the same as the one of k-Way branching in most cases. Even in some testing cases such as in random and car sequencing problems, domain splitting behaves similarly to k-Way and 2-Way branchings. This suggests that domain splitting and 2-Way cannot utilize effectively the least commitment principle in finite domain *CSPs* unlike as in numeric *CSPs*—domain splitting seems to be very efficient in numeric *CSPs* [7].

We also were able to observe that different branch ordering heuristics cause different behavior of the running times of the branching strategies. With the help of the most efficient branch ordering heuristics available for the problem, the difference among the branching strategies becomes too insignificant to distinguish them. With careful consideration, we realize that domain splitting and 2-Way simulate k-Way; they choose the same variables and values. This observation makes us question the effectiveness of the current branch ordering techniques on 2-Way branching and domain splitting. Since most heuristics are designed specifically for k-Way branching, we might need some other ways to choose branch ordering in a way to take full advantage of the least commitment principle.

The key of success in domain splitting lies in the fact that it is less likely to fail in the early stage of the search and eventually leads to more effective pruning once more information is available in a later search stage, because it delays committing until one of domains is split into a singleton. In order to utilize domain splitting better, we first need to find a more effective mechanism to collect the unsuccessful values into one group so that we can reduce the search space once the information is available.

Chapter 7

Conclusion

For many constraint programs with finite domains, efficiency is synonymous with efficient labeling and the constraint programmer is well advised to consider strategies for labeling which reduce the search space and eventually lead to finding a solution faster [9]. Different branching strategies in labeling are implemented and tested for their efficiency in this thesis. When they are empirically compared, there is some evidence that domain splitting improves the running time in optimization finite domain problems and the efficiency increases as the problem gets larger because domain splitting follows the least commitment principle as well as optimal partitioning.

However, most of our test cases for finite domain *CSPs* suggest that the choice of branching strategies does not matter much—provided we are using an effective variable ordering heuristic. This result is rather surprising since domain splitting and 2-Way branching were predicted to be more efficient than k-Way branching. In our experiments, domain splitting and 2-Way branching end up simulating k-Way branching, given the widely used variable ordering heuristics we employed in our study.

The inefficiency of least committed branching strategies in finite domain *CSPs* can be explained by the fact that the least commitment principle ends up reducing the useful information for a propagator to effectively prune, as the principle always prefers a least committed choice over strongly committed choices that contain more information. Also, the intrinsic characteristics of the problems as in N-queens also contribute to the latency of finding a solution. The N-queen problem is a binary constraint problem, and every variable is constrained by every other variables. Furthermore, it is less constrained as N grows. The least commitment principle seems to be more effective for the problems with large sizes; however, in N-queens, it is not the case since constraints get looser as N grows larger.

Based on these empirical studies, we hope that other practitioners follow up with our experiments so that they can show that the branching strategies with least commitment principle can work better than the other methods when the constraints involving the variables to be labeled can gain substantial consistency information from the split domains provided that there is a heuristic available to utilize this information. In spite that domain splitting also produces a larger derivation tree than k-Way and 2-Way branchings, when there are many variables and their domains are large, domain splitting can be more efficient than the other strategies. By splitting a domain into two halves, we may be able to eliminate half of a variable's possible domain values. It can be ideal to split a domain in a way to separate all the inconsistent values from consistent values so that the propagator can eliminate all the inconsistent domain values at once. Designing such a propagator, splitting strategies, and specialized heuristics to better utilized the information carried in 2-Way branching and domain splitting are left for the future work.

In addition, the hybridized branching strategies can be also possible. At the different levels of a search tree, different branching strategies can be employed to maximize the pruning. For instance, first we employ domain splitting to delay any strong commitment, but once we gain some information about the problem, we can reduce the search space by committing strongly. In order to implement this, we first need to find a way to evaluate the good level for different branching techniques. We also leave this topic for future work.

Bibliography

[1] C. Bessiere. Random uniform **CSP** generators. Available online: `http://www.lirmm.fr/~bessiere/generator.html`.

[2] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Fifth International Conference on Logic Programming*, 1998.

[3] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proc. AAAI-96*, 1996.

[4] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, pages 14:263–313, 1980.

[5] ILOG. *ILOG SOLVER:Reference Manual*, 4.2 edition, 1998.

[6] ILOG. *ILOG SOLVER:User Manual*, 4.2 edition, 1998.

[7] N. Jussien and O. Lhomme. Dynamic domain splitting for numeric *CSPs*. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence*. John Wiley & Sons, 1998.

[8] Z. Kiziltan, P. Flener, and B. Hnich. Towards inferring labelling heuristics for csp application domains. `http://www.dis.uu.se/~pierref/astra`.

[9] K. Marriott and P. J. Stuckey. *Programming with Constraint: An Introduction*. MIT press, 1990.

[10] D. G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 555–569, Kinsale, Ireland, 2003.

[11] J. F. Puget and M. Leconte. Beyond the blackbox: Constraints as objects. In J. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 513–527. MIT Press, 1995.

[12] E. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[13] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001.

[14] P. van Hentenryck. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.

[15] W. J. van Hoeve and M. Milano. Decomposition based search: A theoretical and experimental evaluation. `http://homepages.cwi.nl/~wjvh/papers/dbs.pdf`.