Parallel Constraint Solving for Combinatorial Problems

Von der Fakultät 1 - MINT - Mathematik, Informatik, Physik,

Elektro- und Informationstechnik

der Brandenburgischen Technischen Universität Cottbus–Senftenberg

genehmigte Dissertation

zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften

(Dr.-Ing.)

vorgelegt von

Ke Liu

geboren am 15.11.1984 in Jiangxi, China

Vorsitzende/r: Prof. Dr. habil. Douglas W. Cunningham

Gutachter/in: Prof. Dr. rer. nat. habil. Petra Hofstedt

Gutachter/in: Prof. Salvador Abreu, University of Évora

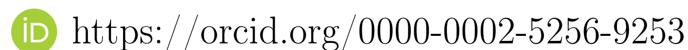
Tag der mündlichen Prüfung: 20.01.2021

DOI:10.26127/BTUOpen-5437

Parallel Constraint Solving for Combinatorial Problems

Ke Liu

bios8086@vip.qq.com


https://orcid.org/0000-0002-5256-9253

Department of Programming Languages and Compilers

Brandenburg University of Technology Cottbus-Senftenberg

Advisor : Prof. Dr. rer. nat. habil. Petra Hofstedt

A dissertation submitted for the degree of
Doctor of Philosophy (Dr.-Ing.)
March 21, 2020

Dedicated to all those who are interested in this topic.

Acknowledgements

First and foremost, I should like to begin by expressing my appreciation to my Ph.D. advisor, Professor Petra Hofstedt, for her support, understanding, and encouragement throughout this work. I am fortunate to have a Ph.D. advisor who is nice and supportive. Without her, I would not get a chance to know that constraint programming is an exciting research area.

Here, I would also like to mention my uncle. He was diagnosed with advanced liver cancer six years ago. But unbearable pain has not crushed him, and he is still tenaciously struggling against cancer. No matter what happens, his courage and optimism will always illuminate the road of my life.

Finally, I would also like to take this opportunity to thank my parents. I have several other people I would like to thank, as well. They are Sven Löffler, Gudrun Pehle, and Katrin Ebert in the department of programming languages and compiler construction of B-TU. I want to thank every one of them for all the support and help they have given me.

Abstract

Nowadays, it has become increasingly hard for constraint solving to keep profiting from the performance improvement of uniprocessor due to the death of Moore's Law. Consequently, with parallelism becoming the standard in computer design, research on parallel constraint solving technique is of vital importance for enhancing the performance of constraint solving. However, it is not an easy task for constraint solving to exploit parallelism effectively. Neither a general solution nor overall guidance is yet available for parallel constraint solving after more than two decades of development efforts from the constraints community.

The main objective of this dissertation is to explore how parallel processors can be utilized to speed up constraint solving, more specifically, to find a solution for new instances or for existing instances faster. We review the literature on exploiting parallelism in constraint solving to help gain insight into the rationale of different types of parallel constraint solving approaches. And on this basis, we show theoretically and empirically that applying parallelism to solving computationally hard problems can hedge against mistakes made by search strategies near the root of the search tree, often achieving superlinear speedups. We also propose a new parallel stochastic portfolio search to solve instances that cannot be solved by the previous parallel portfolio search. Finally, we present a hypergraph decomposition method to allocate constraints to parallel processors for parallel constraint solving.

Contents

1 Introduction **1**

1.1 Contributions and organization 2

2 Background Information **5**

2.1 Constraint Programming 5

2.1.1 Constraint Propagation 7

2.1.2 Backtracking Search 10

2.1.3 Symmetry in Constraint Programming 16

2.2 Parallel Computing 21

2.2.1 Amdahl's Law and Gustafson's Law 23

2.2.2 The Meaning of Gustafson's Law for Parallel Constraint Solving 26

3 The Literature Review on Parallel Constraint Solving **28**

3.1 Parallel Constraint Propagation 29

3.2 Parallelizing the Search Process 32

3.3 Portfolios 44

3.4 Hybrid Approaches 46

3.5 Conclusion 49

4 The Effectiveness of Parallel Constraint Solving **50**

4.1 Early Mistakes 51

4.2 Possible Approaches to Tackle Early Mistakes 53

4.2.1 Restart-Based Search 54

4.2.2 Limited Discrepancy Search 58

4.2.3 Parallel Portfolio Search 62

4.2.4 Embarrassingly Parallel Search 64

4.3 Conclusion 71

5 Case Studies of the EPS Approach **72**
5.1 Social Golfer Problem 73
5.1.1 The Introduction of Social Golfer Problem 73
5.1.2 Background Information 75
5.1.2.1 The Difficulties of Solving the SGP 75
5.1.2.2 Global Constraints for Modelling SGP 76
5.1.3 The Basic Model 77
5.1.4 Instances Solved Sequentially 80
5.1.4.1 7-7-8 etc. 81
5.1.4.2 9-9-10 82
5.1.4.3 13-13-14 etc. 83
5.1.4.4 8-8-9 84
5.1.5 Instances Solved in Parallel 85
5.1.5.1 6-3-8 86
5.1.5.2 6-4-7 88
5.1.5.3 7-3-10 89
5.1.6 Experiments 90
5.1.6.1 Experimental Results on Instance Solved Sequentially 90
5.1.6.2 Experimental Results on Instance Solved in Parallel . 91
5.1.6.3 Discussion 92
5.1.7 Related Work 93
5.1.7.1 Methods from the CSP Literature 93
5.1.7.2 Methods from the Metaheuristic Literature 94
5.1.7.3 Summary 95
5.1.8 Conclusion 95
5.2 Traveling Tournament Problem with Predefined Venues 97
5.2.1 Introduction to the TTPPV 97
5.2.2 Modeling the TTPPV Based on Perfect Matching (The First Model) 100
5.2.2.1 A Model for Perfect Matching 101
5.2.2.2 A Model for the Timetable 102
5.2.2.3 Experimental Results 102
5.2.2.4 A Complete Model 103
5.2.2.5 Executing the Complete Model in Parallel 105
5.2.2.6 Experimental Results 106

5.2.3 An Advanced Modeling Approach for Larger Instances (A Second Model) . . . 107
5.2.3.1 An Advanced Model . . . 107
5.2.3.2 Solving the Model in Parallel for Larger Instances . . 110
5.2.3.3 Experimental Results on the Large Instance Model . 110
5.2.4 Discussion . . . 112
5.2.5 Conclusion . . . 113
5.3 Talent Scheduling Problem . . . 113
5.3.1 The Introduction of the TS . . . 113
5.3.2 The CSP Model of the TS . . . 115
5.3.3 Solving the TS in Parallel . . . 119
5.3.4 Numerical Results . . . 120
5.3.5 Conclusion . . . 121
5.4 Conclusion . . . 122

6 Parallel Stochastic Portfolio 124
6.1 Introduction . . . 124
6.2 The Components of the Current Single-Solver-Based Portfolio Approach 125
6.3 The Limitations of the Current Parallel Portfolio Search . . . 130
6.4 A Novel Parallel Stochastic Portfolio Approach . . . 132
6.5 Experimental Results . . . 134
6.6 Related Work . . . 136
6.7 Discussion and Conclusion . . . 136

7 Towards Parallel Constraint Solving by Hypertree Decomposition 138
7.1 Introduction . . . 138
7.2 Preliminaries . . . 139
7.3 The algorithm *det-k-CP* . . . 142
7.4 Experimental Results . . . 148
7.5 Conclusion and Future Work . . . 151

8 Conclusions 152

A The Solutions of Some SGP Instances 154

Bibliography 155

List of Figures

2.1 A possible solution of the 8-queens. 6
2.2 A fragment of the BT search for the constraint network $\mathcal{N}$ of Example 2. The potential node which is pruned because it violates constraints are labeled with ✗. A ✓ sign indicates that a solution is found. 12
2.3 Another fragment of the BT search for the constraint network $\mathcal{N}$ of Example 2. 14
2.4 A fragment of the BT search formed for the constraint network $\mathcal{N}$ of Example 2, using a better value ordering heuristic. 15
2.5 The 8 symmetries of a solution of 8-Queens. 16
2.6 Example of SBDS on a backtrack search tree for the 8-Queens problem. 21

4.1 Unrecoverable fatal mistake because of the large subtrees below to the mistake B. Nodes B and G are mistakes. 52
4.2 An ideal backtrack search tree formed by an optimal search strategy for a first solution of the same CSP shown in Figure 4.1. 53
4.3 Comparison between heavy-tailed and non-heavy-tailed runtime distribution. CDF stands for Cumulative Density Function log-log scale. (Figure adapted from [69].) 56
4.4 LDS search tree. The text inside of a node stands for the discrepancy of this node. (Figure adapted from [144].) 58
4.5 The four possible situations for a node and its children. A white node and a black node stand for a bad node and a good node, respectively. (Figure adapted from [91].) 59

4.6 Five probes of the one discrepancy iteration on a binary search tree of height four. Pi above an arrow denotes that the node pointed by the arrow is visited on probe i. For example, $P3, P4$ means that a node is visited by both probe 3 and probe 4. The number i inside of a leaf node indicates this leaf node visited on probe i. In each probe, the nodes are numbered for each probe, starting with the root node (numbers in bold type), from zero to four. 61
4.7 The typical possibilities for speedup in EPS. The good and bad nodes are still black and white, respectively; the hollow triangles with a dashed border and the black triangle stand for empty subtrees and subtree with a solution, respectively. 68
4.8 Other two possibilities for speedup in EPS. 69

5.1 A conversion from solutions generated by the model to the potential values of columns of T, the number of teams is 8. (Figure reproduced from [122].) 101

7.1 The hypergraph for the constraint network. (Figure adapted from [80] and reproduced from [120].) 140
7.2 Hypertree decomposition for the hypergraph of Figure 7.1. (Figure adapted from [80] and reproduced from [120].) 141
7.3 A constraint network is divided into eight parts. An edge between two nodes is due to the shared variables. (Figure reproduced from [120].) 142
7.4 A new node is prepended to a degenerated tree with 3 nodes. (Figure reproduced from [120].) 148

List of Tables

5.1 A solution for 7-3-10. The text in bold indicates that the values have been initialized before search. (Table adapted from [125, 124].) 74

5.2 A solution is obtained by our model for 7-3-10 instance, and it is equivalent to the solution depicted in Table 5.1. The bold-italic text indicates that the values have been initialized before search; the italic text stands for the values frozen by Constraints (5.2). (Table adapted from [124, 125].) 78

5.3 A solution of 5-5-6 expressed by groups. It can be converted to the solution expressed by players easily. The submatrices GS_1 and GS_2 are written in red. (Table adapted from [125, 124].) 80

5.4 The second matrix GS_1 for the instance 13-13-14. (Table reproduced from [125, 124].) 84

5.5 The second matrix GS_1 for a solution 8-8-9. (Table reproduced from [125, 124].) 85

5.6 A solution of 6-3-8 expressed by groups. (Table adapted from [125, 124].) 87

5.7 A solution of 6-4-7. The numbers with the same superscript are in non-decreasing order in the second row. (Table adapted from [125, 124].) 88

5.8 Results on the *s*-*s*-(*s+1*) Instances. A superscript “c” means that the instance was open for the constraint satisfaction approach; “dom” and “min” denote the predefined search strategies domOverWDegSearch and minDomLBSearch in Choco Solver, respectively. (Table reproduced from [125, 124].) 91

5.9 Results on the Instances solved in parallel. A superscript “f” means that the instance is solved by computer for the first time. A “-” sign means the program was still running after a period which is equal to the number of workers multiplied by the execution time in parallel. (Table adapted from [125, 124].) 91

5.10 The summary of the most significant results on the SGP from the computer-science community. (Table adapted from [124].) 95
5.11 A feasible solution for a TTPPV problem with 8 teams. 99
5.12 The predefined venue data for the problem shown in Table 5.11. . . . 99
5.13 The comparison between our timetable model and the timetable model presented in [161]. The data in parentheses separated by commas were calculated by our model (left) and the model of [161] (right) respectively. (Table reproduced from [122].) 103
5.14 The experimental results on instance n=10. (Table reproduced from [122].) 106
5.15 Comparison of the three models on different size of instances. The data in parentheses separated by commas were calculated by the first model, the second model, and the model of [161] respectively. N/A indicates that our first model cannot handle the instance. (Table reproduced from [122].) 110
5.16 The experimental results for 18 teams using the second model.(Table adapted from [122].) 111
5.17 The results of simulation parallel executions by executing in sequential way. (Table adapted from [122].) 112
5.18 A feasible solution for a given TS in [191]. The Cost and Duration in the last column and the last row stands for the cost per time unit and the duration of the pieces, respectively. The overall cost of this solution is 14,600. See below for calculating the cost of a solution. (Table reproduced from [126].) 114
5.19 Solving the TS on a multi-core computer. (Table reproduced from [126].) 120
5.20 Using 4 workers to calculate the first part and second part in turn. . . 121
5.21 Optimal solutions with cost 14,600. (Table reproduced from [126].) . . 121

6.1 A possible configuration of portfolio search (PPS). 129
6.2 The comparison between our approach and parallel portfolio search shown in Table 6.1 (PPS) on 32 cores. 134

7.1 Experimental results for *det-k-CP* of the benchmark suite from [81]. (Table reproduced from [120].) 149

A.1 The solution for 6-3-8 transformed from the solution Shown in Table 5.6. (Table adapted from [124].) 154

A.2 A new non-isomorphic solution for the 6-3-8 instance. (Table adapted from [124].) 154
A.3 The solution for 6-4-7 transformed from the solution Shown in Table 5.7. (Table adapted from [124].) 155
A.4 A new non-isomorphic solution for the 7-3-10 instance. (Table reproduced from [124].) 155
A.5 A new non-isomorphic solution for the 7-3-10 instance. (Table adapted from [124].) 155
A.6 A new non-isomorphic solution for the 7-3-10 instance. (Table adapted from [124].) 156
A.7 A solution of 8-8-9 expressed by groups. (Table adapted from [124].) . 156

Chapter 1

Introduction

It is believed that Artificial Intelligence (AI) is one of the critical drivers of the upcoming fourth industrial revolution. Its effect has already been seen in every aspect of society, businesses, and daily life. AI applications, for example, include speech recognition, autonomous vehicles, medical diagnosis, machine translation, game playing, tutoring systems, robotics, smart house, logistics planning, and scheduling factory processes, etc. AI encompasses a considerable variety of subfields, ranging from the general (e.g., perception and learning) to the specific (e.g., autonomous vehicles, etc. as mentioned earlier).

Among all the subfields of AI, constraint programming (CP) is undoubtedly one of the most prominent and fundamental research domains. CP is a potent technique used to tackle combinatorial search problems that naturally arise in most areas of human endeavor. Numerous computational problems in artificial intelligence, computer science, mathematics, operations research, and even biology can be formulated as constraint satisfaction problems (CSPs). By declaratively stating a problem as a set of constraints, a CP solver is employed to find one or all solutions of the problem automatically. A constraint in CP is essentially a restriction or relation defined over a number of decision variables. Some simple examples: a person cannot have two mothers simultaneously; the interior angles of a quadrilateral must add up to 360 degrees; $a \cdot b + c \cdot d = 100$, where $a, b, c, d \in \{0, 1, 2, \ldots, 200\}$.

Real-world problems are generally much more complicated than the examples mentioned above, involving a few hundred variables and constraints or even more. And not only that, the problems modeled as a CSP are often computationally intractable (*NP-complete* or *NP-hard*), implying that the computation time required to solve the problem grows exponentially as the problem size increases. That is to say, if a CSP is NP-complete or NP-hard, we cannot find such an algorithm that could solve the problem in polynomial time (assuming NP$\neq$P). However, it does not mean that it is totally

hopeless trying to solve such a problem. In fact, we can utilize special properties of the problem to develop general algorithms that are suitable for as many problems as possible. Over the past three or four decades, much research efforts in the constraints community have been committed to developing and enhancing general algorithms for solving CSPs, including adopting general search strategies, designing incremental filtering algorithms, introducing local search algorithms, identifying tractable subclasses of the problem, etc. In addition to improvements in algorithmic efficiency, the speed of constraint solvers also improves as processors become better developed.

Nevertheless, the recent quick fade of Moore's Law forces the computer design to turn to utilize multiple processors rather than to increase the performance of uniprocessors. And this change makes it impossible for sequential constraint solving to keep benefiting from the performance improvement of uniprocessors. Hence, the broad adoption of parallel architectures promotes the urgent need for more comprehensive and profound researches on parallel constraint solving. This dissertation is concerned with how constraint solving can exploit parallelism to attain a first solution for computationally hard problems that cannot be solved by the traditional sequential constraint solving. We first look back on the evolution of parallel constraint solving in the last 20 years and try to identify the most promising design choices. We then improve the existing parallel approach in order to solve new instances and solve existing instances faster. Moreover, we propose a new parallel stochastic portfolio search method that aims at enhancing the current single-solver based portfolio search. Finally, a dedicated hypergraph decomposition method used to distribute constraints to parallel processors is presented for parallel constraint solving.

1.1 Contributions and organization

We divide this dissertation into eight chapters. In this section, we sum up the contents and contributions of each chapter. The rest of the dissertation is organized as follows:

Chapter 2. This chapter reviews the background knowledge required for reading this dissertation. We give an introduction to constraint propagation and backtracking search, two fundamental approaches to constraints processing. We also discuss symmetry in constraints, including its definition and symmetry-breaking methods. Finally, some basic notions of parallel computing are discussed, especially for Amdahl's Law and Gustafson's Law. We provide our derivation showing that the two laws are related by an equation.

Chapter 3. This chapter surveys the extensive literature on parallel constraint solving. These studies are grouped into four categories, including parallel constraint propagation, parallelizing the search process, portfolios, and hybrid approaches. In this chapter, we attempt to provide a clear view of the evolution of techniques of parallel constraint solving while giving a lucid explanation of the idea of these techniques.

Chapter 4. The backtracking search systematically traverses the search tree of a given constraint satisfaction problem in a depth-first manner. The advantage of a backtracking style search is that we can always ensure that the resolution process finds a solution or determines the unsatisfiability (i.e., no solution). Nevertheless, we may end up missing a potential solution after a time limit that circumstance allows us to use when solving a computationally hard problem. Such a situation often implies that the systematic backtracking search gets stuck in an empty subtree (no solution) that is too large to visit exhaustively. In this chapter, we summarize the existing techniques for addressing this issue in the context of sequential solving. Then, the rationale of addressing this issue for the use of parallel constraint solving is also provided. Finally, we analyze the situations under which we can expect to gain superlinear speedups.

Chapter 5. In this chapter, we test our theoretical analysis by conducting empirical studies. The way of modeling a problem has a significant effect on how efficiently the problem can be solved. Hence, we put much effort into improving or redesigning the constraint models for these problems. Moreover, to apply the embarrassingly parallel search (EPS) approach to these problems, we design customized search space splitting methods for each problem. Our approaches allowed us to attain solutions for some open instances of the social golfer problem (SGP) [89] that can be formulated as a CSP. For the SGP, superlinear speedups were observed when comparing the EPS approach to sequential constraint solving with the same improved model, in line with our theoretical analysis in the previous chapter. The EPS approach could also achieve better performance than does the sequential run when solving the two constraint optimization problems: the traveling tournament problem with predefined venues [160] and the talent scheduling problem [190].

Chapter 6. One way for constraint solving to exploit parallelism is to run multiple sequential solvers with different parameter settings on the same problem

simultaneously. This method is often called parallel portfolios, which has the advantage of requiring no communication and achieving an excellent level of load balancing. One challenge of applying parallel portfolios is to devise a scalable source of diverse sequential solvers that contributes to orthogonal performance and complementary interest. In this chapter, we present our new parallel stochastic portfolio search, which aims to exploit massively parallel processing. We empirically evaluated our new portfolio approach for three problem classes taken from CSPlib (a benchmark library for constraints), demonstrating promising results.

Chapter 7. If the hypergraph of a CSP has a hypertree with bounded width, the initial intractable problem can be partitioned into several tractable subproblems, implying that we can solve the initial intractable problem in polynomial time [81]. Here, a new dedicated hypergraph decomposition method *det-k-CP* is presented for parallel constraint solving. The result of *det-k-CP*, which conforms with the four conditions for ensuring that the decomposition of a hypergraph is a hypertree, can be used to allocate constraints of a given constraint network to parallel processors. Our benchmark evaluations have shown that *det-k-CP* can relatively evenly decompose a hypergraph for the particular scale of constraint networks.

Chapter 8. This chapter concludes the dissertation.

Chapter 2

Background Information

This chapter provides some background information on constraint programming (CP) and parallel computing required by the subsequent chapters. Section 2.1 introduces the building blocks in the CP, including constraint propagation and backtracking search. Special focus is given to symmetry in the CP. In Section 2.2, we briefly review a classification of parallel computing related to parallel constraint solving presented in this dissertation and two fundamental laws about theoretical speedup for parallel computing.

2.1 Constraint Programming

We start with formal definition of **Constraint Satisfaction Problems** (CSP) that are problems modeled and solved by CP.

Definition 1. *A CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, D, C \rangle$ such that the following properties are satisfied:*

- $X = \{x_0, \ldots, x_n\}$ *is a finite set of decision variables,*
- $D = \{D_{(x_0)}, \ldots, D_{(x_n)}\}$ *contains associated finite domains for each variable in* X*, where any variable* x_i *can take on values in its domain* $D_{(x_i)}$*, and*
- $C = \{c_0, \ldots, c_t\}$ *is a collection of constraints, where each constraint* $c_i \in C$ *is a relation defined over a subset of* X*, and restricts the values that can be simultaneously assigned to these variables.*

If a constraint c_i is defined on a subset of variables $Scope(c_i)$, $Scope(c_i) \subseteq X$, we call $Scope(c_i)$ the *scope* of constraint c_i. A solution of a CSP $\mathcal{P}$ is a full instantiation satisfying all the constraints of $\mathcal{P}$. In a given task, one may need to search a first

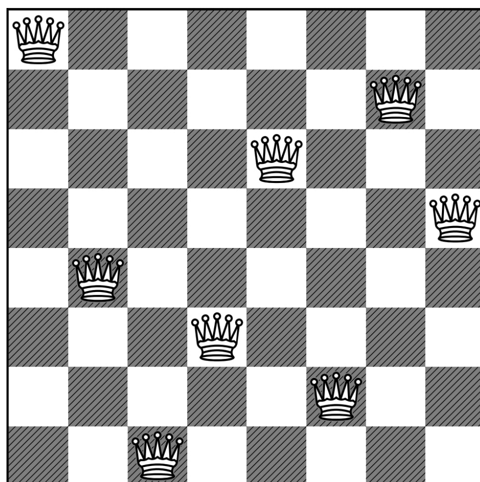


Figure 2.1: A possible solution of the 8-queens.

solution or all solutions or to determine the problem is satisfiable. A CSP is *unsatisfiable* if there is no solution for it. Please note that we consider only the finite discrete CSPs in this dissertation, although there are other possible definitions for CSPs that include variables with infinite, even continuous, domains.

The procedure of finding a solution for a CSP can be generalized to two steps: modeling and resolution process. In practice, a problem $\mathcal{P}$ can be modeled as a CSP with several **global constraints** that describe the restrictions and rules defining the solutions to $\mathcal{P}$. For a CSP $\mathcal{P}$, there are often various modeling approaches, which can strongly influence the run-time behavior, or whether it can be solved in the amount of time we actually are willing to wait. Obviously, we typically require a model that is able to solve the problem in the shortest run-time. Having defined the model of a CSP $\mathcal{P}$, the resolution process is performed by interleaving backtracking search with constraint propagation.

Before we introduce constraint propagation and backtracking search, consider a well-known CSP, the N-Queens problem (i.e., problem 054 in CSPLib [101]). The problem consists of finding a placement of n queens on an $n \times n$ $(n > 2)$ chessboard so that none of the queens can attack each other. In chess, two queens cannot attack each other if they are not in the same row, column, or either diagonal as itself. A solution of the 8-Queens problem is depicted in Figure 2.1. The problem can be represented as a CSP by using eight variables $\{x_1, \ldots, x_8\}$, each having the domain $\{1, \ldots, 8\}$. A variable x_i can denote the position of a queen put in either ith row or ith column of the chessboard. For example, if we let variable x_j stands for column j, the solution shown in Figure 2.1 corresponds to the list of values $[1, 5, 8, 6, 3, 7, 2, 4]$. The constraints

derived from the problem definition can be stated mathematically as follows:

$$\forall_{(i,j \in \{1,\ldots,8\}) \wedge (i<j)} \; (x_i \neq x_j) \quad (2.1)$$

$$\forall_{(i,j \in \{1,\ldots,8\}) \wedge (i \neq j)} \; (\mid i - j \mid \neq \mid x_i - x_j \mid) \quad (2.2)$$

The simplest and easiest method of solving the N-Queens problem is to search exhaustively by the **generate-and-test algorithm** (i.e., brute force algorithm). Suppose that the each of the n variable domains has size d, the overall time complexity of the generate-and-test algorithm is, therefore, $\mathcal{O}(d^n)$. Hence, the search space of the generate-and-test algorithm grows exponentially and thus could not work for instances with large n in practice.

2.1.1 Constraint Propagation

It is practically impossible to tackle NP-complete or NP-hard problems by using the generate-and-test algorithm. Constraint propagation[1], which is fundamental to the resolution process of solving a CSP [17], does a specific type of inference that reduces the search space efficiently.

Most of constraint propagation techniques concentrate on modifications of the domains of variables, which is called *domain-based* constraint propagation. The constraint propagation can be considered as operating over the network of constraints determined by a CSP [162]:

- Each variable of the CSP can be treated as a node of the network.
- Each constraint of the CSP can also be treated as a node.
- A set of possible values $D_{(x)}$ is associated with each variable x.
- There exists an arc between every variable x and the constraint c that is stated over this variable, denoted by $\langle x, c \rangle$.

where the network of constraints is also called a **constraint network**. The main idea of the constraint propagation is to enforce **local consistency** in each part of constraint network by removing inconsistent values throughout the entire constraint network. There are different levels of consistency enforced on the constraint network,

[1]Depending on authors, periods, and contexts, different names may also refer to the constraint propagation, including constraint inference, filtering algorithms, narrowing algorithms, rules iteration, local consistency enforcing, consistency-enforcing algorithms, and consistency algorithms (see for example [40].).

including **node consistency**, **arc consistency**, **path consistency** [146], and **k-consistency** [54]. A variable is node-consistent if every value in its domain does not violate any unary constraint of the variable. Similarly, a variable is arc-consistent if every value in its domain is consistent with the constraints whose scopes cover this variable and other variables. Higher-order consistency techniques, such as k-consistency, take into account k variables at a time and thus remove more values than does arc consistency.

Among all the levels of consistency, arc-consistency plays an essential role in constraint inference due to the following reasons: First, almost all the state of the art constraint solvers have implemented it. Second, improvements in arc-consistency efficiency can benefit other local consistency algorithms [17]. Hence, we shall look at arc-consistency in more detail. Also, since a constraint network consisting only of binary constraints is unusual in practice, the constraint network is non-binary by default; thus, we do not differentiate between **generalized arc consistency** (GAC) and arc consistency.

We now give a formal definition of the GAC, which was proposed by Mackworth [130].

Definition 2. *Given a constraint network $\mathcal{N}$ formed by a CSP $\mathcal{P} = \langle X, D, C \rangle$, a constraint $c_i \in C$, and a variable $x_j \in Scope(c_i)$.*

- *A value $v \in D_{x_j}$ is consistent with c_i iff there are a list of values $v_0, \ldots, v_k$, such that $c_i(x_j = v, x_0 = v_0, \ldots, x_k = v_k)$ is satisfied. Such a list of values is called a support for (x_j, v) on c_i.*

- *The variable x_j is arc-consistent relative to c_i, namely, arc $\langle x_j, c_i \rangle$ is arc-consistent, iff $\forall v \in D_{(x_j)} \wedge x_j \in Scope(c_i)$, there exists a support for (x_j, v) on c_i.*

- *The constraint c_i is generalized arc consistent, iff every variable in its scope is arc-consistent relative to c_i.*

- *The constraint network $\mathcal{N}$ is generalized arc consistent iff all its constraints are generalized arc consistent.*

Example 1. Let us consider a simple constraint network $\mathcal{N}$ involving three variables x, y, and z, with domains $D_{(x)} = \{1, \ldots, 6\}$, $D_{(y)} = \{1, \ldots, 10\}$, and $D_{(z)} = \{1, 2\}$, and the constraints $c1 \equiv x - 5 * y \geq 1$, $c2 \equiv z \neq x$, and $c3 \equiv z \neq y$. The constraint network is not arc-consistent because for each value $v \in D_{(x)} = \{1, 2, 3, 4, 5\}$ we

cannot find a corresponding value for y for which $x - 5 * y \geq 1$. Moreover, there are other inconsistent values; we will discuss them soon.

Constraint propagation is not merely the removal of inconsistent values; it is the process of propagating the domain reduction of a decision variable to all of the variables within the scope of the constraints that are stated over this variable. Therefore, this process can usually result in many more domain reductions. The constraint propagation process continues until no domain of the decision variables can be reduced. Or all the values in the domain of some variables are filtered out, indicating that a failure occurs. A CSP has no solution when an empty domain occurs during the initial constraint propagation.

Going back to the simple constraint network $\mathcal{N}$ in Example 1, there are six arcs: $\langle x, c_1 \rangle$, $\langle y, c_1 \rangle$, $\langle z, c_2 \rangle$, $\langle x, c_2 \rangle$, $\langle z, c_3 \rangle$, and $\langle y, c_3 \rangle$. To ensure every constraint arc-consistent, we maintain a set of arcs *to_do*. At the beginning of the propagation, the set has all the arcs of $\mathcal{N}$ in Example 1 (cf., *to_do* in Algorithm 1). Note that the data structure used here is a set because the processing order of the arcs will not affect propagation results and solutions of the network. Suppose we first select the arc $\langle x, c_1 \rangle$ from *to_do*. The domain reduction of the constraint c_1 shrinks the domain of x to $\{6\}$. No arc related to variable x is needed to be added into *to_do* because there is no other arc not in *to_do*. Suppose that $\langle y, c_1 \rangle$ is selected next. The domain of y is reduced to $\{1\}$. Again, no arc is added to *to_do*. For arcs $\langle z, c_2 \rangle$ and $\langle x, c_2 \rangle$, there is no inconsistent value. Marking arc $\langle z, c_3 \rangle$ consistent shrinks $D_{(z)}$ to the singleton $\{2\}$. Because the arc $\langle x, c_2 \rangle$ has already been calculated before and the scope of constraint c_2 covers the variable z whose domain has been modified, we must replace the arc $\langle x, c_2 \rangle$ into the *to_do* set. Finally, there are only two consistent arcs $\langle y, c_3 \rangle$ and $\langle x, c_2 \rangle$ in the *to_do* set; and no more domain reduction can be achieved. The constraint propagation ends up with the empty *to_do* set, and the final domains are: $D_{(x)} = \{6\}$, $D_{(y)} = \{1\}$, and $D_{(z)} = \{2\}$. Since the domains all have size one, we obtain a unique solution only by constraint propagation.

In many practical problems, it is uncommon for every domain to become a singleton after the initial constraint propagation. We now consider a more general example in which at least one domain contains multiple values after the initial constraint propagation.

Example 2. Let $\mathcal{N}$ be a constraint network having three variables x, y, and z associated with the constraints $c_1 \equiv x > y$ and $c_2 \equiv y > z$, with $D_{(x)} = D_{(y)} = D_{(z)} = \{1, 2, 3, 4, 5\}$. As with the previous example, all the arcs are included in the *to_do*

set at the beginning of propagation. Suppose arc $\langle x, c_1 \rangle$ is considered first. This arc is not arc-consistent because the value 1 in $D_{(x)}$ is not consistent with any value for y in $D_{(y)}$. Therefore, the value 1 is pruned from $D_{(x)}$, and $D_{(x)}$ becomes $\{2, 3, 4, 5\}$. Suppose arc $\langle y, c_1 \rangle$ is considered next; then $D_{(y)}$ is reduced to $\{1, 2, 3, 4\}$. Nevertheless, the arc $\langle z, c_2 \rangle$ could be added into the *to_do* set, but it is already in *to_do*. Suppose that $\langle y, c_2 \rangle$ is selected next, the value 1 is pruned from $D_{(y)}$, and thus $D_{(y)} = \{2, 3, 4\}$. Besides, we must put arc $\langle x, c_1 \rangle$ back to the *to_do* set because $D_{(x)}$ might be reduced further. Suppose arc $\langle z, c_2 \rangle$ is considered next, $D_{(z)}$ is reduced to $\{1, 2, 3\}$. Finally, we consider the arc $\langle x, c_1 \rangle$. Because the value $2 \in D_{(x)}$ is not consistent with c_1, we prune it from $D_{(x)}$, and thus $D_{(x)}$ becomes $\{3, 4, 5\}$. Since x is not involved in any other constraints, no arcs are added into the *to_do* set again. At last, the algorithm terminates with $D_{(x)} = \{3, 4, 5\}$, $D_{(y)} = \{2, 3, 4\}$, $D_{(z)} = \{1, 2, 3\}$.

We have used two examples to illustrate how exactly the generalized arc consistency (GAC) algorithm works. Algorithm 1 represents the main procedure of the GAC, which is also called AC-3 by the inventor of the algorithm Mackworth since it is the third version presented in his paper [130]. Some textbooks, such as [182] and [40], present the GAC-3 algorithm based on pairs of variables. Here, we follow the way of representing an arc by pair consisting of a variable and a constraint used in the textbook written by Mackworth [162]. GAC-3 has a $\mathcal{O}(er^3d^{r+1})$ time complexity, where e, r, and d are the number of constraints, greatest arity among constraints, and largest domain size, respectively [17]. We refer to Chapter 4 of [114] for a more extensive discussion on GAC-3 and its time and space complexity.

The GAC-3 can be viewed as arc-oriented propagation algorithm [17, 134]. The constraints community has also developed other ways of propagating constraints, including AC-4 [142], AC-6 [16, 18], AC-2001 [20], and their non-binary versions, GAC-4 [143] and GAC-2001 [21], where AC-4 and AC-6 are value-oriented.

2.1.2 Backtracking Search

In Section 2.1.1, we have shown that a unique solution could be obtained by arc consistency when all the variables of a CSP $\mathcal{P}$ becomes a singleton, or disprove the CSP $\mathcal{P}$ has a solution when one domain is wiped out. More generally, however, not all values remaining in the domains participate in solutions after enforcing GAC. In that case, we usually resort to the backtrack search.

The naive backtrack search (BT) is the foundation of other more sophisticated backtracking searches (e.g., Forward checking, Conflict-directed backjumping, etc.). In

Algorithm 1: Generalized arc consistency algorithm (GAC-3)

Input: a constraint network $\mathcal{N}$ formed by a CSP $\mathcal{P} = \langle X, D, C \rangle$

Output: returns false whenever an inconsistency is detected and true otherwise

Local variables: *revised*, *to_do*

```
   /* add all the arcs in N into the to_do set */
1  to_do = {⟨x_j, c_i⟩ | x_j ∈ Scope(c_i) ∧ c_i ∈ C ∧ x_j ∈ X};
   /* Constraint propagation starts */
2  while to_do ≠ ∅ do
3      choose and remove ⟨x_j, c_i⟩ from to_do ;
4      revised ← false ;
5      foreach v ∈ D_(x_j) do
6          if no support for (x_j, v) on c_i then
               /* remove inconsistent v from D_(x_j) */
7              D_(x_j) ← D_(x_j) \ v ;
8              revised ← true ;
9          end
10     end
11     if revised then
12         if D_(x_j) = ∅ then
               /* a domain is wiped out */
13             return false;
14         end
           /* add all the related arcs into to_do again */
15         to_do ← to_do ∪ {⟨x_i, c_j⟩ | {x_i, x_j} ∈ Scope(c_j) ∧ c_j ≠ c_i ∧ x_j ≠ x_i} ;
16     end
17 end
18 return true;
```

a BT search tree, the start node at level 0 (the root) is an empty assignment, and a node at level i stands for a series of assignments $\{x_1 = v_1, \ldots, x_i = v_i\}$. At each node of the BT search tree, the BT selects an uninstantiated variable and extends this node by assigning the variable all the possible values in its domain, which leads to the branches out of this node, each of which stands for a choice for this variable. Whenever a value in the domain of the current variable is inconsistent with a constraint in the network, the next value in its domain is checked. If no domain value remains, the BT backtracks. Furthermore, when none of the variable's values are consistent with the current assignment, this situation referred to as a **deadend**. A solution of the BT tree is a complete instantiation to all variables with the satisfaction of all constraints. Figure 2.2 shows a part of the backtrack search tree generated by the BT

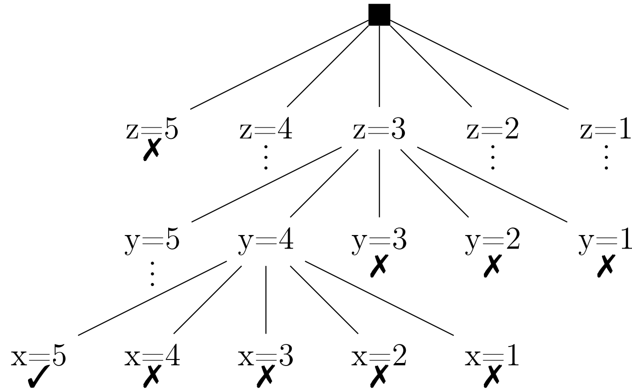


Figure 2.2: A fragment of the BT search for the constraint network $\mathcal{N}$ of Example 2. The potential node which is pruned because it violates constraints are labeled with ✗. A ✓ sign indicates that a solution is found.

for the constraint network $\mathcal{N}$ in Example 2 (page 9). In this example, we assume a static variable ordering [z,y,x], and the value ordering heuristic is to consider upper bound first.

The backtracking search in CSP is essentially a depth-first search that progressively extends a node at a time and backtracks whenever the domain of a variable is empty. Theoretically, we could apply a search alone to solve a CSP $\mathcal{P}$. However, the search space would be too big to handle. For a CSP $\mathcal{P}$ having n variables with domain size d, the branching factor at the top level is $n \cdot d$ since any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1) \cdot d$, and so on for the rest of $n-2$ levels. Consequently, naive BT generates a search tree with $n! \cdot d^n$ leaf nodes, even though only d^n possible assignments exist. Thus, constraint solving interleaves backtracking search and a certain level of constraint propagation rather than performing either search or constraint propagation alone.

In general, the more tight constraint networks are (i.e., enforcing a stronger level of local consistency), the more restricted search space will be. After applying a stronger level of consistency, the search encounters fewer deadends and becomes more efficient. But the computational cost is also increased while enforcing a stronger consistency level. To take an extreme example, one would expect that every value in the domain of every variable participates in solutions of the CSP $\mathcal{P}$ after propagation so that the search would be backtrack-free (i.e., no deadends). Unfortunately, achieving such a high level of consistency is at least as difficult as solving the CSP $\mathcal{P}$ [201], and frequently too hard, probably requiring an exponential number of additional constraints [40].

Algorithm 2: A simple backtracking search

```
   Input: a constraint network 𝒩 formed by a CSP 𝒫 = ⟨X, D, C⟩
   Output: return failure when the CSP has no result otherwise a solution
 1 backtrackingSearch(𝒩):
 2   return backtrack(𝒩,{}) ;
 3 backtrack(𝒩,assignment):
 4   if assignment is a solution then return assignment ;
     /* select an uninstantiated variable by an user-specified
        variable ordering heuristic */
 5   x_j ← selVar(𝒩) ;
     /* try all values with the priority specified by an
        user-specified value ordering heuristic */
 6   foreach v_j ∈ selVal (D_(x_j),𝒩) do
 7     if v_j is consistent with assignment then
         /* add { x_j = v_j } to the assignment */
 8       assignment ← assignment ∪ {x_j = v_j};
 9       propagation ← propagation (𝒩,x_j,v_j) ;
10       if propagation ≠ failure then
           /* if variables become singleton after propagation */
11         add propagation to assignment ;
12         result ← backtrack(𝒩,assignment) ;
13         if result ≠ failure then
14           return result
15         end
16       end
         /* remove value assignments when v_j leads to failure,
            including propagation and backtrack */
17       assignment ← assignment \ {x_j = v_j, x_k = v_k, . . .}
18     end
19   end
20   return failure;
```

Algorithm 2 (adapted from [95, 182]) depicts a simple backtracking search based on the recursive depth-first search for the CSP. The algorithm iteratively selects an uninstantiated variable, and then enumerates all the domain values of the variable, trying to obtain a solution. By varying the functions `selVar` (line 5) and `selVal` (line 6), one can specify a variable ordering heuristic and a value ordering heuristic. In line 8, if a chosen value for the selected variable does not violate any constraint in the network, we then add that value to the assignment. The function `propagation` (line 9) can be used to enforce a certain level of consistency (e.g., arc- or path-consistency). If the propagation succeeds and at least one domain becomes singleton,

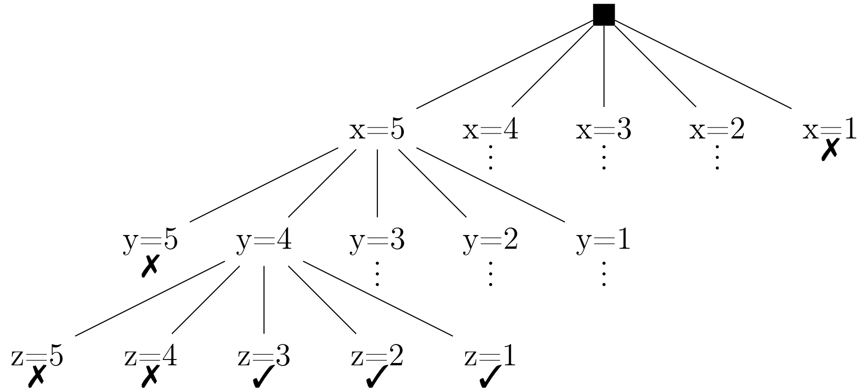


Figure 2.3: Another fragment of the BT search for the constraint network $\mathcal{N}$ of Example 2.

we should also append the value of the fixed variable to the *assignment*, as shown in line 11 of Algorithm 2. If either propagation or backtrack encounters failure due to a value choice, then the *assignment* must be restored to the previous *assignment* by removing that value (line 17); and then a new value is considered in line 6.

Although the order of variable and value does not affect the results of CSPs, the search strategy, including the variable and value selection heuristic, can be critical to efficiently solving the CSPs. For the example of the constraint network $\mathcal{N}$ of Example 2 (page 9), if the variable selection heuristic is set to a static variable ordering [x,y,z], and the value selection heuristic always chooses the upper bound of the variable, we will see the BT tree as shown in Figure 2.3.

In Figure 2.3, the BT search only visits six nodes for obtaining a solution. In contrast, with changing the variable ordering, the BT search needs to visit much more nodes to obtain a first solution, as shown in Figure 2.2. This example illustrates that a better variable selection heuristic can enhance the performance of backtrack search by reducing the exploration of a great number of nodes early in the search. A better value selection heuristic can also help reduce the number of nodes visited for finding a first solution. Let us assume we have a better value selection heuristic for the example illustrated in Figure 2.3. Even using the same variable ordering, we only need to explore three nodes to obtain a first solution, as shown in Figure 2.4. However, the value ordering is irrelevant when our goal is to obtain all solutions. Because in this case, every value is required to be enumerated.

Yet, it is a non-trivial task for us to find an optimal ordering. Generally, the dynamic ordering heuristics are more efficient than the static ordering heuristics. The static ordering heuristics is relatively simple: the variable ordering is fixed before

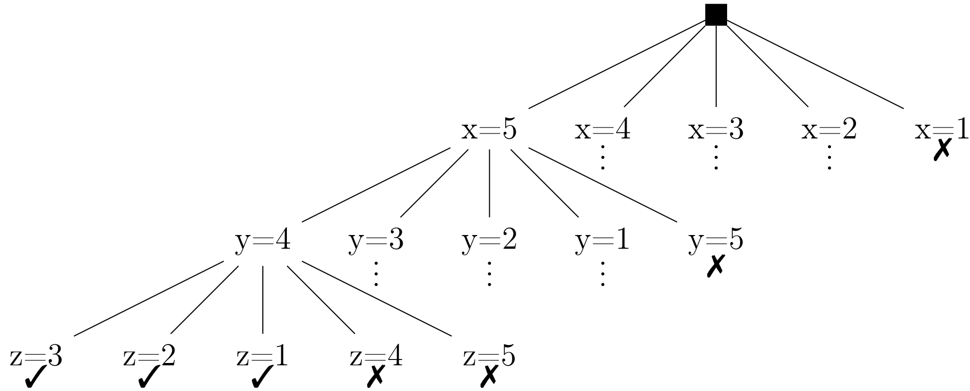


Figure 2.4: A fragment of the BT search formed for the constraint network $\mathcal{N}$ of Example 2, using a better value ordering heuristic.

search (e.g., lexicographic order). In contrast, the dynamic ordering heuristics decide which variable to be selected based on a criterion that is calculated during search. In [201], Van Beek presents a taxonomy that classifies variable ordering heuristics into two groups: heuristics utilizing the domain sizes of the variables and heuristics utilizing the structure of the CSP. In 1965, Golomb & Baument [67] pioneered the idea of utilizing domain size for variable ordering, where a variable with the fewest values left in its domain is always chosen first. More variable ordering heuristics fall into this category will be discussed in Section 6.2. The second category, structure-guided variable ordering heuristics, exploits the graphical representation of the constraint network. For instance, Dechter & Pearl [43] proposed a variable selection heuristic that first instantiates variables that disconnect cycles in the constraint graph of a given CSP. Unfortunately, this type of variable ordering tends to be static or nearly static and cannot compete with domain size based dynamic variable ordering heuristics.

Backtrack search systems are composed of four components [114]: *branching* (how and which decisions to take to go forward to a solution), *propagation* (how and which consistency level to enforce to prune the search space at each step), *backtracking* (how to retreat when encountering a deadend), and *nogood learning* (what information to gather during search to facilitate the subsequent search). Much effort from the constraints community has gone into improving the efficiency of backtracking search, including more sophisticated application-independent heuristics, exploring the right combinations of propagation and backtracking techniques, nogood learning during backtracking search, and non-chronological backtracking. We have introduced the

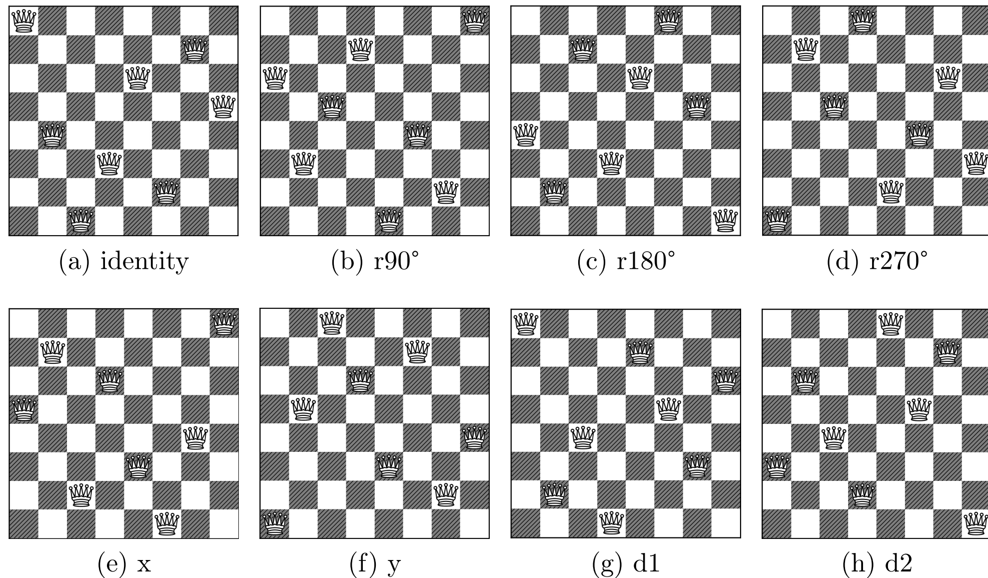


Figure 2.5: The 8 symmetries of a solution of 8-Queens.

first three notions in this section. For further reading on these topics, we recommend [201, 115, 117, 66, 202], and Chapter 8 of [114].

2.1.3 Symmetry in Constraint Programming

Many CSPs exhibit symmetrical and combinatorial nature simultaneously. The CP, a powerful technique to tackle combinatorial problems, sometimes has to deal with the amount of extra search space due to symmetry.

To help understand the concept of symmetry in the CP, we first reconsider the 8-Queens problem shown in Figure 2.1 (page 6). Figure 2.5 depicts a solution of 8-Queens (i.e., the identity symmetry shown in Figure 2.5a) and its seven symmetries, each of which is also a solution of the 8-Queens. The identity symmetry can be rotated by 90°, 180°, and 270° in a clockwise direction, as shown in Figures 2.5b, 2.5c, and 2.5d. Besides, the identity symmetry can also be reflected in the vertical axis, horizontal axis, as well as two main diagonal axes, as shown in the second row of Figure 2.5. In the context of group theory, these eight symmetries, four rotational symmetries and four reflection symmetries, form the dihedral group $\mathbf{D}_4$, since a chessboard is a regular polygon with four sides.

For the CP, the importance of studying symmetry is that we can avoid exploring the subtrees that contain the symmetries of the explored subtree so that less search effort is required to find a solution, or to disprove satisfiability, thereby dramatically enhancing the performance of constraint solving. The reduced search space comes from

two aspects: First, when a solution is found, the symmetric solutions can be obtained automatically without exploring symmetric subtrees (e.g., any solution in Figure 2.5 can readily lead to the rest of seven solutions without search effort). Hence, the symmetric fruitful subtrees of the solution can be omitted. Second, when a consistent instantiation leads to a deadend, the consistent instantiations which are symmetric to it are bound to lead deadends as well. Similarly, the symmetric fruitless search subtrees of the partial solutions can also be omitted.

The constraints community has summarized two types of definition for symmetry on a constraint network: *problem symmetry*, also known as *constraint symmetry*, and *solution symmetry* [62, 34]. Roughly speaking, problem symmetry defines symmetry as a property derived from the problem statement; and solution symmetry defines symmetry as a property based on the solution set.

The notion of solution symmetry is relatively simple, and solution symmetry has two special cases: *variable symmetry* and *value symmetry.* We can often observe value symmetry in CSPs. The nurse rostering problem is a typical example. To devise a periodic (e.g., monthly) duty roster for nursing employees; one must assign nurses (i.e., values) to shifts (i.e., variables). A feasible solution remains to be feasible when interchanging nurses with the same skills (i.e., interchanging values while keeping the ordering of variables fixed). Hence, it is not hard to conclude that a value symmetry is a permutation of the values that preserves solutions. A variable symmetry, on the other hand, is a permutation of the variables that preserves solutions. For two interchangeable variables, both of them must have the same domain, and all constraints must cove them simultaneously. Consider a simple constraint network with only one linear constraint (e.g., $c_1 \cdot x_1 + c_2 \cdot x_2 +, \ldots, +c_n \cdot x_n = 100$). A solution to this network preserves when permuting the variables with the same domain and the same coefficient, i.e., $c_i = c_j \wedge D_{(x_i)} = D_{(x_j)}$. If we consider both variable and value together as a $\langle variable,\ value \rangle$ pair, variable and value symmetry can be generalized to solution symmetry. A solution symmetry is a permutation of the $\langle variable,\ value \rangle$ pairs that preserves solutions.

The definition of problem symmetry also acts on $\langle variable,\ value \rangle$ pairs, but the emphasis is on a consequence of leaving the constraints unchanged, rather than preserving solutions. The rigorous definition of problem symmetry proposed by Cohen *et al.* [34] is defined over *the microstructure complement* of a CSP. The microstructure complement is a hypergraph, the vertices of which have all $\langle variable, value \rangle$ pairs, and a hyperedge of which stands for an assignment disallowed by a constraint or consists of a pair of incompatible assignments for the same variable. The constraints are

preserved since only an automorphism[2] of a microstructure complement is a problem symmetry. Therefore, for a k-ary CSP instance $\mathcal{P}$ (one whose constraints have maximum arity k), k-ary nogoods, which are consistent assignments of up to k variables and cannot lead to a solution of $\mathcal{P}$, can cause solution symmetries but not problem symmetries. The reason is that k-ary nogoods cannot be expressed as a hyperedge of a microstructure complement since only disallowed assignments can form a hyperedge of a microstructure complement. But, the hypergraphs containing the hyperedges that represent these k-ary nogoods are far more than the microstructure complement. So, in general, solution symmetries far outnumber problem symmetries for a given CSP $\mathcal{P}$. Interestingly, Cohen *et al.* [34] show that eliminating problem symmetry can result in the number of solution symmetry increased for the N-Queens problem. Conversely, problem symmetries rise sharply for 5-Queens from 8 to 28,880 after applying path consistency.

In theory, we can generate all the problem symmetries for a given k-ary CSP instance $\mathcal{P}$ by finding the automorphism group of the microstructure complement. Moreover, we can also expand the problem symmetries to the solution symmetries by adding all the nogoods with k or less than k values to the microstructure complement. By doing so, symmetries can be identified automatically. Unfortunately, identifying the satisfying k-ary nogoods may itself be intractable for a large CSP instance $\mathcal{P}$ [19, 34]. To the best of our knowledge, symmetry in CSPs is still mostly detected by applying human insight; and the most published works related to symmetry-breaking presuppose that symmetries are given.

The research on symmetry-breaking has been an active area for the CP. There exist three main categories of approaches for symmetry-breaking: (1) reformulation, (2) breaking symmetry statically, and (3) breaking symmetry dynamically. The reformulation techniques involve remodeling the problem, introducing set variables, etc. Nevertheless, it is hard to generalize the reformulation techniques, and applying the reformulation techniques usually requires considerable insight into problems [62]. But the reformulation techniques have the advantage of combining easily with other symmetry-breaking approaches.

Adding static constraints before search is perhaps the most natural technique for symmetry-breaking. Despite its simplicity, it can be challenging to eliminate all the symmetry by using a constraint or a set of constraints. Sometimes, improperly imposing symmetry breaking constraints on a model might lose solutions. Several

[2]In the context of graph theory, an automorphism of a graph or hypergraph is a bijective mapping of its vertices while preserving the edge-vertex connectivity.

types of constraints are commonly used to break symmetry, including the *arithm* constraint, and the *lexLessEq* constraint. For instance, in [126, 191], the *arithm* constraint is used to eliminate the reversal of a solution. The *lexLessEq* constraint (i.e., $\preceq_{lex}$, *lex-leader*, or the *lexicographic ordering* constraint) lies at the heart of most static methods for breaking variable symmetries [207, 62]. The constraint is defined on two vectors $\overrightarrow{x} = \langle x_1, x_2, \ldots, x_q \rangle$ and $\overrightarrow{y} = \langle y_1, y_2, \ldots, y_q \rangle$ of variables, and ensures that $\overrightarrow{x}$ is lexicographically less or equal than $\overrightarrow{y}$, i.e., $\overrightarrow{x} \preceq_{lex} \overrightarrow{y}$. Formally, we have $\overrightarrow{x} \prec_{lex} \overrightarrow{y}$ iff $\exists_{1 \leq i \leq q}$ such that $\forall_{1 \leq j < i}$, $(x_j = y_j \wedge x_i < y_i)$ and $\overrightarrow{x} \preceq_{lex} \overrightarrow{y}$ iff $\overrightarrow{x} = \overrightarrow{y} \vee \overrightarrow{x} \prec_{lex} \overrightarrow{y}$. In fact, the lexicographic ordering guaranteed by the *lexLessEq* constraint is identical to that of standard in computer science, e.g., $\{12345, 12354, 12435, 12453\} \prec_{lex} 12534$ and $\{12345, 12354, 12435, 12453, 12534\} \preceq_{lex} 12534$. Returning to the 8-Queens problem shown in Figure 2.5, the identity solution can be written as $\{\langle x_1, 1\rangle, \langle x_2, 5\rangle, \langle x_3, 8\rangle, \langle x_4, 6\rangle, \langle x_5, 3\rangle, \langle x_6, 7\rangle, \langle x_7, 2\rangle, \langle x_8, 4\rangle\}$. We can prune symmetrically-equivalent solutions of the identity solution by posting $\overrightarrow{x} \prec_{lex} 15863724$ constraint. Thus, the rest of the solutions are ignored by backtrack search because $\{36428571, 57263148, \ldots\} \not\prec_{lex} 15863724$.

Nevertheless, the *lexicographic ordering* constraint relies heavily on the variable and value selection heuristics. More specifically, if the leftmost solution is not canonical, not all symmetrically equivalent solutions are guaranteed to be pruned. For example, reconsider the 8-Queen problem shown in Figure 2.5, we find the solution $\{57263148\}$ first when applying a different value ordering heuristic. Thus, some solutions, such as $\{36428571, 15863724\}$, cannot be removed by imposing the lexicographic ordering constraint since $\{36428571, 15863724\} \not\prec_{lex} 57263148$. This is in contrast to the dynamic symmetry breaking methods such as SBDS and SBDD, which do not depend on the heuristic. SBDS, an acronym for symmetry breaking during search, was named by Gent & Smith in [64]. However, this naming might trick us into thinking that SBDS is a general technique for symmetry breaking during search. In fact, other ways for breaking symmetry during search, such as symmetry breaking via dominance detection (SBDD), have already been proposed.

The idea behind SBDS is to impose symmetry breaking constraints on a model whenever backtracking from a search decision so as to not revisit the symmetric equivalent of that search decision. A search decision can be viewed as an assignment to a variable, i.e., $var = val$. Let us consider the 8-queen problem again and assume the first search decision in the search tree is $x_1 = 1$, i.e., the queen is placed on position 1 of the first column of chessboard (see Figure. 2.5a). When search backtracks to the root of the search tree, all the possible symmetric equivalent to $x_1 = 1$ should be

avoided. Thus, we ought to post all the symmetric constraints of $x_1 \neq 1$, denoted by $(x_1 \neq 1)^g$, where $g = r90°$, $r180°$, $r270°$, x, y, $d1$, and $d2$. Nevertheless, it is unnecessary to post the symmetric constraints for x, y, $d1$, and $d2$ in the group since these constraints have added by the symmetric constraints for $r90°$, $r270°$, and the branching constraint already (cf. the right branch of Figure 2.6).

Although the notion of SBDS is simple, we should be careful to post constraints as the search tree expands progressively. According to the solution shown in Figure 2.5a, we assume that the next search decision is $x_2 = 5$. We do not post symmetric constraints for this positive decision. However, if we backtrack from $x_2 = 5$ and before turning into branch $x_2 \neq 5$[3], we must post symmetry breaking constraints, but not the constraints like $(x_2 \neq 5)^g$. Because posting a constraint such as $x_4 \neq 2$ ($(x_2 = 5)^{r90} \equiv x_4 = 2$) might rule out some non-isomorphic solutions of the current solution in some future states, if the current partial assignments cannot be extended to a solution. Therefore, we should impose a conditional constraint $x_1 = 1 \Rightarrow (x_2 \neq 5)^g$. Furthermore, not all of the possible symmetries in the group exist for the branches out of the node $x_1 = 1$. Specifically, simply adding constraints, such as $x_1 = 1 \Rightarrow (x_2 \neq 5)^x$, $x_1 = 1 \Rightarrow (x_2 \neq 5)^y$, and $x_1 = 1 \Rightarrow (x_2 \neq 5)^{d2}$, is redundant and incorrect since $x_8 = 1$, $x_1 = 8$, and $x_8 = 8$ are contradicted by the constraints of the CP model and the branching constraint (i.e., $x_1 = 1$). More precisely, $x_1 = 1$ has already indicated $x_8 \neq 1$, $x_1 \neq 8$, and $x_8 \neq 8$ in the branches out of the node $x_2 \neq 5$ because these squares are on the same row, column, diagonal. Besides, we cannot post the constraint $x_1 \neq 1$ for the symmetry *d1* at the branches out of the node $x_1 = 1$. Consequently, we actually add conditional constraints $x_1 = 1 \Rightarrow (x_2 \neq 5)^{r90}$, $x_1 = 1 \Rightarrow (x_2 \neq 5)^{r180}$, and $x_1 = 1 \Rightarrow (x_2 \neq 5)^{r270}$ at the node $x_2 \neq 5$.

SBDS is sound in that no solution is omitted, and only one solution is obtained from each symmetric equivalence space. Moreover, in contrast to the *lexLessEq* constraint, SBDS does not conflict with search heuristics. Also, we have to use symmetry breaking during search, because sometimes we cannot know the search decision before search. One insurmountable problem with SBDS is that when the CSP $\mathcal{P}$ becomes large, we are required to deal with a great number of symmetry breaking constraints.

SBDD, introduced by [49, 53], checks whether or not every node in the search tree entails symmetrically equivalent to one already visited. If this is the case, the branch is pruned. Unlike SBDS, SBDD does not depend on using symmetry breaking

[3]Both SBDS and SBDD are based on 2-way branching since 2-way branching is exponentially more efficient than d-way branching [102], and thus 2-way branching is much more frequently used in the competitive constraint solvers.

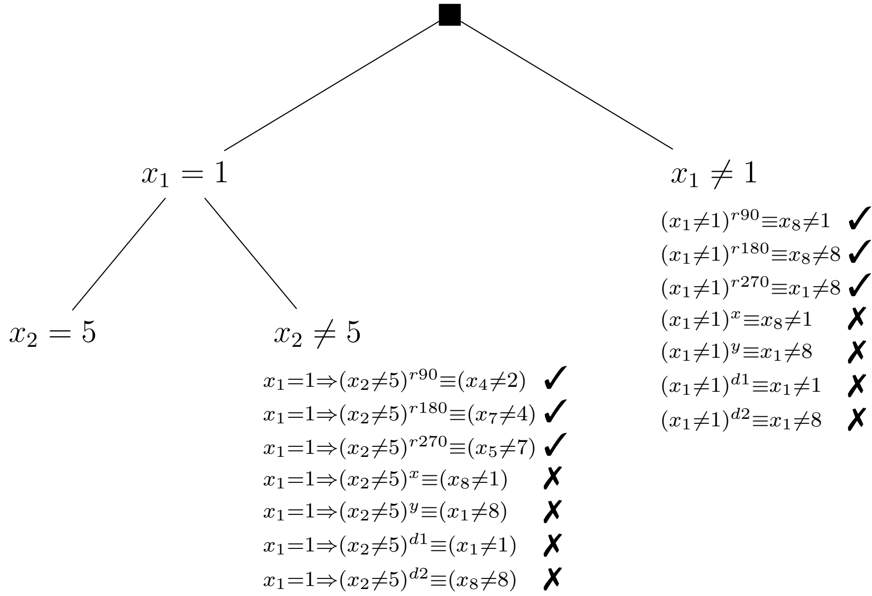


Figure 2.6: Example of SBDS on a backtrack search tree for the 8-Queens problem.

constraints but instead stores information about explored sub-trees and utilizes the dominance detection function to determine if a node is dominated by a previously explored one under some symmetry. Thus, SBDD never generates the children of dominated nodes.

We have introduced some basic notions and techniques related to symmetry in constraint programming. Some advanced and interesting topics (e.g., computational group theory, automatic symmetry detection, and symmetry and local search) are not discussed due to limited space. We encourage the interested reader to study [62, 114] for the missing details.

2.2 Parallel Computing

In 1965 Gordon Moore, who was working as the director of research and development at Fairchild Semiconductor,[4] predicted that the number of transistors in a dense integrated circuit would double approximately every year, which was amended in 1975 to every two years. This world-famous prediction, so-called *Moore's Law*, has been a roadmap for the semiconductor industry for about 50 years. Unfortunately,

[4]Fairchild Semiconductor company was a pioneer in the manufacturing of transistors and integrated circuits, and it has an important historical status in the development of the semiconductor industry. In the 1980s, about half of all the semiconductor companies were direct or indirect descendants of Fairchild, including Intel, AMD, Xilinx, etc. [100].

Moore's Law has almost reached saturation since process scaling has slowed beginning at the 22 nm feature, and thus cannot continuously reduce feature size. Moreover, clock speeds have tapered, as well as thermal and power dissipation envelopes have settled flat [158]. According to Hennessy & Patterson [93], since 2015 single processor performance improvement has been just 3.5% per year, or doubling every 20 years.

However, the demand for increasing performance of the microprocessor continues; thus, the microprocessor industry is motivated to shift to multiple cores or processors instead of a single core. In addition, apart from multi-cores and multi-processors within a single computer, multiple stand-alone computers can be used to build computing cluster, MPPs (Massively Parallel Processing), and grids to take advantage of parallelism (namely parallel processing) for improving performance. **Parallel computing** is to utilize multiple computing resources, such as multi-core microprocessor or a computing cluster, to solve a computational problem simultaneously. The general process of parallel computing can be summarized as follows:

1. A problem is partitioned into discrete sub-tasks, which can be solved concurrently.
2. Each sub-task is mapped into different processors.
3. Instructions from each sub-task execute simultaneously on different processors.
4. The final result is combined. Besides, an overall control or coordination mechanism might be employed during parallel processing.

In [93], the authors classify parallelism as two different types in applications:

- *Data-level parallelism (DLP)* requires that the problem has a large amount of data that can be operated at the same time. The parallelism is achieved by distributing data across processors, and then each processor performs the same task (e.g., executing the same code) on different data.
- *Task-level parallelism (TLP)* requires that the problem can be broken apart into many different tasks. The parallelism is achieved by executing a different task on the same or different data.

As will be discussed in Chapter 3, the techniques of parallel constraint solving can fit into the two categories. Besides, parallelism can also be classified according to the granularity of parallelism, i.e., the size of the computations being performed simultaneously between processors [188]. Coarse-grained parallelism works on a small

number of large tasks, and communication and synchronization are infrequent among processors. Fine-grained parallelism, by contrast, executes a large number of small tasks and often requires frequent communication or synchronization [156]. One extreme case, processors never or rarely communicate or synchronize to each other, is called embarrassing parallelism. In the following chapters, we will show that techniques based on embarrassingly parallel computation play an essential role in parallel constraint solving.

Several factors prevent parallelism from obtaining better performance. From the hardware perspective, the power wall of microprocessors and the slow improvement of transistors due to the slowing of Moore's Law cause deceleration in improvements of processor [93]. Besides, once the communication cost dominates computational cost, partitioning the workload over more processors will cause performance degradation instead of improvement.

2.2.1 Amdahl's Law and Gustafson's Law

Another reason is the limitation of speedup determined by the task itself, which is prescribed by *Amdahl's Law* [7]. In the context of parallel computing, speedup measures the performance gain that can be obtained by parallelism. In Amdahl's Law, the speedup $S_A(n)$ is defined as the ratio between the execution time of a sequential program $T_s(1)$ and the execution time of a parallel program $T_p(n)$ that completes the same task on n parallel processors; that is,

$$S_A(n) = \frac{T_s(1)}{T_p(n)} \tag{2.3}$$

For a given task, Amdahl's Law prescribes the speedup that we could expect on a multiprocessor computer or system. Assume a task consists of a parallelizable fraction f_{Ap} and a serial fraction $(1 - f_{Ap})$. Thus, the time required to process this task by a single processor and n parallel processors is given by:

$$T_s(1) = (1 - f_{Ap}) \cdot \tau + f_{Ap} \cdot \tau \tag{2.4}$$

$$T_p(n) = (1 - f_{Ap}) \cdot \tau + \frac{f_{Ap} \cdot \tau}{n} \tag{2.5}$$

where τ denotes the processing time on a uniprocessor. Hence, the theoretical speedup $S_A(n)$, obtained by using n processors, can be calculated as follows:

$$S_A(n) = \frac{(1 - f_{Ap}) \cdot \tau + f_{Ap} \cdot \tau}{(1 - f_{Ap}) \cdot \tau + \frac{f_{Ap} \cdot \tau}{n}} = \frac{1}{(1 - f_{Ap}) + \frac{f_{Ap}}{n}} \tag{2.6}$$

If we can use an unlimited number of parallel processors, the theoretical speedup $S_A(n)$ approaches its limit as the number of parallel processors n approaches to infinity, and we write:

$$\lim_{n \to \infty} \frac{1}{(1 - f_{Ap}) + \frac{f_{Ap}}{n}} = \frac{1}{1 - f_{Ap}} \tag{2.7}$$

Equation 2.7 indicates that the theoretical speedup is always limited by the proportion of the sequential part in a computation. Amdahl's Law might transmit a pessimistic view of parallel computing to us. For instance, if the sequential part accounts for 10% of a given task, the maximal speedup we can gain is 10 times, even if we can use an arbitrary number of processors for parallelism. Apparently, a speedup of 10 times is likely to be unsatisfactory. Furthermore, communications will cause further degradation of performance in reality. The prerequisite of Amdahl's Law considers only that the problem size (or the workload) is fixed for both sequential processing and parallel processing. Hence, we also call Amdahl's Law the *fixed-size speedup* model [194].

Nevertheless, if we switch from the fixed problem size to the scalable problem size perspective, the speedup is no longer limited by the proportion of the sequential part. *Gustafson's Law* [84], which is named after the American computer scientist and businessman John Leroy Gustafson, states that parallelism increases in an application when the problem size increases. Gustafson's Law calculates the speedup by workload, instead of execution time used by Amdahl's Law. Specifically, the speedup $S_G(n)$ can be defined as the scaled workload solved in parallel divided by the workload solved sequentially, and the workloads completed by both sequential processing and parallel processing are measured in the same amount of time. Thus, $S_G(n)$ is given now by:

$$S_G(n) = \frac{W_p(n)}{W_s(1)} \tag{2.8}$$

where $W_s(1)$ is the original workload and $W_p(n)$ is the scaled workload running on n parallel processors. Again, supposing a task is composed of a scaled parallelizable fraction f_{Gp} and a serial fraction $(1 - f_{Gp})$. Besides, the scalable workload is only in the parallel processing part. Thus, the workload completed by n processors in parallel is:

$$W_p(n) = (1 - f_{Gp}) \cdot W_s(1) + f_{Gp} \cdot W_s(1) \cdot n \tag{2.9}$$

Therefore,

$$S_G(n) = \frac{W_p(n)}{W_s(1)} = \frac{(1 - f_{Gp}) \cdot W_s(1) + f_{Gp} \cdot W_s(1) \cdot n}{W_s(1)} = 1 - f_{Gp} + f_{Gp} \cdot n \tag{2.10}$$

Gustafson's Law is based on two prerequisites: First, the parallel workload should scale up (i.e., the scalable computing). Second, both sequential processing and parallel

processing with n processors work on the same computational task in the same amount of time. That is to say, "the problem size scale-up is bounded by the execution time [194]." Hence, Gustafson's Law can also be called the *fixed-time speedup* model. As implied by Equation (2.10), the fixed-time speedup is a linear function of the number of parallel processors n if the workload is scaled up in order to maintain a fixed execution time.

The speedups obtained by the two laws seem contradictory. However, the definitions of the two serial or parallel fractions in the two laws are fundamentally different. As Shi pointed out in [189], the parallel processing community had misunderstood *the two serial percentages* in nearly three decades, where the two serial percentages are $(1 - f_{Ap})$ in Equation (2.4) and $(1 - f_{Gp})$ in Equation (2.9).

The misunderstanding about f_{Ap} and f_{Gp} is due to the neglect of the prerequisites of the two laws. We now show that f_{Ap} and f_{Gp} are not identical, but related by an equation. Since the problem size is fixed in Amdahl's Law, f_{Ap} can be written as:

$$f_{Ap} = \frac{w_{Ap}}{w_{Ap} + w_s} \tag{2.11}$$

where w_{Ap} and w_s are non-scaled parallel and serial workload, respectively. To obtain f_{Gp} for Gustafson's Law, note that the speedup $S_G(n)$ can also be expressed as:

$$S_G(n) = \frac{w_{Gp} \cdot n + w_s}{w_{Gp} + w_s} \tag{2.12}$$

where w_{Gp} and w_s are scaled parallel and serial workload, respectively; and n is the number of parallel processors. Taking the right-hand sides of Equations (2.10) and (2.12) gives:

$$1 - f_{Gp} + f_{Gp} \cdot n = \frac{w_{Gp} \cdot n + w_s}{w_{Gp} + w_s} \tag{2.13}$$

We then solve Equation (2.13) for f_{Gp} to obtain:

$$f_{Gp} = \frac{w_{Gp}}{w_{Gp} + w_s} \tag{2.14}$$

The relationship between the non-scaled parallel workload w_{Ap} of Equation (2.11) and scaled parallel workload w_{Gp} of Equation (2.14) is:

$$w_{Ap} = n \cdot w_{Gp} \tag{2.15}$$

Equation 2.15 is the root of confusion in which w_{Ap} is the parallel workload of the entire problem, and w_{Gp} is the parallel workload for one processor. When we solve the

system of equations consisting of Equations (2.11), (2.14), and (2.15), we can express f_{Ap} in terms of f_{Gp} by:

$$f_{Ap} = \frac{f_{Gp} \cdot n}{1 + f_{Gp} \cdot (n-1)} \tag{2.16}$$

If we convert f_{Gp} into f_{Ap} (or vice versa) before calculating speedup, the two laws should give the same answer. For example, for a task with $f_{Gp} = 0.9$ when using four processors, Gustafson's Law predicts the speedup $= 3.7$. In order to use Amdahl's Law correctly, we translate $f_{Gp} = 0.9$ into $f_{Ap} = 0.972973$ using Equation (2.16). Finally, the speedup calculated by Amdahl's Law is also 3.7.

In conclusion, our derivation has shown that both Amdahl's and Gustafson's laws are correct ways to predict speedup under their own respective prerequisites.

2.2.2 The Meaning of Gustafson's Law for Parallel Constraint Solving

Due to the combinatorial nature of CSPs, the size of the search space often grows exponentially (e.g., the N-Queens problem). Hence, sequential solving cannot scale efficiently with problem size, and it is often impossible to explore the whole search tree of a CSP in reasonable execution time. For a parallelizable hard CSP, it would be no problem to provide sufficient scaled up workload in a fixed time for a large-scale parallel system. Therefore, as Gustafson's Law implies, it is beneficial to solve CSPs by a large-scale parallel system as the speedup can grow linearly with the problem size. Amdahl's law, on the other hand, only apply if we do not want to solve more difficult CSPs (i.e., increase the problem size) when given more computing power. Thus, since we are always attempting to solve larger and harder instances, we believe that the scalable computing concept of parallel processing should be applied to parallel constraint solving.

The theoretical speedup in the fixed-size speedup model and fixed-time speedup model are bounded by the reciprocal of serial percentage and the number of parallel processors, respectively. In both cases, the speedup gained by parallelism must be less than the number of parallel processors. Nevertheless, we sometimes observe that a speedup $S(n)$ is greater than the number of parallel processors n, or $S(n) > n$, which is called *superlinear speedup*. In [175, 189], the authors use the structural characteristics of the serial algorithm to evaluate a parallel performance. A sequential algorithm can be classified as either structure persistent (SP) or non-structure persistent (NSP). In both cases, superlinear speedup could be expected. The former means that the number of instructions executed by the sequential processing is greater than

or equal to all its parallel implementations, for all inputs. The latter requires that at least one of its parallel implementation, for at least one input, needs the less total number of instructions than the sequential processing. One typical explanation for superlinear speedup achieved by an SP algorithm is that having more cache memory in parallel execution reduces the number of clocks per instruction for memory access. For obtaining a first solution, search algorithms, which terminate the search when one of the processors finds a solution, and then all the other processors stop the execution immediately, are the typical SP algorithm. In this case, the number of executed instructions of parallel processing might be much less than the sequential processing. And thus superlinear speedup appears. In subsequent chapters, we will show parallel constraint solving can often achieve superlinear speedups.

In computer science, parallel computing is a relatively mature discipline that has been developing for several decades. We are unable to cover all the important concepts of parallel computing in a chapter. For further information, the interested reader can consult the textbooks such as [93, 82, 60, 199].

Chapter 3

The Literature Review on Parallel Constraint Solving

As we exit the era of Moore's Law, exploiting parallel processing might be one of the few effective methods that can still allow us to benefit from hardware improvements for solving larger and harder CSPs. As discussed in Chapter 2, techniques for processing constraints can roughly be classified into two main categories: (1) inference and (2) search [40]. Hence, it is natural to consider parallelizing constraint propagation and the search process. The constraints community has indeed extensively researched the parallelization of these two techniques, including parallel constraint propagation and consistency, and parallelizing the search process. Besides, another category of parallel constraint solving, parallel portfolio, takes advantage of parallel processors by running a group of diverse solvers to attack the same problem simultaneously until one of them obtains a solution.

In this chapter, we concentrate on approaches to parallel processing that belong to abovementioned three categories proposed by the constraints community only. Two great surveys about parallel constraint solving [61, 169] have previously appeared, and thus most of the studies to be surveyed in this chapter are inevitably overlapped with them. Here, we divide the work on parallel constraint solving into four general categories: parallel constraint propagation, parallelizing the search process, portfolios, and hybrid approaches. The literature is presented in chronological order of publishing time to help the reader grasp the evolution of parallel constraint solving. Besides, we omit the techniques regarding distributed constraint programming, and parallel SAT solving. The interested readers are directed to [50, 210] for a detailed introduction and survey of distributed constraint programming, and the survey of parallel solving in SAT [132].

3.1 Parallel Constraint Propagation

In Section 2.1.1, we have seen AC-3, which might be the most natural technique for tightening a constraint network. However, we can improve AC-3 by storing information to avoid rechecking the same constraint during the propagation of deletions, which is called AC-4 and proposed by Mohr & Henderson in 1986 [142]. In 1987, Samal & Henderson [183] gave parallel versions of AC-1, AC-2, AC-3, and AC-4 algorithms to achieve arc consistency for shared memory parallel computers. The idea of their parallel arc consistency algorithms is to parallelize the *for* loop using as many processors as iterations in the *for* loop of the sequential counterparts. They claim that, for enforcing arc consistency, any parallel algorithm must have $\mathcal{O}(nd)$ in the worst case, where n is the number of variables and d is the largest domain size. Nevertheless, the proof for this claim is based on an unlimited number of processors and no communication overheads, which is unrealistic. They experienced linear speedups on their undescribed benchmarks with maximal 50 variables by using a BBN Butterfly computer.[1]

Kasif [105] in 1990 pointed out that arc consistency is inherently a sequential process, and parallel arc consistency is unlikely to improve sequential arc consistency too much. One can only expect a parallel algorithm with time complexity more than $\mathcal{O}(nd)$ by using a polynomial number of processors. Interestingly, Cooper & Swain [35] proposed a massively parallel digital circuit in 1992, called arc consistency (AC) chip, to compute the arc consistency problem. AC-4 can be seen as a uniprocessor simulation of the functionality of the AC chip representation. This work is a pioneering attempt to the "hardware for AI" stream that directly adapts computers architecture to problems solving. After 27 years of this work, we are experiencing the renaissance of hardware acceleration for AI applications again, especially neural networks, machine learning, and computer vision. Also in 1992, Zhang & Mackworth [214, 213] investigated structure-driven parallel constraint processing algorithms. They proposed algorithms to compute arc consistency of the CSPs which can be represented by an acyclic constraint network of bounded width. They also proved that efficient algorithms exist for both sequential and parallel processing in such a network. Then, the algorithms were tested on a network of transputers.[2] For the particular problems, the

[1] The BBN Butterfly, which was built in the 1980s, is a massively parallel computer with up to 512 Motorola processors.

[2] The transputer is a series of RISC-like microprocessors from the 1980s, featuring integrated memory and bidirectional bit serial links, intended for parallel computing [208]. The instruction set supports communication, concurrent processes and process scheduling [214].

speedups gained were close to linear. Nevertheless, the algorithms of this work require an acyclic constraint network of bounded treewidth as input. The time complexity of the algorithm for compiling a constraint network into an acyclic one, that is join-tree clustering algorithm, is $\mathcal{O}(r * k^{w^{*(d)}+1})$, where k is the maximum domain size of the variables of a given CSP and $w^{*(d)}$ is the induced width of the ordered graph for the CSP [40, 41]. Unfortunately, with the increase in problem size, obtaining an acyclic network soon becomes unsolvable.

In 1998 Nguyen & Deville [149] presented a distributed AC-4 (DisAC-4) for distributed memory computers using message passing communication. In the DisAC-4 algorithm, each worker runs the same code but handles its set of variables by maintaining the AC-4 data structures. When a worker detects an inconsistent value, this worker must notify this detection to all the other workers and collect inconsistent values from the other workers. Just like the fixed point of sequential constraint propagation, every worker halts when no more inconsistent values are removed and generated, i.e., the entire system reaches the fixed point. The experimental results show that the speedups gained by DisAC-4 on the 45-Queens problem and reverse 80-Queens problem were 3 times and 5 times, respectively, when using eight processors. The DisAC-4 did not consider whether or not the system balance the workload among the workers. Because even if each worker deals with the same number of variables, the workloads for deletion and propagation are not guaranteed to be evenly distributed.

Unlike the studies mentioned above that restrict a constraint network to be binary, Ruiz-Andio *et al.* [181] (1998) improved the parallel arc consistency to n-ary functional constraints. A constraint is translated into an *indexical* that can be viewed as solving an equation for variables. For instance, the arithmetic constraint $x_1 = x_2 + 4$ over finite integer domains is translated into two indexicals: (1) $I_1 \equiv x_1$ in $min(x_2)+4$ to $max(x_2)+4$, and (2) $I_2 \equiv x_2$ in $min(x_1)-4$ to $max(x_1)-4$. The constraint propagation is performed on indexicals. The parallelization is realized by partitioning a CSP $\mathcal{P}$ into n disjoint subsets of constraints statically, and then map these partitions to n processors. Due to its partitioning method and the complexities of the graph structure of CSPs, the scopes of constraints allocated to the processors are almost certain to be overlapped. Therefore, when each processor executes sequential constraint propagation to its subset of constraints, communication, which is either required by domain deletions shared among processors or the detection of termination, is unavoidable. The way of distributing constraints among processors and the frequency of updating shared variables are critical factors affecting the performance. The former decides the workload balancing, and the latter causes the communication overhead. However, there

exists a trade-off between the workload distribution and communication overheads; that is, the improvement of constraints distribution is decreasing the communication overheads while increasing the unbalanced propagation workloads. The authors stated that achieving better-balanced workload distribution is more important than minimizing the communication costs to the performance gain. For the two benchmarks used in the experiments, the maximum speedups were limited to between 2 to 3 times, even though many more processors were used.

A parallel propagation algorithm for solving numerical problems was introduced by Granvilliers & Hains in 2000 [83], where the variable domains are sets of reals, and lowest upper bound and greatest lower bound are floating-point numbers. An immediate adaptation of AC-3 was parallelized. The experimental results evaluated on a massively parallel system Cray T3E-1200 with 64 processors varies from speedups 1.02 to 5.7 times, depending on the instances of the problems. The parallel efficiency of this parallel propagation algorithm on a 64-processors parallel system is rather low. In 2002 Hamadi [86] utilized the bidirectionality property of constraints to reduce the communication overheads for distributed arc consistency. More precisely, an agent avoids unnecessary message passing for constraint propagation by deducing domain reductions in its acquaintances. The experimental results show that their algorithm DisAC-9 required, in the best case, 6.25% message passing of DisAC-6 when solving randomly generated problems on an IBM sp^2 supercomputer with 15 agents (processors). However, the maximum speedup that can be gained by using 15 agents was only 3.64 times. More importantly, negative speedups were observed for all the problems when using 90 agents, which indicates that scalability is problematic for DisAC-9.

Campeotto *et al.* [27] (2014) made the first effort to explore the use of graphics processing unit (GPU) in constraint propagation. The parallelism is achieved by parallelizing constraint propagators. A *constraint propagator* implemented in a constraint solver is a software abstraction which by execution performs constraint propagation [185]. In a constraint propagation system of the state of the art constraint solvers, a constraint is typically implemented by a set of propagators. Their approach to GPU-based constraint propagation can be viewed as three different levels of parallelism. The first level of parallelism is that a thread performs domain reductions for each variable. A block of threads is specified to handle a set of propagators of a constraint at the second level. At the third level, the propagators of all the constraints for a CSP are partitioned into two groups. One group is proceeded by CPU and the other by GPU. If the number of propagators exceeds a given threshold on either GPU or CPU,

propagators can be moved between them so as to balance the workloads. Moreover, to reduce the time required by memory transactions between CPU and GPU, the CPU is responsible for handling a large number of efficient propagators while GPU executes few expensive propagators. The algorithm merges the domains states that are produced by GPU and CPU, respectively, at each iteration of the constraint propagation. Hence, this approach is typical heterogeneous computing since two different types of processors are employed. It is worth mentioning that significant performance enhancement arises when involving more complex constraints. When comparing using GPU working together with CPU to only using CPU, modest speedups were obtained, with the highest being approximately 7 times.

So far, all the works we have surveyed are based on the algorithms that achieve arc consistency. There exist stronger consistencies that remove more inconsistent values than does arc consistency. Singleton arc consistent is one such strong consistency. A constraint network is singleton arc consistent if every value in the network is singleton arc consistent. After instantiating any variable, singleton arc consistency imposes generalized arc consistency on the rest of the constraint network so that the rest of the constraint network is arc consistent. The Ph.D. work of Gharbi [65] (2015) introduced a master/workers architecture, where the master is a sequential CSP solver, and the workers establish a partial singleton arc consistency and send the discovered facts to the master to avoid useless search space. The author admits that the empirical study gives inconclusive results since 93% of 876 instances achieve speedup less than 2 times with eight processors.

3.2 Parallelizing the Search Process

In [61], Gent *et al.* give a definition of parallelizing the search process for parallel constraint solving that has the following features: (1) each worker is close to being a standard sequential constraint process, but (2) they are collectively orchestrated to be part of the same overall search process. We discuss the techniques in this category in the following sections, including work-sharing/stealing, parallel local search, parallel discrepancy search, and problem decomposition.

Work-Sharing/Stealing

Perhaps the first research on parallelizing the search process for constraint solving was published by Perron [159] (1999). In this work, a collection of workers run on

different processes and explore the different regions of the same search tree. The parallelism is achieved by distributing open nodes to different workers, where an open node is a search frontier that leads to individual parts of the search tree. When allocating a new open node to a busy worker, the worker must recompute the maximum common path between the new open node and the current node, and then backtrack to the nearest node in the common path. A virtual communication layer manages to maintain task allocation, load balancing, and termination detection. The parallel search approach can be combined with limited discrepancy search (LDS). Unfortunately, no implementation details of the architecture and parallel search algorithm were given. The experiments were conducted on a four processors Pentium Pro 200Mz computer running Linux. The speedups gained by this parallel search algorithm with the depth-first search were positive but less convincing.

Schulte [184] presented the architecture of concurrent search engine to exploit parallelism in 2000. The engine consists of several workers and a single manager, where the manager and a worker can be viewed as autonomous agents that communicate by exchanging messages. A worker explores nodes of the search tree and generates new nodes. Similar to the virtual communication layer in [159], the manager is responsible for *initialization*, *finding*, *termination detection*, *collecting solutions*, and *stopping search*. In the phase of *initialization*, the manager activates workers by sending a message for the root node of the search tree. The *finding* during the resolution process refers to an approach to workload balancing achieved by work sharing. Specifically, a node from a busy worker's work pool is forwarded to an idle worker by the manager. The author implemented the concurrent search engine on networked computers. Once the manager is created on a networked computer, the manager spawns the process that are run as a worker on a different networked computer. Again as in [159], a worker needs to recompute the path from the root to the node when acquiring a new node. The experiments show substantial speedups using six workers, ranging from 3.17 to 5.21 times. Besides, the granularity for all combinations of examples and workers were close to 10%, which means the work-sharing scheme is a coarse granularity and sufficient for workload balancing.

In 2004 Zoeteweij & Arbab [215] presented a parallel constraint solver constructed by autonomous component solvers. A time-out mechanism was introduced to provide implicit load balancing, i.e., one busy solver generates new subproblems and shares the subproblems to an idle solver when the busy solver time-out elapses. Each component solver is required to be able to publish its search frontiers for work-sharing. Unlike the aforementioned two studies [159, 184], the search frontier is maintained explicitly

by copying all the search decisions that construct subproblems, rather than recomputing. The results of solving three instances are given, showing speedups of between 10 to 15 times on 16 processors and parallel efficiency from 0.75 to 0.97.

Both work-sharing and work-stealing are two scheduling paradigms to re-distribute work when performing parallel computing [97]. In work sharing, busy workers actively migrate their pending tasks to idle workers. This strategy is, in principle, relatively simple to implement; and usually needs a global manager for workload distribution. For example, as mentioned above, [159] and [184] employ a centralized manager to mediate work sharing. Also, in [184], the parallel solver implements a global manager that consists of two components: a *Store* asynchronously buffers incoming subproblems; a R_3 synchronously forwards a subproblem to an idle solver. In work stealing, on the other hand, idle workers attempt to steal tasks from busy workers. This paradigm has advantages that a busy worker pays little overhead to enable stealing and better scalability. Besides, distributed task pool is maintained by each worker, instead of a centralized work pool or manager. Michel *et al.* [140] (2007) parallelized constraint solving without changes to the sequential code, using work-stealing mechanism. Every worker involves only two different types of tasks: exploring its search tree and generating subproblems to be stolen. Whenever a worker is starving and requests a subproblem, a busy worker generates subproblems explicitly and then returns to its exploration. A subproblem to be stolen is essentially a search node denoted by a *continuation* and a semantic path, where the continuation resumes the execution of the search node by backtracking to the longest common point and posting all the renaming constraints. Note that a central pool was used in the experiments; thus, a starving worker does not need to decide which worker to steal when two workers or even more are capable of providing subproblems. But the authors claim that a distributed pool can also be used. The parallel implementations with depth-first search and limited discrepancy search (LDS) were tested on the N-Queens, Scene Allocation, Graph Colouring, and Golomb Ruler problem. The speedups were close to linear, varying from 2.21 to 4.76 times when using four processors. The superlinear speedups gained with LDS was attributed to the disruption of the normal search order.

It is difficult to use single work-stealing strategy to handle complex and diverse CSPs. As an example, assuming that a solution is located in the leftmost part of the search tree (the bottom left corner), if the work-stealing strategy is stealing high (near the root), only one worker explores towards to that part of the search tree. Consequently, the total number of visited nodes is increased, leading to low parallel effi-

ciency. In this case, the more parallel processors are used, the lower parallel efficiency we obtain. Another example is when an instance of a CSP is large and hard, the backtrack search tree must be deep, and a high level (near the root) mistake cannot be recovered from for hours. When the solution happens to be in the rightmost part of the search tree, stealing left and low is unlikely to solve the problem. Chu *et al.* [31] in 2009, developed *confidence-based work stealing*, an adaptive work-stealing strategy, to address this issue by estimating the ratio of solution densities between the subtrees at each node, where the ratio is formulated as confidence. By dynamically adjusting the confidence of each node during resolution process, the algorithm can dynamically decide how to allocate the processing power according to confidences for two branches, thereby achieving "near-optimal" stealing pattern. The mathematical model of updating confidence reflects changes to the exploration of a barren subtree in an intuitive way. That is, if one part of a search tree has been thoroughly searched without producing a solution, the updated confidence then guides more processing power to steal from an unexplored subtree that is as different from the previously explored subtrees as possible. The experimental results demonstrate the effectiveness of this approach, especially for satisfaction problems. The instances solved by parallel algorithm vastly outnumbered the instances solved by the sequential version by 21 to 7 on the Knights problems, 82 to 15 on the Perfect-square problem, and 100 to 4 on the N-Queens problems. Nevertheless, the experimental results also indicate the performance of the parallel algorithm (i.e., runtime, solved instances) determined by the initial setting of confidence, which requires the user's domain knowledge. For example, with a poor confidence value, the parallel runs took longer than sequential runs to solve the N-Queens problems. Although the authors proposed a method to estimate the confidence, this method needs to be further evaluated.

Xie & Davenport [209] (2010) proposed a framework of implementing a constraint programming solver on a massively parallel supercomputer in their position paper. The basic idea of their parallelization scheme is basically the same as those studies [159, 184] we mentioned earlier, i.e., every worker explores an independent part of the search tree; a master maintains a centralized work pool and distributes the tasks to the workers. The authors claim that, for massively parallel computing, the master processor can be an obstacle to further improving the performance of parallelism due to a lot of task requests from workers and consequent task dispatching. Thus, they proposed that scalability can be improved by introducing multiple masters. Unfortunately, they have not managed to conduct an experiment that can

prove the effectiveness and scalability of the proposed approach on a supercomputer with more than 1,024 processors.

Kotthoff & Moore [109] (2010) proposed to use additional constraints to partition search space into parts so that different workers explore independent subtrees in parallel. The idea behind their approach is to split the domain of a variable into several intervals via additional constraints. Assume that the domain of variable x_2 is $\{0, 1, 2, 3\}$ and a current partial instantiation (tuple) $\overrightarrow{x} = \langle x_0 = 0, x_1 = 9 \rangle$. Two new models can be generated by imposing the constraints $x_2 < 2$ and $x_2 > 1$ on the original model. The problem of this method is obvious, because both tuple $\langle x_0 = 0, x_1 = 9, x_2 = 0 \rangle$ and $\langle x_0 = 0, x_1 = 9, x_2 = 1 \rangle$ may prove to be nogoods soon in constraint propagation, even if they are consistent tuples now. Thus, it is difficult to partition the search space evenly by simply posting additional constraints. The authors suggest addressing this problem by dynamically spawning the new models for idle workers. The major procedures of the proposed approach are as follows: First, a breakpoint (e.g., a node $x_2 = 2$) is set on the first worker. Second, start solving the problem on the first worker. The first worker halts if it finds a solution or proves that the problem is unsatisfiable. Otherwise, when reaching the breakpoint, the first worker pauses to generate new models by imposing domain splitting constraints on the remaining search space. The first worker is then resumed from where it was paused by adding a set of restart nogoods that help avoid visiting the search space explored before. Third, whenever a worker becomes idle, the busy workers stops and generates split models for the idle one. Besides, the split models are submitted to the job server. To ensure that the search space explored by a worker will not be searched again, the constraint solver must have the ability to output restart nogoods. The proposed parallel approach has not been experimentally tested. Workload balancing, communication overhead, and scalability remain a challenge for this approach. For instance, for a given variable used to partition the search space, if most of the values in its domain are proven to lead a small *empty subtree* in the subsequent search, all the workers whose search space are allocated by these values have to communicate with the job server for new tasks simultaneously.[3] In this case, apart from the communication overheads and contention, these workers would be lying idle until they have obtained new tasks. The proposed approach did not consider this situation that may occur frequently. Recomputation is unnecessary in this approach because of the restart nogoods. But it is unknown whether in practice nogoods used in this approach is better than recomputation mentioned earlier.

[3]In this dissertation, an empty subtree refers to a subtree without a solution.

Machado *et al.* (2013) [129] described a work-stealing based dynamic workload balancing mechanism developed on top of MPI and *pthreads* [128], targeted at massively parallel processing. The basic work-stealing scheme is similar to previous studies, such as for example [140, 31]. The main novelty of their approach is the introduction of the shared and private region for work pool, where the shared region is used for work-stealing, and the private region can only be accessed by the worker itself. When a worker exhausts its work pool, it attempts to steal work from the shared region of the victim, which belongs to the neighborhood of the worker, without the victim's awareness (i.e, local steal). If this attempt fails, a worker uses one-sided communication to seek a victim with a surplus of workload without disturbing the victim (i.e., remote steal).[4] Experimental results on enumerating all the solutions of the N-Queens problem with n=17 exhibited almost linear speedups with a parallel efficiency of 96% using 512 cores. Besides, the number of failed steals increases with more cores used but remained low. The experimental results on a single quadratic assignment problem optimization instance were similar.

In this dissertation, we mainly focus on the finite discrete CSP, the variables of which have finite discrete domains. But real-life problems often pose challenges beyond finite discrete CSPs. In 2014, Ishii *et al.* [103] presented a parallel branch and prune algorithm for numerical constraint problem, which employs search space splitting and dynamic workload sharing. The algorithm consists of two main phases: (1) the preprocess distributes workloads to the workers, and (2) the postprocess balances the workload of every worker during resolution process. At the beginning of the preprocess phase, the whole search space is possessed in a queue. Then, the preprocess divides the queue into two and then dispatches a portion to another worker. At the end of the preprocess phase, the overall distribution routing is formed as a binary tree. During the postprocess phase, each worker shares the workloads within a few predefined neighbor workers. Results are presented on three benchmark problems, showing speedups up to 32.3 times using 40 cores of the multi-core computer and up to 119 times using 256 cores of the computer cluster. It can be observed a significant communication overhead due to a large number of boxes (i.e., a Cartesian product of intervals [15]) being sent between workers for workload balancing when the problem size is too big to be handled by a small number of workers.

[4] One-sided communication is a technique used to simplify programming and reduce the workload caused by communication. As its name suggests, only the sender involves data transfer, whereas the receiver does not [128].

Parallel Local Search

Local search (LS) starts at an *initial position*, which is a complete random instantiation for all the variables of a given problem. In each step, a heuristic guides the search process to move to a position selected from the *local neighborhood* [96]. LS repeats this process until a predefined *termination criterion* is reached. In order to avoid stagnation of the search process, local search algorithms usually introduce some kinds of randomization, including the generation of initial positions as well as many cases during search. The essential difference between LS and systematic backtracking (BT) is the strong commitment to variables assignments of BT, whereas LS can reassign any variable at any point during search when given sufficient randomness [164].

Several different techniques have been proposed in the last two decades to blend the CP and LS for solving combinatorial problems: by exploring the large neighborhood with CP, by defining the search for the best neighbor as a constrained optimization problem, by using global search techniques for exploring the neighborhood defined by a fragment of the current solution [52]. Besides, another local search method for constraint solving, adaptive search (AS) [33], was proposed a decade ago. In AS, a problem is formulated as a CSP; every constraint needs to define an error function that indicates the violations of constraints for each tuple of variable values. In each iteration, a variable causing the highest error is chosen; AS then selects a value in the domain of the variable based on the min-conflict heuristic. To compute the error for a variable, the errors of all the constraints whose scope covers the variable are accumulated. By projecting constraint errors onto the relevant variables, the AS method offers an opportunity to repair the worst variable by changing its current value to the most promising value. Besides, a short-term memory mechanism and a reset mechanism are employed to escape stagnation around local minima.

Caniou *et al.* [28, 29] (2011; 2014) presented a parallel implementation of the AS method and investigated its performance results on computers with several hundred or thousand parallel processors. A multi-walk adaptation of AS is used to parallelize the search process, i.e., running an instance of AS to explore the different parts of search space using different cores simultaneously. Both multi-walks with communication (cooperative multi-walks) and without communication (independent multi-walks) are considered. Independent multi-walks achieved linear speedups up to 8,192 cores for large instances of the Costas Array Problem ($n = 21$, $n = 22$ and $n = 23$). The authors explained that the linear speedups achieved by their approach could be attributed to exponential runtime distribution over the search space. Similar to the EPS

approach that we will discuss on page 43, the introduction of communication was not able to improve the performance of the method.

Munera *et al.* [147] (2013) presented X10 implementations of the AS method.[5] Their experiments identified that the data-parallel implementation is an effective way to exploit parallelism; functional implementations, by contrast, cannot yield any speedup. Experimental results on the Costas Array Problem is similar to [28, 29], showing close to linear speedup. However, for other problems (e.g., the magic square), the speedup tends to flatten out as more cores are used, which confirms the experimental results reported in [28, 29]. In 2014, Munera *et al.* [148] proposed a parametric cooperative local search framework aimed at enhancing the performance of parallel independent multi-walks strategy. In this framework, the available workers are grouped into several *Teams*; communication is required among and inside Teams. Inter-team communication is required to guarantee diversification of the search space while intra-team communication is required to ensure intensification that guides the search to a promising region of the search space. The X10 implementations of the framework were compared to the independent multi-works version, showing that the former beats the latter. Nevertheless, it remains unclear if this cooperative search framework is superior to the cooperative multi-walks presented in [29] since this framework has not been evaluated on hardware with more than 32 cores, and the metrics used for assessing the execution process may not be reliable.

Parallelising the AS method has also been explored on GPU. Arbelaez & Codognet [8] (2014) showed that executing the entire local search process on the GPU achieved up to 17 times w.r.t. a well-tuned sequential run on a CPU. Notably, results on the Magic Square Problem with $n = 400$ showed that using a single GPU can achieve better performance than using a supercomputer with 64 cores (see [28]). Unfortunately, the detailed description of the GPU implementation was not provided in the paper. The authors merely stated that their approach exploits parallelism at two different levels: multi-walk parallelism can be achieved by using different blocks of a GPU while inside a block, large neighborhoods are evaluated simultaneously in a single-walk manner.

Parallel Discrepancy Search

We have already mentioned Limited Discrepancy Search (LDS) twice but without explanation. Limited discrepancy search [92] was developed to offset a small number

[5]X10 is a programming language specialized in parallel computing and developed by IBM.

of "wrong turns" at which search strategy begins to direct the backtracking search towards an empty subtree. The term discrepancy stands for any decision point which violates the heuristic's choice. The effectiveness of LDS is based on the premise that heuristic has hopefully made only a few mistakes, and LDS offers the opportunity to bypass these small number of "wrong turns" at a low price. LDS proceeds in a series of depth-first iterations. On the i-th iteration, LDS explores those root-to-leaf paths with up to i discrepancies. LDS needs to cooperate with a value ordering heuristic that sorts the values of a variable to be branched on from the most likely one to succeed to the least likely one. In other words, from the value ordering heuristic perspective, the leftmost child is the most likely one to lead to a solution, and the rightmost child is the least likely one. Thus, when provided with a high-quality value ordering heuristic, the solution density of the subtrees explored by LDS decreases as the number of discrepancies increases. Therefore, LDS is likely to explore fewer nodes to obtain a first solution compared to the backtracking search.

Moisan *et al.* [144] (2013) described a variation of LDS, parallel discrepancy-based search (PDS), which is the first to show how to parallelize LDS. In PDS, the parallel processors explore the leaf nodes of a search tree in the same order as would the sequential LDS do. During resolution process, every branching decision made by each processor selects the node, which can lead to the leaf node that can only be visited by that processor. Thanks to the round-robin assignation of the processes, the algorithm can achieve a balanced workload distribution. The authors derived the expression for calculating the ratio of the number of nodes visited by PDS with ρ processors over the number of nodes visited by LDS on the same problem with n binary variables, written as $\frac{PDS_\rho(n)}{LDS(n)}$. The expression implies that the speedup increases linearly with the increasing number of variables. The statistical results show that PDS can always achieve performance gain with more processors, and the average computation time to obtain a first solution decreases with the quality of the heuristic improving. The instances of an industrial problem with 65,142 variables and 50,238 constraints were solved by using 512, 1,024, 2,048, and 4,096 processors. For the scalability of PDS, the experimental results of the industrial instances are in line with the statistical results. The speedup was still increasing when using 4,096 processors, where the speedup is calculated by dividing the number of leaf nodes explored by multiple processors by the number of leaf nodes explored by one processor. Both theoretical analysis and empirical results demonstrate that PDS is a successful approach to exploiting parallelism for the CP.

Nevertheless, we believe that the following intrinsic reasons of PDS's parallel mechanism impede its further development and application: (1) PDS demands a deter-

ministic search strategy (variable and value ordering heuristics). Because search space splitting and workload balancing achieved by PDS are guaranteed by a uniform search strategy and a unique processor ID for every processor, non-deterministic variable or value ordering heuristics are bound to break predefined search space splitting as well as workload balancing. (2) The parallel efficiency declines, as either the number of processors or of variables grows. We can observe this trend from the experiments, and we can also deduce this trend from the ratio $\frac{PDS_\rho(n)}{LDS(n)}$, i.e., the ratio grows when increasing either n or ρ, which implies that, for a given problem, more nodes need to be explored repeatedly when using more processors. (3) We see a frequent fluctuation in speedup for low probability values, where the probability is the chance of obtaining a solution in the left subtree when the current partial assignment is consistent. It is challenging to ensure the heuristic with high probability for CSPs in practice, especially maintaining high probability for every branching decision.

LDS does not distinguish all discrepancies to their depth. However, a value ordering heuristic is more likely to select higher quality values at lower levels of a search tree since more information is accumulated. In other words, "wrong turns" are more likely to occur near the root of the search tree. Depth-bounded Discrepancy Search (DDS) [203] iteratively increases the depth bound to restrict discrepancies high in a search tree strictly. More specifically, given a search tree with depth n, at each iteration k, $1 \leq k \leq n$, DDS respects the value ordering heuristic below level k and explores all value assignments that violate the value ordering heuristic at level k. Besides, all the branches in the search tree above level $k-1$ are visited at iteration k. Both DDS and LDS explore the same number of nodes [145] when searching a complete tree, and the leaves at the bottom of the search tree are visited precisely once.

Moisan *et al.* [145] (2014) proposed a parallelization of DDS, which is called parallel Depth-bounded Discrepancy search (PDDS). The theoretical analysis of PDDS indicates that the number of nodes visited by PDDS is less than PDS when searching for a complete binary tree. PDDS can outperform PDS since exploring discrepancies near the root of the search tree can lead to a solution faster. The experimental result shows that even the sequential DDS can compete with PDS running on 512 workers. The authors compared the performance of PDDS with PDS on solving datasets of industrial problem. PDDS spent much less computation time than PDS to obtain a solution of a given quality when solving the optimization problem. In [145], the speedups of PDDS were calculated as the ratio of the number of leaf nodes explored by multiple workers divided by the number of leaf nodes explored by one worker. Both theoretical analysis and empirical results prove that speedup increases linearly, with

the increasing number of processors. We have pointed out the three problems of PDS. PDDS mitigates the second problem of PDS since fewer nodes are explored repeatedly by PDDS, compared to PDS. However, we believe that PDDS still need to face the rest of two problems that exist in PDS since a deterministic search strategy and good value heuristic are still demanded by PDDS.

Problem Decomposition

The term *problem decomposition*, as a separate category of parallel constraint solving approaches, first appeared in the book chapter [169] written by Régin & Malapert. But we still regard problem decomposition as a subcategory of *parallelizing the search process* since the entire search tree is ultimately partitioned into a set of subtrees even if this is accomplished by decomposing the original problem. A parallel constraint solving approach can be classified as problem decomposition if the following two properties hold: (1) No communication or almost negligible communication between the workers is required. (2) The original problem is decomposed into a set of disjoint subproblems.

From 2013 to 2016, a series of papers by Régin and co-authors [170, 171, 173, 131] have introduced the concept of embarrassingly parallel computation for parallel constraint solving, which is called embarrassingly parallel search (EPS). The EPS approach has several advantages, including good scalability, workload balancing, and simple implementation, which allow it to be competing with the work-stealing approach and solve hard CSPs.

The problem decomposition of EPS is realized by choosing a subset of variables and generating the assignments to these variables that are not inconsistent with any constraint of a CSP $\mathcal{P}$. Each partial assignment represents a subproblem of original problem. A master is responsible for producing subproblems and adding them to a queue. Then, an idle worker fetches a subproblem from the queue and solves it. Every worker repeats this process; the parallel resolution terminates until all the subproblems in the queue have been solved. For some CSPs, it is hard to ensure that each subproblem lead to a fixed size of search space. For instance, one subproblem may soon be proved to lead to a fruitless subtree while another subproblem requires a much longer execution time to terminate. Thus, to balance the workload among workers, each worker is assigned to many subproblems. Consequently, the active times of all the workers are close to each other. Besides, since all the subproblems are mutually independent, and no communication is required during the resolution, embarrassingly parallel computation is automatically achieved. Note that for an optimization

problem, sharing the incumbent objective value between workers might improve their current resolution but also might lead to a decrease in performance due to communication overhead. According to [169], the authors did not observe any adverse impact of no communication when using EPS to solve optimization problems.

Two methods are proposed to generate subproblems: a top-down method and a bottom-up method. The top-down method utilizes Depth-Bounded Depth First Search (DBDFS) to decompose a CSP $\mathcal{P}$ into q subproblems. DBDFS repeatedly deepens the bound of the depth-first search until $|A_Y| \geq q$ where $|A_Y|$ stands for the number of subproblems generated by a subset of variables Y. Unlike the iterative deepening depth-first search of the top-down method, the bottom-up method tries to identify the depth d in a depth-first manner at which q subproblems can be generated. More specifically, the procedure of the bottom-up method finds the highest (near the root) search frontier with roughly q open nodes (i.e., q subproblems) via sampling and estimation. The procedure consists of three phases. First, the procedure builds a partial search tree by sampling. Second, the level of widths of the real tree is estimated via the partial search tree. Third, the depth d that can guarantee q subproblems is decided with a greedy heuristic.

The EPS approach has been verified on three representative parallel computing platforms: multi-core, data center, and cloud computing. Besides, the authors also implemented the EPS approach on three constraint solvers, including Choco2 2.1.5 written in Java, Gecode 4.2.1, and OR-tools rev. 3163 written in C++.[6] Constraint satisfaction and optimization problems were considered, and the problems of obtaining a first solution were ignored. The implementations of EPS were also compared with the work-stealing approach in Gecode. On the multi-core computer, the EPS approach showed close to linear speedups for most of the instances, whereas it never happened for the work-sealing approach. When solving a same problem, Choco2 is slower and less efficient than Gecode and OR-tools. The similar phenomenon of speedups was also observed on data center and cloud computing platforms.

Another paradigm called SelfSplit proposed by Fischetti *et al.* [51] in 2014 can also be categorized as problem decomposition. The most apparent feature of SelfSplit is that each worker autonomously decides on its task to process without communication. The process of SelfSplit can be roughly divided into three phases: (1) every worker builds the exactly same enumeration tree during the sampling phase; the sampling phase terminates when sufficient open nodes have been generated, (2) each

[6]At the time of writing the latest versions of these solvers are Choco solver 4.10.1 [165], Gecode, 6.2.0 [186], and OR-tools 7.0 [195].

worker applies a deterministic rule to identify and explore the sub-search space that belongs to it, and (3) a special worker is in charge of merging the output of the other workers after finishing its task. For hard computational problems, the sampling phase requires only a very small proportion of the overall computation. SelfSplit is shown to achieve linear speedups up to 16 processors and acceptable sublinear speedups up to 64 processors on the selected instances of the problems. Currently, SelfSplit does not require communication. It is unclear whether or not introducing a limited amount of communication between workers can improve the performance of solving optimization problems.

Menouer *et al.* [138] (2016) showed that mixing problem decomposition and dynamic work-sharing could yield better results than either technique on its own. The mixed parallel approach has two steps: First, the static decomposition generates subtrees to be searched by the parallel processors just like the EPS approach. Then, the second step employs dynamic partitioning to overcome load imbalance between computing cores. The dynamic partitioning is essentially a work-sharing technique that a busy worker shares the work via a global priority queue whenever idle worker(s) exist(s). Although this mixed parallel approach is targeted at running on shared and distributed memory architectures, experiments were performed using two computers with only 12 cores.

3.3 Portfolios

Parallel portfolio for constraint solving exploit and guard against the variability of performance, which can be observed between different constraint solvers, different algorithms (i.e., search strategies for CP), or parameter tuning (e.g., parameter settings of restart strategy). The rationale behind the portfolio search is to execute several assets of the same problem simultaneously and independently, where each asset typically uses a distinct search strategy, restart strategy, even model and solver. By doing so, the likelihood of finding a first solution is increased, or the resolution time of finding a first solution is shortened. Thus, the search robustness is enhanced. The idea of portfolio can also be applied to sequential constraint solving. For example, Gecode [186] supports sequential portfolios that consist of several different assets being executed in a simple round-robin scheme.

The parallel portfolio usually does not require communication between workers and can inherently achieve a balanced workload. Moreover, it can be implemented

into constraint solvers with little effort. Although this approach can improve the performance of constraint solving, it might result in redundancy between workers. Another challenge for a parallel portfolio solver is to devise a scalable source of diverse assets. The more detailed discussion of the limitations of the current parallel portfolio search will be presented in Section 6.3.

In CP, we often restart the resolution process when reaching fail-limit to overcome heavy-tailed distributions [73]. Cire *et al.* [32] (2014) developed a simple technique for parallelizing the Luby restart strategy [127] that requires no communication. To parallelize the Luby restart strategy, each worker has its own local copy of a scheduling class that assigns restarts and their respective fail-limits. The scheduling class of each worker computes and infers the next Luby restart fail-limit. In this approach, all the parallel processors execute the Luby restart strategy, as a whole, in parallel. The latest next Luby restart fail-limit computed by the scheduling class is always assigned to the worker with the lowest number of accumulated fails so far. Randomization is introduced along with every restart; thus, each worker almost works on independent search space. Theoretical analysis indicated this approach achieves linear speedups. However, experiment results on hard instances (e.g., the Costas Array n=20) showed sublinear speedups.

There exists one type of parallel portfolio that consists of several different constraint solvers. The basic philosophy underlying this approach is "using or adapting solutions to old problems" [152, 174]. More specifically, this type of parallel portfolio approach selects the most promising solvers from a candidate pool based on static features or by learning the dynamic behavior of solvers [169]. Amadini *et al.* [5] (2015) introduced a parallel CP portfolio solver, *sunny-cp2*, incorporating 12 different constraint solvers. Given a CSP $\mathcal{P}$, the first step (pre-solving phase) involves running a static schedule of solvers and detecting the instances similar to the CSP $\mathcal{P}$, called neighborhood. *Sunny-cp2* maintains a knowledge base that includes feature vector of 5,527 CSPs and the runtime information of each solver of the portfolio on these instances. In the pre-solving phase, the neighborhood is computed by using one core through two steps: (1) the numerical attributes (e.g., number of constraints, of variables, etc.), which characterizes the CSP $\mathcal{P}$, are extracted, and (2) the extracted numerical attributes is then used for selecting a set of the k nearest neighbours of the CSP $\mathcal{P}$ within that knowledge base. With the neighborhood of $\mathcal{P}$, *sunny-cp2* can dynamically compute the sequential schedule, and the solving phase runs the dynamic schedule on different cores to solve the problem. The performance of *sunny-cp2* was validated on heterogeneous and large benchmarks, showing that the technique

outperforms all its constituent solvers, its earlier version *sunny-cp* [4], and *ppfolio* [179]. It should be noted that this type of parallel portfolio approach cannot enhance the problem-solving ability of an individual constraint solver. The metric for the assessment of the performance on solving CSPs is typically the number of CSPs solved by at least one constituent solver within a time limit.

Recently, Archibald *et al.* [10] (2019) parallelize constraint solving by having each worker perform a search employing the slightly-random value ordering heuristic with a different random seed and sharing nogoods on restarts. The parallel approach was evaluated on two different parallel platforms: shared memory architecture and distributed memory architecture. The gain from using 36 threads on shared memory computer are modest, giving an *aggregate speedup* [94] of 12.7 times. When using distributed, the aggregate speedups are also reported, with speedups ranging from 57.1 times for five hosts and 95.5 times for ten hosts.

All of the parallel techniques we discuss in this dissertation aim at enhancing constraint solving performance directly. The parallel portfolio approach can also be used to automatically build an efficient CSP model when given an abstract problem specification; see for example [2, 151]. Most of the researches on the parallel portfolio approach are essentially concentrated in SAT solving, and they are beyond the scope of this dissertation. Thus, please refer to survey article [61] for further information.

3.4 Hybrid Approaches

In the previous sections, we have surveyed the work on the three main categories of parallel constraint programming, including parallel constraint propagation, parallelizing the search process, and portfolios. However, some studies in parallel constraint solving cannot be classified under the three categories since they employ multiple approaches.

In 2011, Rolf, in his Ph.D. work [176, 177], described an approach that parallelizes both consistency and search. He introduced two terminologies: parallel consistency means a collection of constraints of a CSP $\mathcal{P}$ is partitioned into subsets then each subset of constraints is enforced consistency in parallel; parallel search means a domain of a variable is divided into subsets and independent search space is explored in parallel. The parallel approach exploits thread-level parallelism. Parallel search is performed by a set of search threads, each of which is assisted by several consistency threads. A search thread is required to switch to a consistency thread after making an assignment and before processing to the next level of the search tree. Every

consistency thread retrieves a set of constraints from a constraint queue to perform propagation. When all the constraints have been processed, the algorithm commits all prunings and resumes the search. The idea to combine parallel consistency and parallel search is based on the judgment that many CSPs spend a magnitude more time on constraint propagation than search. However, no quantitative analysis or empirical result about this judgment was provided. There are at least several factors that have an impact on the execution time of propagation, such as constraint programming model, consistency level, filter algorithms used by the constraints in the model. The author identified the three bottlenecks that can degrade performance: (1) since independent search spaces do not exchange data during resolution process, the algorithm repeatedly evaluates constraints, (2) synchronization is performed until the slowest consistency thread completes its work, and (3) parallel search naturally puts too much stress on the memory bus [61].

In 2012, Yun & Epstein [211] proposed a hybrid approach (called SPREAD) combining portfolio-based methods and search space splitting. SPREAD is composed of two phases: a time-limited portfolio search phase followed by a search space splitting phase. In the portfolio search phase, all workers execute the same constraint programming model but with distinct random seeds. The purpose of portfolio search phase is to collect *weights* and *base* to support search space splitting phase, where weights are the average of the variable weights (i.e., the number of domain wipeouts by the constraints whose scope contains this variable) obtained from all the workers, and the base is the average of the number of backtracks obtained from all the workers as well. However, if any worker finds a solution or proves that the problem is unsatisfiable, the execution terminates. Otherwise, the splitting phase starts to partition the search space by using the variables with the highest weights sorted by the weights learned during the portfolio phase. If a worker cannot solve the subproblem within a fixed backtrack count limit, SPREAD recursively separates the returned subproblems via new splitting variables. The authors divided the 500 instances into three solving difficulty levels, including the hard problem set, the harder problem set, and the challenge set. Then, they compared SPREAD with the other parallelization methods using a Cray XE6M supercomputer with 160 nodes, and each node has two octuple-core processors. For the most challenging problem set with 133 instances, given a 30 minutes timeout, SPREAD solved 26 instances, which had not been addressed by the other methods. For the 248 instances classified as the harder problem set, given 30 minutes per problem, SPREAD solved around 60 instances compared with about 40 instances solved by the other parallel approaches. It is worth

mentioning that the learned knowledge from the portfolio phase improved the performance since SPREAD could solve 24.37% more instances of the hard problem set than NWSPREAD (i.e., SPREAD without the learned knowledge). The experimental results show that SPREAD, a complete parallel constraint solving method, offers an opportunity to solve unsolvable instances. We believe that SPREAD still has potential for enhancements. The authors emphasize that intensification is more critical than diversification when solving hard CSPs. Therefore, a single search strategy with small variations was used at the portfolio search phase. As a direct consequence, the variable selection of the search space splitting phase is prone to be misinformed by information gathered during the portfolio phase, which is also pointed out by the authors. Here we would like to suggest introducing diversification, i.e., multiple search strategies during the portfolio search phase, or various criteria for sorting variables during the splitting phase.

Ehlers & Stuckey [48] (2016) investigated combining portfolio search, search space splitting, and probing the objective value for optimization problems. Objective probing is to estimate bounds on the objective, and use some solver processes to probe whether or not a solution satisfying the estimated bound can be searched. By using tighter objective bounds, more search space containing solutions with worse objective bounds can be pruned as soon as possible so that redundant search is avoided. A lazy clause generation (LCG) solver, which employs SAT-style learning with constraint propagation, is parallelized. Learned clauses and incumbent objective value can be exchanged between workers during parallel resolution. Similar to EPS, a subset of variables is selected to split the search space. The authors claim that the variables, which do not occur in *the minimum unsatisfiable core*, are less likely to accelerate the parallel processing. Hence, they used the Variable State Independent Decaying Sum (VSIDS) activities to select variables for the search space splitting. Unfortunately, no detailed information about the *minimum unsatisfiable core* was provided, and the reason why the variables selected by VSIDS can lead to better speedup was also ignored. The hybrid solver consists of three groups having the same number of workers. The first group uses the search space splitting. Workers from the second group perform portfolio search after estimating the respective estimated objective values. The remaining workers behave like the second group, but re-estimate bounds on the objective when reaching the time limit. Modest speedups are reported when using 64 cores, with the average being 16 times on all instances and 140 times on the difficult ones.

3.5 Conclusion

We have comprehensively surveyed the literature on parallel constraint solving. The constraints community has extensively studied this area for more than two decades. Early researches on parallel constraint solving are concentrated mainly in parallelizing constraint propagation. But unfortunately, parallelizing propagation suffers from low parallel efficiency and limited scalability since synchronization cannot be avoided. The parallel approaches related to parallelizing the search process have become the research hotspot for parallel constraint solving since around the 2000s. The state of the art constraint solvers have already provided the work-stealing architecture for parallelizing the search process as well as the parallel portfolio search to enhance search. We can also see the potential of combing the problem decomposition approach or the parallel local search with massively parallel processing to accelerate constraint solving.

Parallel computing is now the norm. However, parallel constraint solving is far from a mature science yet, and its implementations can differ from one solver to another. We can expect that the techniques for parallel constraint solving will most likely evolve in the future. We also hope to see more results of parallel constraint solving in the coming years, and this survey will provide a panoramic review to future researchers wishing to improve the current situation.

Chapter 4

The Effectiveness of Parallel Constraint Solving

We often experience the situation that the resolution process ends up with no solution after a very long execution time when solving computationally hard CSP instances (e.g., some combinatorial problems formulated as CSPs). For these CSPs, the search space is typically too large to explore exhaustively by current search-based techniques and hardware. Unfortunately, we are unlikely to address this issue fundamentally, unless some disruptive technologies such as quantum computing can replace the current classical computing.

Nevertheless, one could argue that he/she can rely on better heuristics to accelerate the resolution process of finding a solution if he/she does not wish to obtain all solutions to a CSP. Indeed, if the problem is not difficult and not large, a suitable heuristic can guide the search to the fraction of the search space with solutions. If the problem is, however, difficult and large, backtracking search puts too much pressure on the heuristics early in the search. In other words, the success of the backtracking search depends on the success of the branching decisions made by the heuristic early on in the search.

In this chapter, we investigate how to overcome the limitations of the backtracking search for enhancing its performance. In Section 4.1, we discuss how early mistakes affect backtracking search. We then focus on techniques that can compensate for bad branching decisions made by the heuristic early on in the search, in Section 4.2. Specifically, we endeavor to quantify the properties of problem decomposition and parallel stochastic portfolio search, which leads to a detailed analysis. Finally, Section 4.3 concludes this chapter.

4.1 Early Mistakes

In the context of backtracking search, a *bad* node is a node such that the subtree below it does not lead to any solution, and we will encounter deadend(s) in the subtree. If a node is not a bad node, it is a good node. Furthermore, if a child of a *good* node is a bad node, this bad node is called a *mistake*. More specifically, having branched a mistake node, the search strategy (i.e., variable and value ordering heuristic) used for guiding search begins to direct backtracking search towards a subtree without a solution (i.e., empty subtree).[1] We can also define a mistake using the notion of a *backdoor*. A backdoor consists of decision variables, the consistent instantiations of which can lead to a solvable subproblem (i.e., solved in polynomial time) [70, 71, 201]. Then, a mistake can be defined as a selection of a variable not included in a minimal backdoor.

Consider a possible backtrack search tree shown in Figure 4.1. The bad and good nodes are drawn with black text on a white background and white text on a black background, respectively. The triangles stand for large subtrees, of which the hollow triangles with a dashed border stand for empty subtrees, and the black triangle stands for a subtree with a solution or solutions. By convention, we assume that the backtracking search iterates from left to right. In this example, the first commitment at node B is a fatal mistake if the subtree below to it is empty and too large to be visited exhaustively. Once the search strategy employed in the backtracking search selects node B and commits to that successor's subtree, any subsequent decision is in vain. As a direct consequence of mistake B, the constraint solving is bound to end up falling into the trap of endless futile backtrack search.[2]

A mistake made by the search strategy implies that constraint propagation cannot remove the value of the mistake when making the branching decision. If a mistake has occurred at the early stage of the backtracking search (i.e., near the root), we call it an **early mistake**. A mistake made near the leaf nodes of a search tree is usually not fatal because the mistake can recover from backtracking quickly. However, an early mistake can be extremely harmful to constraint solving. Specifically, if the empty subtree below an early mistake is vast, the backtracking search will commit to

[1] In chapter 3 (page 36), we have stated that an empty subtree denotes the subtree that cannot lead to a solution.

[2] If there is no limitation on the execution time required for the resolution process, the resolution process of solving any CSP will eventually halt. The execution time can be longer than one hour, day, week, month, year, decade, century, or even at the time of our solar system's death. The setting of the time limit depends on the amount of time one is willing to wait. Thus, strictly speaking, the word "endless" is improper. Please note that, in this dissertation, we do not consider dynamic CSPs that are prone to change over time; thus, the CSP is determined and fixed before search.

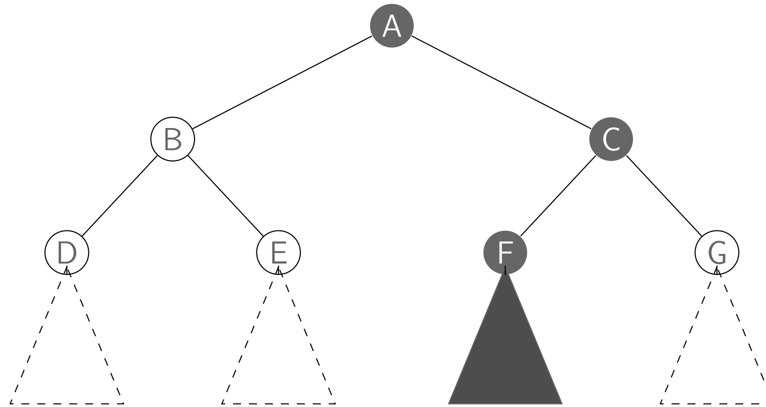


Figure 4.1: Unrecoverable fatal mistake because of the large subtrees below to the mistake B. Nodes B and G are mistakes.

exploring this vast empty subtree in a very long execution time before returning to the branching decision that mattered. Consequently, the search effort is wasted on that empty subtree, and the problem cannot be solved in the amount of time permitted.

The concept of early mistakes dates back at least to the Ph.D. work of Harvey (1995) that first introduces the techniques of randomization and restarts within a backtracking search and limited discrepancy search [91, 92]. Besides, the early mistake has also been identified in the research areas of the propositional satisfiability problem (SAT). For example, Crawford & Baker [37] pointed out that the early mistake causes poor performance in solving a scheduling problem using the DPLL algorithm that is based on backtracking search. Similarly, the same phenomenon is also observed in the context of the branch and bound algorithm [133].

To improve the efficiency of the backtracking search, one can employ a search strategy to guide the backtracking search toward the subtrees that are more likely to contain solutions, if he/she does not wish to find all solutions to a CSP. Ideally, an optimal search strategy can always select the best candidate variable and assign the best candidate value to the variable without encountering any deadends (cf., for example, Figure 4.2). Consider that the probability of success in finding a solution can be calculated as $\prod_{i=1}^{d} p_i$, where p_i is the probability of selecting a good node at level i, and d is the depth of the backtrack search tree. An optimal search strategy can be viewed as, for every branching decision, the probability of selecting the best candidate variable and value is a certain event, i.e., $p_i = 1$.

Nevertheless, even determining the first variable for an optimal search strategy is a nontrivial task [119]. Some effective variable/value ordering heuristics and abstract

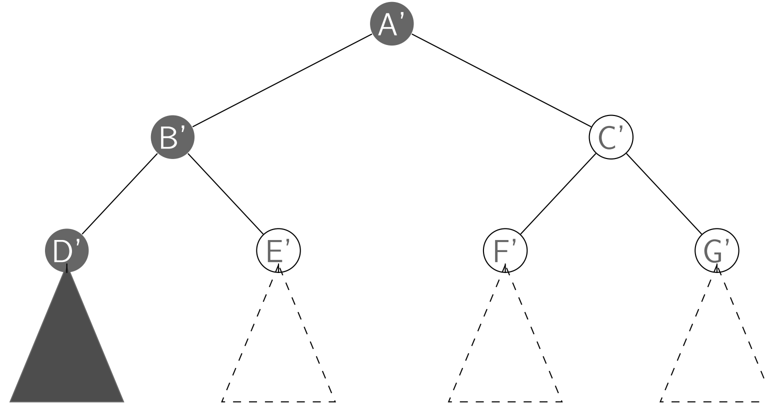


Figure 4.2: An ideal backtrack search tree formed by an optimal search strategy for a first solution of the same CSP shown in Figure 4.1.

search strategy, which essentially aims at increasing p_i, have been proposed and implemented in most of the state of the art constraint solvers. But, in reality, a low-level branching decision (near the leaf nodes of search tree) is probably more reliable than a high-level branching decision (near the root), i.e., $p_i > p_j$, for $i > j$. The reason is that more information is gathered, and the problem size is reduced due to constraint propagation and the partial instantiation of variables. It is not uncommon to see that a large and difficult CSP instance cannot be solved in very long execution time (e.g., a week), even if we try all the existing search strategies. This situation implies that the probabilities of finding good nodes early in the search are rather low, and early mistakes have occurred.

4.2 Possible Approaches to Tackle Early Mistakes

Recognizing that early mistakes made by search strategy are culprits that might render a solvable CSP to an insoluble CSP, we naturally think of how we can eliminate or at least alleviate the effects of early mistakes. In this section, we will investigate the existing non-systematic techniques that address the problem due to early mistakes, including discrepancy search, portfolio search, and restart-based search.

One way of dealing with early mistakes is to backjump immediately to the source of failure that is several decision levels above the current level, instead of chronological backtracking, i.e., backtrack to the preceding variable in the ordering. This look-back technique is called *non-chronological backtracking* [40, 114], which requires that constraint solver learns the reasons for the failures (i.e., nogoods) and store the recorded failures (i.e., nogood recording) [201]. Non-chronological backtracking using

nogood recording can efficiently reduce the search space while still being a systematic search approach [114]. However, we do not intend to discuss the non-chronological backtracking algorithms here since, to the best of our knowledge, it has not been used for parallelizing constraint solving.

4.2.1 Restart-Based Search

When backtracking search trapped in a barren subspace due to a mistake of ordering heuristics, a natural and intuitive way to deal with this issue could perhaps abandon the current search effort and restart with a different search space exploration. The questions then arise: (1) When to abandon the current search? and (2) How should a repetitive search space be avoided when commencing a new search? One method of choice for answering the first question is to use a restart strategy prescribing a sequence of steps for restarts (i.e., cutoffs). Meanwhile, to address the second question, we avoid the search space explored before by introducing randomization to a deterministic backtracking search.

Restart strategies

If a constraint solver is unable to solve a CSP in r_i steps (i.e., cutoffs), then the solver will be restarted and commence the search with a different region of the search space. If the solver cannot find a solution after running for the next r_{i+1} steps, the solver will be restarted again, and so on. We call this infinite sequence of predefined steps $\{r_1, r_2, r_3, \ldots, r_i, r_{i+1}, \ldots\}$ *restart strategy*, where each cutoff r_i could be either infinity or a positive integer.

By adding randomization to the deterministic backtracking search, we can observe that the running time for solving a given CSP varies from one run to another. Thus, we can model this problem-solving behavior observed during runs as a discrete random variable. If we let the random variable T denote the number of backtrack steps required to solve a CSP $\mathcal{P}$ using a randomized backtracking algorithm $\mathcal{A}$, then $Pr(T = t)$ or $f(t)$ denotes the probability that $\mathcal{A}$ terminates using exactly t steps. Moreover, let $F(t)$ be the probability that $\mathcal{A}$ terminates using t or fewer steps, i.e., F is the cumulative distribution function of f. Thus, $1 - F(t)$ is the probability that $\mathcal{A}$ terminates using more than t steps, which is also called the *survival function*.

Luby *et al.* [127] show that the optimal restart strategy is a sequence of fixed cutoffs when the runtime distribution of a randomized algorithm is known (i.e., f is known), given by $R_l = (r, r, r, \ldots)$, where r is a positive integer. They also provide

an optimal restart strategy minimizing the expected runtime for the situation that the runtime distribution is unknown. The optimal restart strategy is called universal strategy and given by $R_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1 \ldots)$. Geometrical restart strategy $R_g = (1, r, r^2, r^3, \ldots)$ [204] restarts constraint solver in a geometric growth manner. Nevertheless, this restart strategy cannot guarantee its worst-case performance, and the expected runtime of the strategy can be arbitrarily longer than the optimal strategy [201]. However, it has the advantage of being less easily influenced by the details of the runtime distribution [70].

Both Luby's and geometrical restart strategies are predefined static restart strategy. By contrast, the dynamic restart strategy [106, 180] employs a Bayesian model to predict the runtime of the backtracking algorithm on the current instance by considering the real-time observations about the attributes of problem instances and solver behavior. The experimental results show that the dynamic restart strategy can outperform its static counterparts. However, the dynamic restart strategies have not become prevalent among the state of the art constraint solvers. First, it is computationally expensive to collect meaningful runtime data of an instance, mainly when its runtime exhibits enormous variations. Second, the predicted restart strategy could still be sub-optimal for the given instance, even though the obtained model captures the overall behavior of the backtracking algorithm on the training set [58].

Carefully examining the runtime distribution of a randomized backtracking algorithm on a CSP $\mathcal{P}$ can help us gain insight into the effectiveness of restarting search and identify the difficulty of the given problem. Gomes *et al.* [73, 72] pointed out that different randomized backtracking algorithms on the same problem can exhibit considerably different behavior between runs, i.e., the variance of runtime distribution is enormous. For example, one run obtains a solution in a few seconds, whereas another run cannot result in a solution even after a few weeks. These fluctuations in the runtime of backtracking algorithms are characterized by the *heavy-tailed distribution*, which indicates that the trail of the distribution decays polynomially. As mentioned earlier, the survival function of a randomized algorithm is the probability that the underlying algorithm requires more steps than given steps t to solve the problem, given by:

$$Pr(T > t) = 1 - F(t) \tag{4.1}$$

where $F(t)$ is the cumulative distribution function of random variable T. The random variable T has heavy-tailed distribution if:

$$Pr(T > t) \sim C \cdot t^{-\alpha}, t > 0 \tag{4.2}$$

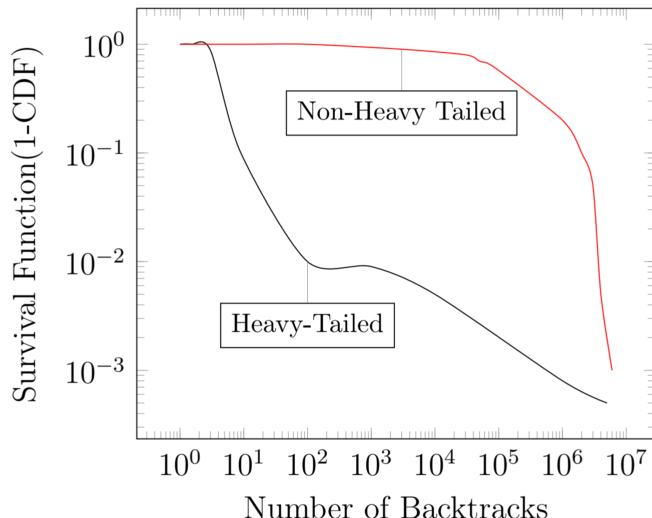


Figure 4.3: Comparison between heavy-tailed and non-heavy-tailed runtime distribution. CDF stands for Cumulative Density Function log-log scale. (Figure adapted from [69].)

where C and α are positive constants [73]. This distribution has infinite variance and finite mean when $1 < \alpha < 2$ and infinite variance and mean when $0 < \alpha \leq 1$ [73, 114]. A clear indication of heavy-tailed distribution for a given CSP $\mathcal{P}$ is that the log-log plot of the survival functions exhibits almost linear behavior with slope determined by α, as shown in Figure 4.3.

Although a CSP $\mathcal{P}$ whose runtime distribution is heavy-tailed indicates that backtrack algorithms might run for long execution time, it also suggests that $\mathcal{P}$ can be solved efficiently. By employing restarts, not only does a constraint solver circumvent a heavy-tailed long, but also increases the probability of encountering successful short runs [69]. By comparison, a CSP $\mathcal{P}$ that exhibits non-heavy-tailed behavior is inherently difficult to solve.

Gomes *et al.* [68, 69] contribute an in-depth study of how heavy-tailed behavior occurs. They introduce two useful notions to help explain the heavy-tailed distribution. First, an inconsistent subtree (IST) is a maximal subtree that does not contain a node that occurs in any solution. Second, the maximum depth of an inconsistent subtree is referred to as the inconsistent subtree depth (ISTD). They show that "when the backtrack search heuristic has a good probability of finding relatively shallow IST, and this probability decreases exponentially as the depth of the inconsistent subtrees increases, heavy-tailed behavior occurs [69]." Moreover, if the distributions of ISTD do not decrease exponentially, i.e., the variance of ISTD is not significant (cf., for example, Figure 3 of [69]), the problem exhibit non-heavy-tailed.

In essence, the root of an IST is a mistake defined in Section 4.1. For the heavy-tailed problems, an exponential decline in the distribution of the depth of early mistakes implies that restarts has a chance to circumvent the region of the search space that early mistakes frequently occur.

The methods of adding randomization to a backtracking search

There exist several possible choices to introduce randomization into complete, systematic, backtrack search procedures:

1. randomizing the variable ordering;
2. randomizing the value ordering;
3. randomizing tie breaking.

Randomizing the variable and value ordering first appear in the Ph.D. work of Harvey [164]. For the tie-breaking method, when several choices are ranked equally, the search strategy selects among them at random. This small adjustment can dramatically alter the region of the search space, especially when the heuristic makes the branching decision early on in the search. However, we often see that the search strategy only grades one choice as the highest score. To handle this situation, Gomes *et al.* [73] expand the choice set for random tie-breaking by allowing H-percent of the highest score to be regarded equivalent.

The methods of adding randomization mentioned above are search strategy independent. Recently, Demirović *et al.* [45] use *solution-based phase saving* to simulate local search behavior in complete constraint solvers for constrained optimization problems (COPs). The phase saving originates from SAT solvers. When phase saving works with restarts, the last value assigned to a variable in search is given priority over all other values in the domain of that variable in the next time the variable is branched on. The solution-based phase saving is a value-selection heuristic that differs from phase saving in respect: it gives priority to the values of the feasible solution found by the previous run. Thus, this method is worthless to solving CSPs. For COPs, however, it is desirable to attain another feasible solution with a better objective value. The authors note that combing solution-based phase saving, activity-based search, restarts, and nogood learning can add randomization to the systemic backtracking search, thereby increasing the diversification of search. They observe that the most active variables from the previous run are likely to be branched on first in

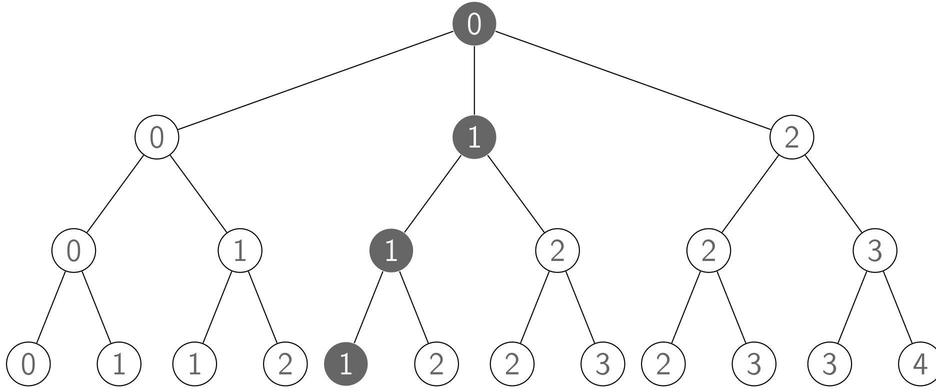


Figure 4.4: LDS search tree. The text inside of a node stands for the discrepancy of this node. (Figure adapted from [144].)

the current run because the variables selected first in the last run have low activity values due to involving fewer conflicts directly.[3]

4.2.2 Limited Discrepancy Search

We have discussed Limited Discrepancy Search (LDS) [91, 92] when reviewing Parallel Discrepancy Search. In this section, we shall concentrate on its probability model.

LDS is the first non-systematic search approach to be developed against early mistakes caused by the value-selection heuristic. When using LDS, the search tree is not traversed in a chronological backtracking manner. Visually, the search jumps from one region of the search tree to another. This jump phenomenon is the result of discrepancies that the search disobeys the value-selection heuristic. On k^{th} iteration, the probes, which can be drawn as paths from the root to leaf nodes, traverse nodes with up to k discrepancies. For example, the first probe of the 1th iteration is illustrated in the black nodes in Figure 4.4 in which the discrepancy occurs at the black node near the root.

To construct the probability model of LDS, Harvey first defines *the heuristic probability*, p, which is the likelihood of selecting a good left child at a good node. Besides, the mistake probability, m, is the likelihood that a randomly selected child of a good node is bad. To simplify the analysis, the heuristic probability p is assumed as constant throughout the search tree, although Harvey notes that p increases with depth since the heuristic is more informed in lower of the search tree.

[3]To better understand this observation, the reader needs to have an understanding of the activity-based search strategy [139]. We will also briefly introduce this search strategy in Section 6.2.

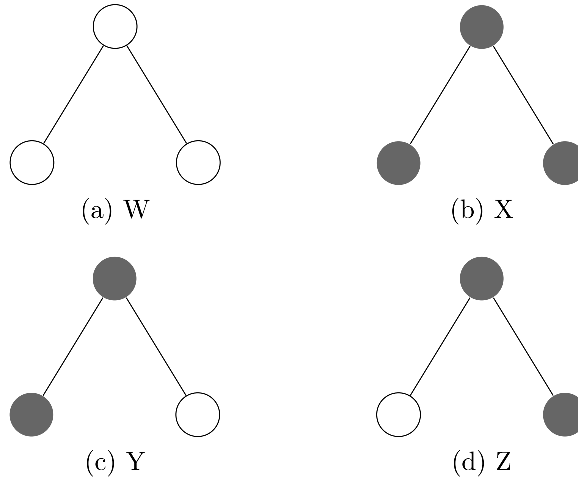


Figure 4.5: The four possible situations for a node and its children. A white node and a black node stand for a bad node and a good node, respectively. (Figure adapted from [91].)

For 2-way branching, he summarizes four possible situations, namely W, X, Y, and Z shown in Figure 4.5, for a node and its children. Figure 4.5a illustrates situation W where a bad node leads two bad children. The situations that a good node has at least one good child are shown in Figures 4.5b, 4.5c, and 4.5d, where situations X and Y have a good left child but situation Z does not. And the difference between situations X and Y are whether their left child is good node or not.

As we mentioned above, the heuristic probability p is the likelihood of selecting a good left child at a good node. Hence, the conditional probability of both situations X and Y occurred given that the parent node is good (i.e., $\neg W$) is exactly p because both of them have a good left child. Thus, we write:

$$Pr(X \vee Y \mid \neg W) = p \tag{4.3}$$

The mistake probability m is the likelihood a good node has a bad child. Therefore, the conditional probability of the situations that have a bad child, i.e., Y and Z, occurred given that is it not W is $2m$. Thus, we have:

$$Pr(Y \vee Z \mid \neg W) = 2m \tag{4.4}$$

The conditional probabilities of other possible combinations X, Y, and Z can be obtained:

$$\begin{aligned} Pr(X \mid \neg W) &= 1 - 2m & Pr(Z \mid \neg W) &= 1 - p \\ Pr(Y \mid \neg W) &= p + 2m - 1 & Pr(X \vee Z \mid \neg W) &= 2 - p - 2m \end{aligned} \tag{4.5}$$

The reason for defining the four situations is that the probability that a probe succeeds can be expressed by using these situations. Given a search tree of height d and on k^{th} iteration, the probability model can be represented by calculating the likelihood of finding a solution using i or less than i probes, $a_{i,d}$, in terms of p and m. The probability $a_{i,d}$ can be defined by:

$$a_{i,d} \equiv Pr(\exists_{l<i} succeed_l) \tag{4.6}$$

where $succeed_l$ denotes a successful run or multiple successful runs in i probes or fewer. In order to calculate the likelihood of finding a solution in i probes or fewer ($a_{i,d}$), it is better to consider its complement, namely the likelihood of not finding a solution in i probes or fewer, given by:

$$\forall_{d,i>0},\ a_{i,d} = 1 - (1 - a_{i-1,d}) \cdot (1 - s_{i-1}) \tag{4.7}$$

where $(1 - a_{i-1,d})$ is the likelihood of failing to obtain a solution in $i-1$ probes or fewer and $(1 - s_{i-1})$ the likelihood of failing to obtain a solution on next probe (probe i) given that all the previous probes have failed. And s_i is defined as the product of probabilities of all the nodes on probe i. That is,

$$s_i = \prod_{j=0}^{d-1} g_{i,j} \tag{4.8}$$

where $g_{i,j}$ is the probability of expanding to a good node from j on probe i given that all earlier probes have failed and is defined by:

$$g_{i,j} \equiv Pr(good_{i,j+1} \mid good_{i,j} \wedge \neg(\exists_{l<i} succeed_l)) \tag{4.9}$$

where $good_{i,j}$ denotes that node j of probe i is a good node.

In [203], Walsh pointed out that the combinatorics involved in computing $g_{i,j}$ are very complex. For instance, in the example illustrated in Figure 4.6, probe 2 ($P2$) and probe 3 ($P3$) share three common nodes in their path, but since their third nodes (i.e., node 3) are alternatives. These common nodes imply that the failure of the second probe decreases the likelihood of the success of the third probe, whereas the alternative nodes increase the likelihood of the success of the third probe. Since whether or not a node is good can depend on previous probe, Harvey derives expressions for $g_{i,j}$ for three cases, $j > i$, $j = i$, and $j < i$.

For $j > i$ the failure of previous probes cannot affect the success probability for the current node. For example, $g_{1,2}$ is unrelated to whether or not probe 0 ($P0$) succeeds (cf. Figure 4.6). Thus,

$$\forall_{j>i},\ g_{i,j} = Pr(good_{i,j+1} \mid good_{i,j}) = Pr(X \vee Y \mid \neg W) = p \tag{4.10}$$

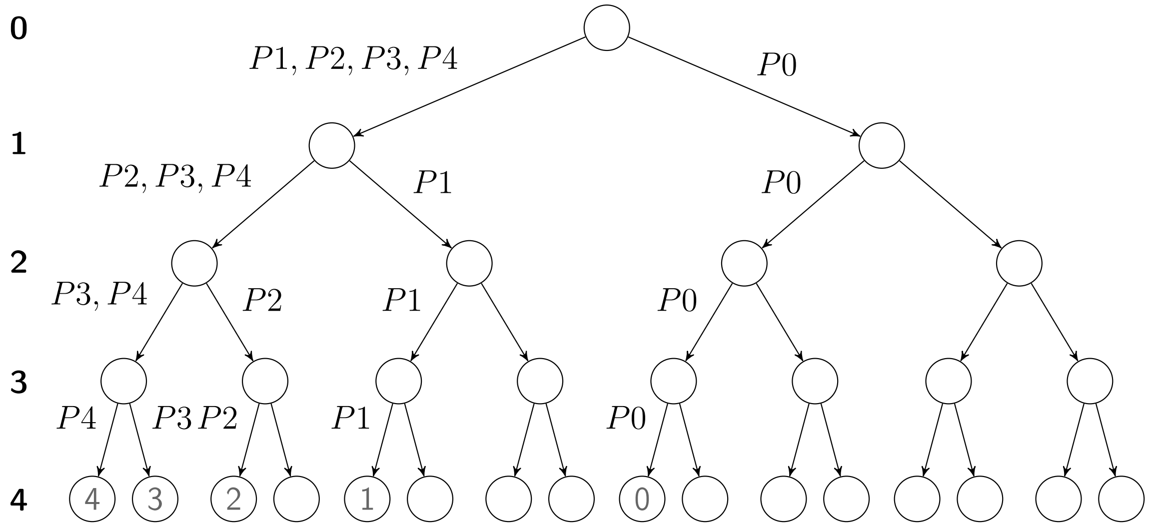


Figure 4.6: Five probes of the one discrepancy iteration on a binary search tree of height four. Pi above an arrow denotes that the node pointed by the arrow is visited on probe i. For example, $P3, P4$ means that a node is visited by both probe 3 and probe 4. The number i inside of a leaf node indicates this leaf node visited on probe i. In each probe, the nodes are numbered for each probe, starting with the root node (numbers in bold type), from zero to four.

The case that $j = i$ is also not affected by the failure of the earlier probes. But at this node, the probe is going to violate the heuristic and visit its left child (e.g., node 2 of $P2$ in Figure 4.6). Thus, situation Y should be replaced by situation Z in Equation (4.10), given by:

$$\forall_{j=i},\ g_{i,j} = Pr(good_{i,j+1} \mid good_{i,j}) = Pr(X \vee Z \mid \neg W) = 2 - p - 2m \qquad (4.11)$$

For the case $j < i$, we should take into consideration the influence of the failure of previous probes. Let us consider the probability, $g_{3,2}$, that node 3 of probe 3 is good (i.e., $good_{3,3}$) given that its parent in the same probe is good (i.e., $good_{3,2}$) and probe 2 failed (i.e., $\neg(\exists_{l<3} succeed_l)$). In this case, as illustrated in Figure 4.6, the parent, namely node 2 of probe 3, must be either in situation X or situation Y in order to guarantee that node 3 of probe 3 is good, denoted by $X_{2,2} \vee Y_{2,2}$. Note that $X_{i,j}$ means that node j of probe i is in line with situation X. There are three possibilities to ensure that node 2 of probe 3 is good and probe 2 failed: $X_{2,2} \wedge Z_{2,3}$, $Y_{2,2} \wedge W_{2,3}$, and $Z_{2,2} \wedge Z_{2,3}$. For instance, in probe 2, nodes 2 and 3 are good, but the left child of node 3 are bad, implying that a solution cannot be found in probe 2 and node 2 will be good in probe 3. Moreover, since node 3 will be good in probe 3, the situations of these nodes can be expressed as $X_{2,2} \wedge Z_{2,3}$. The probability of $g_{3,2}$ is computed from

Equation (4.9), given by:

$$\begin{aligned} g_{3,2} &= Pr(good_{3,3} \mid good_{3,2} \wedge \neg(\exists_{l<3} succeed_l)) \\ &= Pr(X_{2,2} \vee Y_{2,2} \mid X_{2,2}Z_{2,3} \vee Y_{2,2}W_{2,3} \vee Z_{22}Z_{2,3}) \end{aligned} \tag{4.12}$$

The process of reasoning of $g_{i,j}$ for $j < i$ is too long to cover due to limited space.[4] But we have already shown the basic idea of how Harvey constructs the probability model of LDS in terms of p and m. Harvey also gives a recursive definition of the likelihood of success for backtracking. He then performs a theoretical comparison of LDS with chronological backtracking (DFS). Given a problem of height 100 with $m = 0.1$, the probability of finding a solution for LDS rises steadily as the number of probes grows, with approximately eight-fold increased likelihood compared with DFS (cf., Figure 3.6 of [91]). As we have mentioned at the beginning of the Chapter, variable ordering heuristics tend to be less informed and is prone to error near the root of the search tree. Walsh [203] proposes Depth-bouned Discrepancy Search (DDS) to improve LDS by exploring more discrepancies at the top of search tree, but visiting fewer redundant nodes than LDS.

The theoretical analysis and the empirical study shows that the discrepancy strategies like LDS and DDS are effective methods of obtaining better feasible solutions in optimization problems in large and under-constrained search trees. However, the discrepancy strategies do not apply to insoluble CSPs, where the backtracking search tree is required to be traversed completely. In this case, it is completely unnecessary to use discrepancy strategies. Moreover, both the theoretical analysis provided by Harvey and the empirical study given by Walsh neglect to consider domain reductions brought on by constraint propagation. Thus, in practice, considerable large overheads caused by repeatedly visiting nodes and unbalanced search trees are to be expected when comparing the number of nodes explored by the discrepancy strategies and DFS. Just recently, Archibald *et al.* [10] present empirical comparisons, which show that DDS is not well-suited for unsatisfiable decision problems.

4.2.3 Parallel Portfolio Search

We have already introduced the parallel portfolio in Section 3.3. Depth-first backtracking is in the sense of audacious search strategy, and finding a relatively good variable orderings to avoid early mistakes is a challenging task. For a given CSP, especially for problem instances that exhibit heavy-tailed behavior, there exit significant runtime variance between different search strategies, between the same search

[4] The interested reader is directed to pages 61-66 of [91] for detailed reasoning.

strategy with randomization, between different restart policies, between different constraint solvers, between different CSP models, or even between different parameter settings.

One way to utilize this vulnerability is to run an ensemble of different assets of a sequential solver to solve the same CSP, simultaneously. As a result, one can expect a smaller overall runtime and a smaller variance to find a first solution. When a parallel portfolio consists of assets with deterministic algorithms (e.g., deterministic search strategy), the expected runtime is determined by the asset that is the first to obtain a solution.[5] In this case, the parallel portfolio cannot improve the performance better than the the fastest sequential asset; conversely, the parallel execution brings with unnecessary redundancy.

Nevertheless, if we combine different stochastic algorithms (e.g., adding randomization to the search strategy) into a *parallel stochastic portfolio*, the probability of finding a first solution increases. Let us assume that we have n workers, and the portfolio consists of n stochastic assets in which each asset A_i is executed on a different worker. Thus, $Pr(A_i = t)$ is given by the probability that the worker i $(1 \leq i \leq n)$ uses exactly t backtracking steps. Now, by Equation (4.1), the survival function of the stochastic algorithm A_i is $1 - Pr(A_i = t) = Pr(A_i > t)$. Then, the probability that at least one of the stochastic algorithms find a solution in t steps is given by:

$$Pr(P = t) = 1 - \prod_{i=1}^{n} Pr(A_i > t) \tag{4.13}$$

Before analyzing Equation (4.13), we first let an asset whose probability of finding a first solution with t steps is greater than zero (i.e., $Pr(A_i = t) > 0$) call an *effective asset*. Although Equation (4.13) is very simple, it indicates the following two benefits of employing parallel stochastic portfolio:

1. $Pr(P = t)$ is greater than the probability of any its assets of obtaining a solution with t steps when at least two of its assets are effective assets, which can be expressed by:

$$\begin{aligned} 1 \leq i < j \leq n, (\exists_i \ (Pr(A_i = t) > 0)) \wedge (\exists_j \ (Pr(A_j = t) > 0) \\ \Rightarrow \ 1 \leq i \leq n, \ \forall_i \ (Pr(P = t) > Pr(A_i = t))) \end{aligned} \tag{4.14}$$

   Therefore, executing a set of effective assets in parallel increases the likelihood of finding a solution with given steps.

[5]Please see page 44 for an definition of the asset.

2. $Pr(P = t)$ rises as the number of effective assets grows. Further, in theory, $Pr(P = t)$ approaches to 1 as the number of effective assets approaches to infinity. That is,

$$\lim_{k \to \infty} Pr(P = t) = \lim_{k \to \infty} 1 - \prod_{i=1}^{n} Pr(A_i > t) = 1 \tag{4.15}$$

where k, $1 < k \leq n$, is the number of effective assets. Equation (4.15) implies that any CSP $\mathcal{P}$ could be solved by using the parallel stochastic portfolio when given an infinite number of processors and effective assets.[6] Unfortunately, it is unrealistic to expect an infinite number of processors. But, it is not a problem to ensure a sufficient number of distinct assets for massively parallel processors. In Section 6.4, we will show an approach that generates a sufficient number of distinct assets.

We have shown theoretically that the parallel stochastic portfolio can be a significant improvement over sequential solving or a parallel portfolio with deterministic algorithms. We will discuss parallel stochastic portfolio at length in Chapter 6. In particular, the experimental results that test the optimistic outlook of Equation 4.13 will be given.

4.2.4 Embarrassingly Parallel Search

We have briefly introduced the embarrassingly parallel search (EPS) approach in the survey of Problem Decomposition (see Section 3.2). The EPS approach generates consistent partial assignments over a set of variables, each of which can be viewed as a subproblem that is not inconsistent with the constraint propagation. The rationale behind EPS is to achieve a balanced workload by assigning a larger number of subproblems to each worker because the differences in resolution times between subproblems cannot affect the total workload distribution. The simplest and most basic method of problem decomposition can be divided into three phases:

1. We select a static ordering of the variables.
2. We perform Depth-Bounded Depth First Search (DBDFS) to generate the right number of subproblems.

[6] Alternatively, put another way, if all the possible variable and value orderings for a given CSP are exhaustively tried in parallel, then the optimal search strategy is bound to be used, thereby finding a solution with n steps, where n is the number of variables of the given CSP.

3. We send the subproblems to the workers; each work receives the same number of subproblems.

The main challenge for problem decomposition is to identify the depth at which the frontier of depth bounded DFS contains the right number of subproblems. As introduced in Section 3.2, Malapert *et al.* [131, 169] proposed two methods for generating a correct number of subproblems: The top-down decomposition starts from the root node, and incrementally explores the next levels. In contrast, the bottom-up decomposition starts from a level deep enough and climbs back to the previous levels.

The EPS approach was intentionally developed for obtaining all the solutions of a given CSP $\mathcal{P}$ [170, 171, 131, 169]. Nevertheless, the EPS approach is also useful for obtaining a first solution. In this section, we first build a probability model to explain why the EPS approach can increase the likelihood of finding a first solution. Then, we show how EPS can offset early mistakes of ordering heuristic and why superlinear speedup can be expected when applying EPS to constraint solving.

The probability model

The underlying principle of accelerating constraint solving is the same for different variants of EPS, no matter what workload distribution mechanism or problem decomposition approach is employed. To illustrate the benefits of applying EPS to constraint solving, suppose that our goal is to find a first solution for a given CSP. As we will show, the probability of finding a first solution using EPS is increased compared to the sequential solving.

Let us assume that we have m processors (or workers), and the CSP $\mathcal{P}$ to be solved has only one solution and n variables in which k variables are used to decompose the original problem. Besides, EPS is a deterministic algorithm. In this case, $Pr(W_i = t)$ denotes that $worker_i$ obtains the solution using exactly t backtracks when given a distinct set of subproblems, and the following expression gives the probability function for EPS:

$$\sum_{i=1}^{m} Pr(W_i = t) \tag{4.16}$$

Each term $Pr(W_i = t)$ can be computed by:

$$Pr(W_i = t) = \prod_{j=1}^{k} p_{s_j} \cdot \prod_{j=k+1}^{n} p_{r_j} \tag{4.17}$$

where p_{s_j} is the probability of the subproblems assigned to $worker_i$ containing a value that participates in a solution. Furthermore, $\prod_{j=1}^{k} p_{s_j}$ is the probability of having a

subproblem (i.e., partial assignments) that can be extended to a solution, while p_{r_j} is similar to the probability we defined in Equation (4.8), which is the probability of extending to a good node under the subspaces defined by the given subproblems. Let us further assume that both parallel run and sequential run use the same variable ordering to compare the probability of EPS against that of sequential solving. As mentioned in Section 4.1 (page 52), the probability of finding a solution can be computed by taking the product of the probability that the search strategy selects a good node to be branched on, given by:

$$Pr(Seq = t) = \prod_{j=1}^{n} p_j \tag{4.18}$$

The EPS approach decomposes the CSP $\mathcal{P}$ into q subproblems that are consistent with constraint propagation, and then each worker works on its own $\frac{q}{m}$ subproblems. Thus, if the CSP $\mathcal{P}$ to be solved has one solution, there must be one worker obtained the subproblem that can lead to the solution. To this worker, $p_{s_j} > p_j$ because the variables used for problem decomposition have reduced domain size and $p_{r_j} > p_j$ because the domain values for these variables that are inconsistent with assigned subproblems are removed by constraint propagation.[7] Therefore, this worker has a higher probability of finding a solution in exactly t backtracks relative to sequential run:

$$\prod_{j=1}^{k} p_{s_j} \cdot \prod_{j=k+1}^{n} p_{r_j} > \prod_{j=1}^{n} p_j \tag{4.19}$$

As long as one worker increases the likelihood of finding a first solution, EPS can obtain a solution faster than sequential solving can, even though the rest of the workers have no probability of solving the problem. The foregoing analysis then allow us to make the following statement:

$$\sum_{i=1}^{m} Pr(W_i = t) > Pr(Seq = t) \tag{4.20}$$

Consider an extreme case that every worker is assigned only one subproblem of a computationally hard problem, and this subproblem is a partial assignment of a large set of variables (e.g., 20 variables) that can be extended to a solution. Then

$$\prod_{j=1}^{k} p_{s_j} \cdot \prod_{j=k+1}^{n} p_{r_j} \gg \prod_{j=1}^{n} p_j \tag{4.21}$$

[7] Our discussion is based on general cases, i.e., most of heuristics tend to be more accurate in a narrower domain. Indeed, an optimal heuristic can always select the best node no matter how large the domain size is. However, finding optimal heuristic is a computationally difficult task, and it is unnecessary to employ parallelism for a first solution if we really have such optimal heuristic.

Because the terms p_{s_j} in Equation (4.21) are equal to one. Note that all this would require a sufficiently large number of workers. In this case, EPS has a much higher probability of obtaining a solution than does a sequential run. That is,

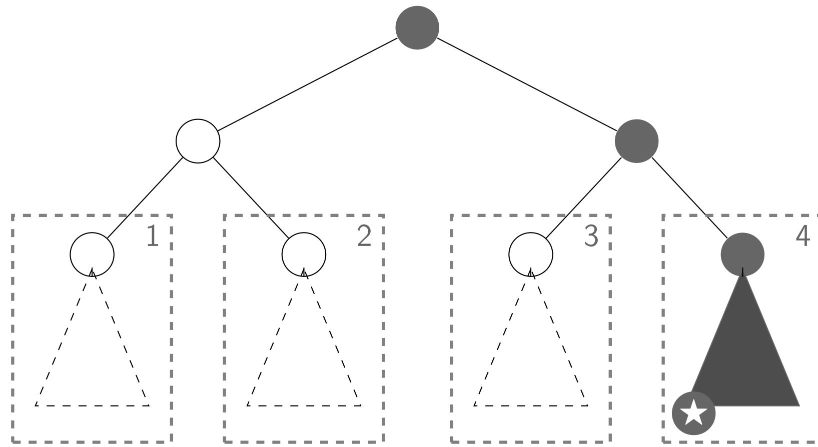
$$\sum_{i=1}^{m} Pr(W_i = t) \gg Pr(Seq = t) \tag{4.22}$$

Indeed, each worker in the EPS approach is supposed to deal with a large number of subproblems to avoid unbalanced workload (e.g., mostly between 10 and 100 subproblems per worker [131]). But, when given the same set of subproblems, a first solution is more likely to be obtained in less time by more workers ($\frac{q}{m} \leq 1$), if we ignore load balancing.

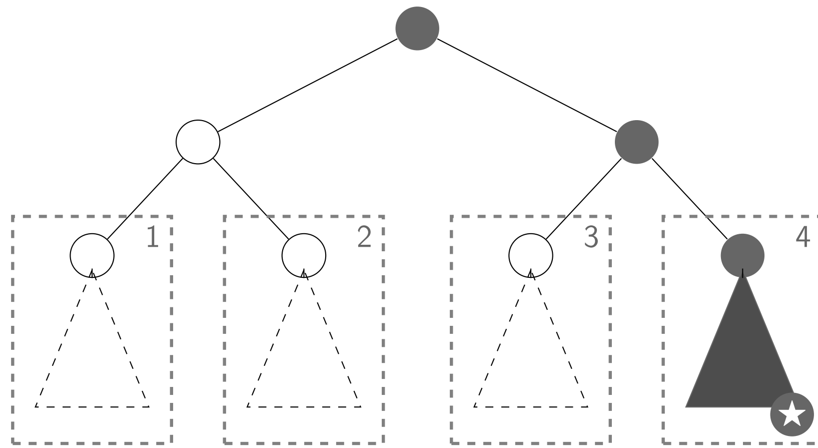
The Potential for Speedup

We have shown theoretically that the EPS approach can be a significant improvement over sequential solving when the goal is to find a first solution. In the following, we discuss the possibilities that we can expect to gain a speedup or even superlinear speedup. As mentioned in Section 3.2, we view EPS as a parallel method that explores different subtrees in parallel since different subproblems lead to explore different subtrees. With problem decomposition, we cannot predict whether or not a subproblem can lead to a solution. Also, we cannot predict the resolution time of each subproblem. Thus, it is difficult to ensure a fixed amount of work between workers, even though each worker is assigned many subproblems. We should not always expect to gain a linear speedup. Instead, the speedup obtained will depend not just on the number of processors used, but also on the way of distributing subproblems.

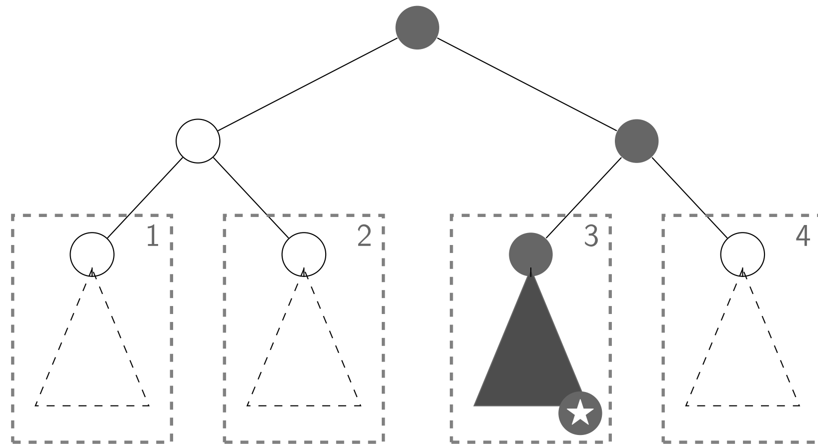
If our goal is to obtain a first solution, there exist various possible outcomes, as illustrated in Figures 4.7 and 4.8. Here, we assume the search space being partitioned among four workers and the depth-first search iterating over children from left to right. To simplify the analysis, we assume that both the sequential run and parallel run employ the same search strategy and consistency-enforcing algorithm. Moreover, the parallel search preserves the sequential search order, e.g., we do not distribute subproblems in an out-of-order manner. However, it may be possible to further improve the performance by also introducing randomness for workload distribution or other techniques such as different search strategies for different workers in practice. Note that the nodes in Figures 4.7 and 4.8 are not eliminable for the given constraint propagation level.



(a) Superlinear Speedup

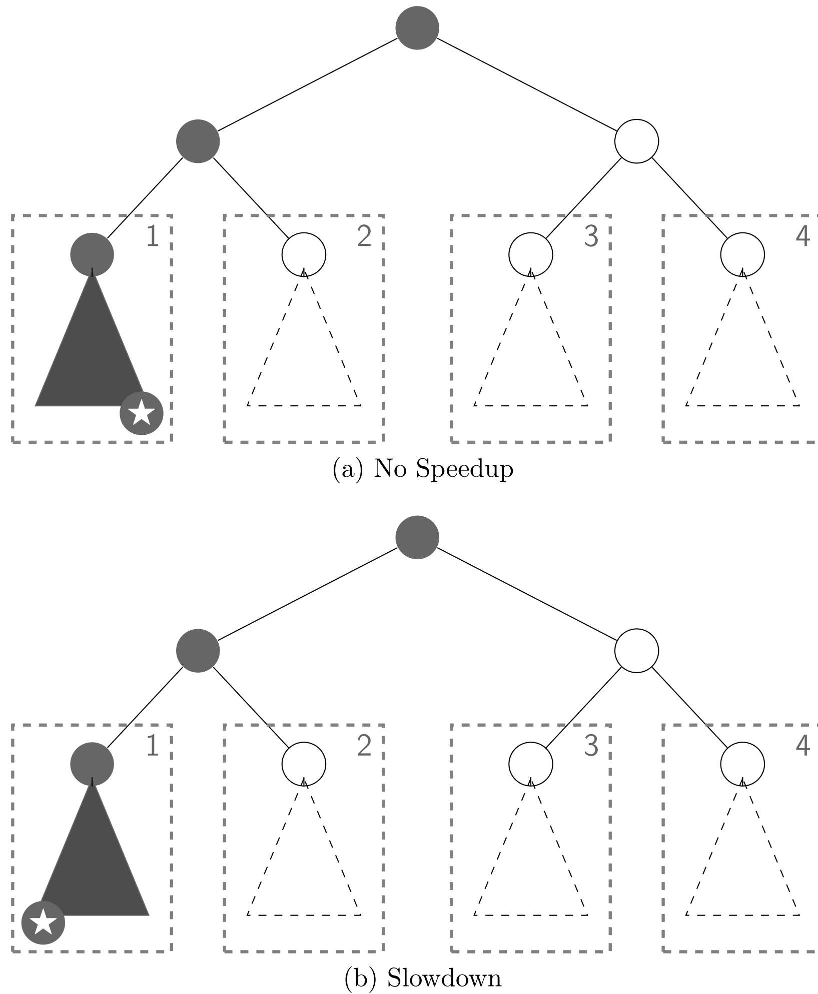


(b) Linear Speedup



(c) Sublinear Speedup

Figure 4.7: The typical possibilities for speedup in EPS. The good and bad nodes are still black and white, respectively; the hollow triangles with a dashed border and the black triangle stand for empty subtrees and subtree with a solution, respectively.



(a) No Speedup

(b) Slowdown

Figure 4.8: Other two possibilities for speedup in EPS.

We can gain a superlinear speedup because one of the extra workers obtains a first solution that is in the leftmost part of the subtree, with possibly an order of magnitude speedup from the sequential run or even more, as depicted in Figure 4.7a. In this situation, the sequential run usually gets stuck in a barren subtree first due to early mistakes made by the search strategy. Whereas the strong commitments to the variable selections made by a search strategy at the early stages in the search (i.e., early mistakes) can be avoided by exploiting parallelism. Moreover, the parallelism can prune the search for a first solution in comparison to sequential solving. In other words, the parallel run finds a solution after exploring fewer nodes than the sequential run, making it possible to obtain superlinear speedup. The irregularity of subproblem 4 has no impact on the parallel execution time of EPS. That is to say, even if the res-

olution time required by completely searching subproblem 4 takes much longer than other subproblems do, we can still achieve a superlinear speedup since the solution is in the leftmost of the search tree of subproblem 4 (see ✪ in Figure 4.7a). By contrast, if the solution is in the very rightmost part of the subtree and the subtree is too large to search thoroughly (cf. Figure 4.7b), the extra workers end up contributing nothing to find a first solution.

We may also achieve a linear speedup (speedup of n with n processors) or a sublinear speedup ($1 < speedup < n$ with n processors), as shown in Figures 4.7b and 4.7c. In both cases, the first solution is located at the rightmost part of the subtree (see subtree 4 of Figure 4.7b and subtree 3 of Figure 4.7c). The location of the solution in the subtrees implies that all the nodes of the subtree are required to be visited before obtaining the solution in the parallel run. Besides, the sequential run also need explore all the subtrees with a smaller number (e.g., subtrees 1, 2, and 3 in Figure 4.7b) if the parallel search preserves the sequential search order.

Not only may the extra workers contribute nothing to find a first solution, but they may also decelerate the resolution process, as depicted in Figure 4.8. Applying parallelism to solving such problems is meaningless since the sequential run can already solve the problems in a reasonable time. Especially for the situation given in Figure 4.8b, the search strategy can always select the most suitable candidate variable and value (i.e., good node). Thus, parallelism leads to a slowdown due to the execution time of the non-parallelizable part. However, as the probability product indicated in Equation (4.18), even if the search strategy makes mistakes very rarely, the early mistakes in the search can still cause the failure of the search, especially when solving a large computationally hard problem. Thus, we should not expect the situation given in Figure 4.8b to occur when solving a large computationally hard problem.

In summary, the location of the solution in the subtrees after problem decomposition is the key to success in finding a solution when solving a large computationally hard problem. If we can find a solution for such a problem in parallel and the sequential run fails, we probably would have gained a superlinear speedup. This is because heuristics tend to be less informed and make more mistakes at the top of the search tree. The EPS approach address this problem from two aspects: First, parallelism can hedge against weak heuristic choices. Second, since the values that are inconsistent with assigned subproblems are removed, and the variables used for problem decomposition have reduced domain sizes, the heuristics must be more reliable in the parallel run than in sequential run.

4.3 Conclusion

Four main techniques for addressing the failure of DFS caused by early mistakes have been investigated. Although these techniques are orthogonal, the rationale for them is to introduce diversity to avoid commitment to the wrong branching decisions made by the heuristic early on in the search. In our analysis of parallel portfolio search and problem decomposition, we have attempted to provide explanations on why they can lead to performance improvements.

We are optimistic about the applicability of parallel portfolio search and problem decomposition to finding the first solution of computationally hard problems. Thus, we argue that parallel constraint solving, including the parallel stochastic portfolio and the EPS approach, is an effective approach to tackle computationally hard problems not only because more nodes are explored simultaneously, but also because it can remedy the early mistakes caused by search strategies.

Chapter 5

Case Studies of the EPS Approach

The idea of the original EPS approach is to partition a problem into many subproblems using a Depth-Bounded Depth-First Search (DBDFS). The decomposition method automatically decides the depth of DBDFS according to the number of workers and the expected number of subproblems per worker. Then, each worker solves the same number of subproblems that are not trivially detected as inconsistent so that workload balancing can be achieved when searching for all the solutions to the problem.

Nevertheless, we can also apply the EPS approach to solve a computationally hard problem to obtain a first solution. Indeed, it might seem self-evident that the EPS approach can be used to tackle many large and hard problem instances encountered in real-life applications and combinatorics. However, as mentioned in the previous chapter, there are various possible outcomes when applying EPS on such problems, and we have analyzed the search space properties that lead to different possibilities.

In this chapter, we verify the theoretical arguments by solving three hard problems taken from the Problem Library for Constraints (i.e., CSPLib [38]), including the Social Golfer Problem (SGP) [89], the Traveling Tournament Problem with Predefined Venues (TTPPV) [160], and the Talent Scheduling Problem (TS) [190]. We chose them for a number of reasons. These problems are related to scheduling problems with apparent practical significance and have been widely studied in the constraints community. Besides, we can increase the difficulty of these problems infinitely by increasing the size of the instances, which is valuable for future research. To tackle larger and harder instances for these problems in parallel, we provide customized decomposition methods for generating subproblems. Moreover, we improve or redesign the CSP models since the formulations of a problem can significantly affect the problem solving efficiency [192].

In order to make this chapter more self-contained, we report the detailed problem-solving process and our findings for the three problems separately, given in Sections 5.1 through 5.3. Finally, we conclude in Section 5.4.

5.1 Social Golfer Problem

In this section, we show how to attain solutions for some open instances of the Social Golfer Problem (SGP); the emphasis is on building the CSP model and exploiting parallelism (i.e., the EPS approach) to solve new instances. Besides, we survey the extensive literature on solving the SGP, including the best results they have achieved, and analyse the cause of difficulties in solving the SGP. This section is a revised version of our previous works [125, 124] and is organized as follows. Section 5.1.1 analyses the causes of the difficulties of solving the SGP in the context of the CSP and introduces the constraints required to encode the model. A modeling approach improved on the model proposed by [13] are described in Section 5.1.3, and some instance-specific constraints are presented in Section 5.1.4. In addition, we elaborate on how to employ the EPS approach to solve the SGP in Section 5.1.5. We then present the experimental results in Section 5.1.6. In Section 5.1.7, we classify the researches on the SGP and also survey the studies relevant to the SGP outside the context of the CSP. We finally conclude in Section 5.1.8.

5.1.1 The Introduction of Social Golfer Problem

The SGP, i.e., problem 010 in CSPLib [89], is a typical combinatorial optimization problem that has attracted significant attention from the constraints community because of its highly symmetrical and combinatorial nature. The original SGP, which was posted to `sci.op-research` in May 1998 [89, 197], can be stated as follows: In a golf club, 32 golfers wish to play in foursomes for 10 weeks. Is it possible to find a schedule for maximum socialization; that is, each golfer can only meet any other no more than once? In fact, the SGP dates back to Thomas Penyngton Kirkman's 1850 query [108] in which the number of golfers and the size of a group are 15 and 3, respectively.

Hence, we can readily generalize the SGP to the following: The SGP consists of scheduling $n{=}g * s$ players into g groups of s players for w weeks so that any two players are assigned to the same group at most once in w weeks. According to the constraints community's convention on this problem, an instance of the SGP is denoted by a triple g-s-w, where g is the number of groups, s is the number of players

within a group, and w is the number of weeks in the schedule. In addition, we can also regard the SGP as a discrete optimization problem that maximizes the number of weeks w^* for a given g and s, where $w^* \leq \frac{g \cdot s-1}{s-1}$. Clearly, a solution for an instance g-s-w^* indicates itself as the solution for all instances g-s-w with $0 < w < w^*$ by deleting any row(s) of the solution of g-s-w^*. In practice, the computational difficulty of solving the g-s-w^* and g-s-w instance is often not in the same order of magnitude due to the huge difference in the solution density of two instances. For example, the 8-4-9 instance can be solved in a second on a state-of-the-art constraint solver. The 8-4-10 instance, by contrast, is still unsolvable for constraint approach at the time of writing, although at least three non-isomorphic solutions are known to exist [197]. In light of this, this research concentrates on solving the g-s-w^* instances with maximal number of weeks, which we call *the full instance*. For instance, Table 5.1 depicts one solution for the full instance 7-3-10, where each row is a permutation of 21 ($7 \cdot 3 = 21$) golfers and consists of 7 groups, and every group comprises 3 golfers. Since no two golfers play in the same group more than once, any two groups from different rows in Table 5.1 can share at most one golfer.

Week \ Group	1			2			3			4			5			6			7		
0	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***
1	***0***	3	6	***1***	9	12	***2***	15	18	4	7	16	5	10	13	8	11	19	14	17	20
2	***0***	9	13	***1***	3	16	***2***	8	10	4	6	20	5	11	15	7	14	18	12	17	19
3	***0***	16	20	***1***	5	18	***2***	3	11	4	9	19	6	14	15	7	10	12	8	13	17
4	***0***	7	11	***1***	4	15	***2***	6	12	3	13	20	5	8	14	10	16	19	9	17	18
5	***0***	8	15	***1***	10	14	***2***	9	20	3	7	19	4	13	18	5	12	16	6	11	17
6	***0***	12	18	***1***	8	20	***2***	5	19	3	10	17	4	11	14	7	9	15	6	13	16
7	***0***	4	10	***1***	7	17	***2***	14	16	3	8	18	5	6	9	13	15	19	11	12	20
8	***0***	5	17	***1***	6	19	***2***	7	13	3	9	14	4	8	12	10	15	20	11	16	18
9	***0***	14	19	***1***	11	13	***2***	4	17	3	12	15	5	7	20	8	9	16	6	10	18

Table 5.1: A solution for 7-3-10. The text in bold indicates that the values have been initialized before search. (Table adapted from [125, 124].)

The research on the SGP is not only meaningful to itself, but also for other Constraint Satisfaction Problems (CSPs) that exhibit symmetrical and combinatorial nature. For example, balanced incomplete block design (BIBD), problem 028 in CSPLib [163], is a standard combinatorial problem from design theory and also a test bed for symmetry breaking methods. Moreover, steel mill slab design [141], which is a real industry problem, can also benefit from the SGP. The reason is that we are likely to face the same difficulties as the SGP when solving other CSPs through the constraint satisfaction approach.

5.1.2 Background Information

In this section, we first explain why it is challenging to solve the SGP in the context of the CSP. Then we review some constraints relevant to our model of the SGP.

5.1.2.1 The Difficulties of Solving the SGP

At first sight, the SGP is a simple-sounding question. And indeed, one can model the problem by using several frequently-used constraints derived from the problem definition. The constraint satisfaction approach, however, still has enormous difficulties in obtaining the solution even for some small instances (e.g. 7-4-9, 8-4-10, etc.).[1] We believe that the following two reasons result in the difficulties of the SGP:

The Difficulty Caused by Symmetries. The inherent highly symmetrical nature of the SGP cannot be entirely known before solving process. There exist four types of symmetries: (1) We can permute the w weeks, that is, arbitrarily ordered weeks ($w!$ symmetries). (2) Within each week, we can (separately) permute the g groups, that is, interchangeable groups inside weeks ($g!$ symmetries). (3) Within each group, we can permute the s players, that is, interchangeable players inside groups ($s!$ symmetries). (4) Finally, we can also permute the n players ($n!$ symmetries), which can also be viewed by renumbering n golfers. The first three types of symmetries can be relatively easy removed through model reformulation or static symmetry breaking constraints. Nevertheless, it is difficult to eliminate all the symmetries among players caused by the fourth type of symmetry. For example, if players $[16, 17, 18, 19, 20, 21]$ in Table 5.1 replace with $[19, 20, 21, 16, 17, 18]$ in turn, an isomorphism of the solution depicted in Table 5.1 will be generated even if the first row of the solution is fixed with $[1, \ldots, 21]$. Apparently, we are unable to foresee this symmetry before search. Consequently, the unnecessary symmetrical search space is explored redundantly.

The Difficulty Caused by Early Mistakes. It is common to observe that some unfortunate choices of variables early on are to blame for a long-running search process [73]. The SGP has also been experienced such phenomena. More precisely, invalid partial assignments lead to the backtrack search to trap in a barren part of the search space since no consistent assignment can be found. More importantly, it is often hard to determine the usefulness of a partial assignment until almost all variables are instantiated; and these invalid partial assignments predominate in the overall search space.

[1] By a "small instance", we mean the number of golfers rather than the time complexity.

5.1.2.2 Global Constraints for Modelling SGP

We now briefly introduce the global constraints involved in the CSP model of the SGP. The *allDifferent*[2] constraint is the most influential global constraint in constraint programming and widely implemented in almost every constraint solver, such as Choco solver [165] and Gecode [186]. Formally, let X_a denote a subset of variables of X, the *allDifferent*(X_a) constraint can be defined as:

$$\forall_{x_i \in X_a} \forall_{x_j \in X_a} \ (i \neq j \implies x_i \neq x_j)$$

The *global cardinality constraint* $GCC(X_g, V, O)$ is defined using two lists of variables X_g and O, and an array of integer values V, where $X_g = \{x_l, \ldots, x_m\} \subseteq X$ and O is a list of variables not defined in X and predefines the range of the number of occurrences for each value in V. The GCC constraint restricts each value $v_i \in V$ appearing exactly o_{i_j} times in X_g, where o_{i_j} is in the domain of o_i and $o_i \in O$. More formally:

$$\{(d_l, \ldots, d_m)|\ d_l \in D_{(x_l)} \wedge, \ldots, \wedge d_m \in D_{(x_m)} \wedge \forall_i \forall_{o_j \in O_i} \ (occur(v_i, (d_l, \ldots, d_m)) = o_j)\}$$

where *occur* counts the number of occurrences of v_i in $(d_l, \ldots, d_m)$.

The *count* constraint is similar to the GCC, but with the restriction for only one value. More precisely, the *count*(v, X_c, O) constraint only restricts the number of occurrences of value v for the list of variables $X_c = \{x_l, \ldots, x_m\}$, given by:

$$\{(d_l, \ldots, d_m)|\ d_l \in D_{(x_l)} \wedge, \ldots, \wedge d_m \in D_{(x_m)} \wedge \forall o_j \in O \ (occur(v, (d_l, \ldots, d_m) = o_j)\}$$

The *table* constraint is another one of the most frequently-used constraints in practice. For an ordered subset of variables $X_o = \{x_i, \ldots, x_j\} \subseteq X$, a positive (negative) *table* constraint defines that any solution of the CSP P must (not) be explicitly assigned to a tuple in the tuples that consists of the allowed (disallowed) combinations of values for X_o. For a given list of tuples T, we can state the positive *table* constraint as:

$$\left\{(d_i, \ldots, d_j) \mid d_i \in D_{(x_i)}, \ldots, d_j \in D_{(x_j)}\right\} \subseteq T$$

Finally, the *arithm* constraint is used to enforce arithmetical relations between integer variables or between integer variables and integer values. For example, an integer value $v \in D_{(x_i)}$ can be assigned to an integer variable x_i by using the *arithm* constraint, i.e., $x_i = v$.

[2]This dissertation follows the naming convention and order of the arguments of constraints in Choco solver.

5.1.3 The Basic Model

There are various ways of modeling the SGP as a CSP proposed in the literature, which is one of the reasons why the problem is so compelling. Here, we use a model improved on the model presented in [13] due to its untapped potential. Specifically, we can add more constraints into the model to tackle larger instances piece by piece.

The decision variables of our model is a $w \times n$ matrix G in which each element $G_{i,j}$ of the matrix G represents that player j is assigned to group $G_{i,j}$ in week i. Hence, the domain of decision variable $G_{i,j}$ is a set of integers $\{1 \ldots g\}$, where $0 \leq i < w,\ 0 \leq j < n$.[3] The major advantage of the decision variables defined in this model is that the range of the variables are reduced from $\{1 \ldots n\}$ to $\{1 \ldots g\}$ while keeping an unchanged number of variables, compared with the naive model the derived from the problem definition.

We mentioned, in Section 5.1.2.1, that symmetries among players are difficult to handle and only dynamic checks can remove them completely. However, we can partially eliminate the symmetries among players by fixing the first week, namely the first row of the matrix G, defined by:

$$\forall_{j \in J}\ (G_{0,j} = j/s + 1), J = \{j \in \mathbb{Z}|\ 0 \leq j < n\} \tag{5.1}$$

where the operator "/" denotes integer division (cf. Table 5.2). Equation (5.1) produces a sequence of integers from 1 to g in non-descending order, and every integer continuously repeats itself exactly s times. Moreover, we can also freeze the first s columns by assigning the first s players to the first s groups after the first week (cf. Table 5.2), given by:

$$\begin{aligned} \forall_{i \in I} \forall_{j \in J}\ (G_{i,j} = j + 1) \\ I = \{i \in \mathbb{Z}|\ 0 < i < w\}, J = \{j \in \mathbb{Z}|\ 0 \leq j < s\} \end{aligned} \tag{5.2}$$

By applying Equation (5.2) to the model, we can significantly reduce the search space. Note that, in the implementation, we can realize Equations (5.1) and (5.2) by either restricting the domain of the variables or using the *arithm* constraint. Therefore, we use the term equation instead of constraint.

By the definition of the SGP, n different players are divided into g groups, which implies that each group includes exactly s players. Thus, the constraints required by

[3]In Section 5.1, we follow the Zero-based index.

Week \ Player	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	***1***	***1***	***1***	***2***	***2***	***2***	***3***	***3***	***3***	***4***	***4***	***4***	***5***	***5***	***5***	***6***	***6***	***6***	***7***	***7***	***7***
1	*1*	*2*	*3*	1	4	5	1	4	6	2	5	6	2	5	7	3	4	7	3	6	7
2	*1*	*2*	*3*	2	4	5	4	6	3	1	3	5	7	1	6	5	2	7	6	7	4
3	*1*	*2*	*3*	3	4	2	5	6	7	4	6	3	6	7	5	5	1	7	2	4	1
4	*1*	*2*	*3*	4	2	5	3	1	5	7	6	1	3	4	5	2	6	7	7	6	4
5	*1*	*2*	*3*	4	5	6	7	4	1	3	2	7	6	5	2	1	6	7	5	4	3
6	*1*	*2*	*3*	4	5	3	7	6	2	6	4	5	1	7	5	6	7	4	1	3	2
7	*1*	*2*	*3*	4	1	5	5	2	4	5	1	7	7	6	3	6	3	2	4	6	7
8	*1*	*2*	*3*	4	5	1	2	3	5	4	6	7	5	3	4	6	7	1	7	2	6
9	*1*	*2*	*3*	4	3	5	7	5	6	6	7	2	4	2	1	4	6	3	7	1	5

Table 5.2: A solution is obtained by our model for 7-3-10 instance, and it is equivalent to the solution depicted in Table 5.1. The bold-italic text indicates that the values have been initialized before search; the italic text stands for the values frozen by Constraints (5.2). (Table adapted from [124, 125].)

this property, which are imposed on the rows of the matrix G, can be stated as:

$$\begin{aligned} \forall_{i \in I} \ (GCC(G_{i,*}, V, O)) \\ I = \{i \in \mathbb{Z} | \ 0 < i < w\} \ , \ V = [1, \ldots, g] \\ \forall_{j \in J} \ (o_j \in O = [s, s]), \ J = \{j \in \mathbb{Z} | \ 0 \leq j < g\} \end{aligned} \tag{5.3}$$

where the length of O is g. The constraints (5.3) ensure that every value in the array of integers $[1 \ldots g]$ must occur exactly s times in all the rows of the matrix G (cf. Table 5.2).[4]

The restriction that no player meets any other player more than once can be interpreted as saying that no two columns of the matrix G have the same value at the same row more than once, given by:

$$\begin{aligned} \sum_{0 \leq i < w} \mid G_{i,j_1} - G_{i,j_2} = 0 \mid \leq 1 \\ j_1 \in \mathbb{Z}, \ j_2 \in \mathbb{Z} \ , 0 \leq j_1 < j_2 < n \end{aligned} \tag{5.4}$$

Constraints (5.3) and (5.4) are the only two constraints presented in [13]. In particular, unlike [13], we implement Constraint (5.4) in a different way to avoid using the *reified* constraints because these constraints often slow the resolution speed down in solvers like Choco [122, 165]. Specifically, the need for the *reified* constraints can be bypassed by introducing a $w \times m$ matrix C. We subtract every column from all other columns in the matrix G and the differences between two columns of the matrix G are assigned to a column of the other matrix C. Therefore, the number of columns of

[4]For the *globalCardinality* constraint in Choco solver, we can ensure that a value $v_j \in V$ occurs exactly s times by setting the upper bounder and lower bounder of $o_j \in O$ to s.

matrix C must be $m = \binom{n}{2}$. Having defined the matrix C, the two matrices G and C are linked by the equations expressed by *arithm* constraints, given by:

$$\begin{aligned} \forall_{i \in I} \forall_{j_1 \in J_1} \forall_{j_2 \in J_2} \; (G_{i,j_1} - G_{i,j_2} = C_{i,j_3}) \\ (j_1 < j_2) \wedge (0 \leq j_3 < m) \wedge (j_3 \in \mathbb{Z}) \\ I = \{i \in \mathbb{Z} |\; 0 \leq i < w\} \\ J_1 = \{j_1 \in \mathbb{Z} |\; 0 \leq j_1 < n\} \\ J_2 = \{j_2 \in \mathbb{Z} |\; 0 \leq j_2 < n\} \end{aligned} \tag{5.5}$$

Next, we impose the *count* constraint on every column of the matrix C so that the number of occurrences of value 0 on each column is no more than once due to the requirement that no two players can meet twice. So the constraints are defined by:

$$\begin{aligned} \forall_{j \in J} \; (count(0, C_{*,j}, occ)) \\ J = \{j \in \mathbb{Z} |\; 0 \leq j < m\}, \; occ = [0, 1] \end{aligned} \tag{5.6}$$

where occ is an integer variable whose domain is $\{0, 1\}$. Thus, the conjunction of Constraints (5.5) and (5.6) can logically realize the restrictions required from Constraint (5.4). Not only that, Constraints (5.5) and (5.6) can avoid the performance degradation that would be introduced by the use of the *reified* constraints.

So far, all the constraints as mentioned earlier have fully satisfied all the restrictions defined by the definition of the SGP and can be used to solve some small instances (e.g., 3-3-4, 5-3-7).[5] Nevertheless, we can further shrink the search space by placing logically redundant *implied constraints* without changing the set of solutions [192]. Equation (5.1) has already fixed the first row of the matrix G, which implies that those players who have met in the first week cannot play in the same group in the subsequent weeks. Therefore, the *allDifferent* constraint can be used to enforce the groups of these players are pairwise distinct after the first week, and we express these *allDifferent* constraints by:

$$\begin{aligned} \forall_i \forall_{(j \neq j') \wedge (j/s = j'/s)} \; (G_{i,j} \neq G_{i,j'}) \\ i \in \{i \in \mathbb{Z} |\; 0 \leq i < w\} \\ j, j' \in \{x \in \mathbb{Z} |\; 0 \leq x < n\} \end{aligned} \tag{5.7}$$

In summary, the basic model comprises Equations (5.1), (5.2), and Constraints (5.3), (5.5), (5.6), and (5.7). Nevertheless, the problem-solving ability of this model can be

[5]Our experiments showed that the model consisting of only these constraints were not able to solve the instances larger than 5-3-7.

greatly improved by the introduction of additional constraints, such as static symmetry breaking constraints, and the constraints derived by instance-specific pattern. In the subsequent sections, we will present the additional constraints dedicated to different types of instances based on this model and discuss how to solve the instances, which cannot be solved sequentially, in parallel.

5.1.4 Instances Solved Sequentially

For a given number of groups g and a group size s, our goal is to compute a first solution for a full instance g-s-w^*, where w^* represents the maximum number of weeks. In this section, we consider a particular type of instance s-s-$(s{+}1)$, which means that the number of groups in each week is the same as the number of players in the groups, and the number of weeks is equal to the number of groups within each week plus one. Moreover, the number of weeks is maximized because $\frac{s \cdot s - 1}{s-1} = s+1$. The specific properties of the instances of the form s-s-$(s{+}1)$ enable us to discover instance-specific constraints. Furthermore, we utilize observed patterns from the relatively small instances to deduce more instance-specific constraints for the instances of the form *odd-odd-(odd+1)* (henceforward, *o-o-(o+1)*), especially for the form *prime-prime-(prime+1)* (henceforward, *p-p-(p+1)*), and the form *even-even-(even+1)* (henceforward, *e-e-(e+1)*).

Before introducing the constraints, we first define submatrices of the matrix G formed by removing the first row of G and selecting columns $[j \ldots (j+s-1)]$, where j must be divisible by s, i.e., $j \% s = 0$.[6] Thus, G has exactly s such $(w-1) \times s$ submatrices, each of which has $w-1$ rows and s columns. The i-th submatrix of G is denoted by GS_i, where $0 \leq i \leq s$-1 (cf. Table 5.3).

Week \ Player	0	1	2	3	4	5	6	7	8	9	10	12	13	13	14	15	16	17	18	19	20	21	22	23	24
0	*1*	*1*	*1*	*1*	*1*	*2*	*2*	*2*	*2*	*2*	*3*	*3*	*3*	*3*	*3*	*4*	*4*	*4*	*4*	*4*	*5*	*5*	*5*	*5*	*5*
1	*1*	*2*	*3*	*4*	*5*	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
2	*1*	*2*	*3*	*4*	*5*	2	5	1	3	4	3	1	4	5	2	4	3	5	2	1	5	4	2	1	3
3	*1*	*2*	*3*	*4*	*5*	3	1	4	5	2	2	5	1	3	4	5	4	2	1	3	4	3	5	2	1
4	*1*	*2*	*3*	*4*	*5*	4	3	5	2	1	5	4	2	1	3	3	1	4	5	2	2	5	1	3	4
5	*1*	*2*	*3*	*4*	*5*	5	4	2	1	3	4	3	5	2	1	2	5	1	3	4	3	1	4	5	2

Table 5.3: A solution of 5-5-6 expressed by groups. It can be converted to the solution expressed by players easily. The submatrices GS_1 and GS_2 are written in red. (Table adapted from [125, 124].)

[6] The % (modulo) operator yields the remainder from the division of the first operand by the second.

5.1.4.1 7-7-8 etc.

For the instances of the form s-s-(s+1), every player must play with every other exactly once since $s * s$-1 is divisible by s-1. Thus, players whose number is greater than s must meet every player whose number is less than s exactly once in every week except the first week since the first row and the first s columns of the matrix G are frozen by Equation (5.1) and (5.2). To put it another way, since the first s players, in turn, are assigned to the first s groups after the first week and there are only s groups within each week, all the rest of n-s players have to be assigned to these s groups to avoid meeting the first s players more than once. Based on this analysis, we place the following constraints on the columns of the matrix G:

$$\begin{gathered} \forall_{(i \in I) \wedge (i' \in I) \wedge (i \neq i')} \forall_{j \in J} \; (G_{i,j} \neq G_{i',j}) \\ I = \{x \in \mathbb{Z} | \; 0 < x < w\}, \; J = \{x \in \mathbb{Z} | \; s \leq x < n\} \end{gathered} \tag{5.8}$$

Constraint (5.8) states that starting with submatrix GS_1 of the matrix G, every column in the matrix GS_i $(i \geq 1)$ must be pairwise distinct, which can be implemented by the *allDifferent* constraint (cf. Table 5.3). Therefore, all the possible values of columns (column space) of the matrix GS_i $(i \geq 1)$ is reduced from s^s to $s!$ by introducing the Constraint (5.8), which is a significant search space reduction.

In Section 5.1.3, we have presented Equation (5.1) to fix the first row of the matrix G. We can also fix the second row of the instances of the form s-s-(s+1) for the following reason. In Constraint (5.7), we have explained that s players assigned in the same group in the first week cannot meet again in the subsequent weeks. Besides, for the form s-s-(s+1), there are only s different groups, which implies that the possible groups assigned to these s players must be a permutation of the set of integers $\{1 \dots s\}$. Thus, every row of the submatrix GS_i $(i \geq 1)$ is a permutation of the set of integers $\{1 \dots s\}$. Moreover, arbitrary swapping two columns in the submatrix GS_i $(i \geq 1)$ leads to an isomorphism even when the first row of the matrix G is fixed by Equation (5.1). Therefore, for the instances of the form s-s-(s+1), we fix all the first rows of the submatrix GS_i $(i \geq 1)$ with the array $[1 \dots s]$ (cf. the second row of Table 5.3), which can be expressed as:

$$\forall_{j \in J} \; (G_{1,j} = j\%s + 1), \; J = \{x \in \mathbb{Z} | \; s \leq x < n\} \tag{5.9}$$

Thus, the symmetries caused by renumbering players in the second row can be eliminated by imposing Constraint (5.9).

In summary, the model used to tackle 5-5-6, 6-6-7, and 7-7-8 consists of the constraints of the basic model and the additional constraints including Constraint (5.8) and (5.9).

5.1.4.2 9-9-10

The additional constraints for 7-7-8 are insufficient to solve 9-9-10 within a reasonable time since the size of the problem grows significantly. One possible way to tackle the larger instance is to shrink the overall search space by imposing more instance-specific constraints.

We considered the solutions of the 4-4-5, 5-5-6, and 7-7-8 instance and discover that GS_1 can always be a symmetric matrix, namely $GS_1 = GS_1^T$. Hence, we conjecture that 9-9-10 can also have a symmetric submatrix and then impose the following constraints on the decision variables G:

$$\begin{gathered} \forall_{i \in I} \forall_{j \in J} \ (G_{i,j} = G_{(j-s),(i+s)}) \\ I = \{x \in \mathbb{Z} | \ 0 \leq x < w\}, \ J = \{x \in \mathbb{Z} | \ s \leq x < 2 * s\} \end{gathered} \tag{5.10}$$

Constraint (5.10) states that the entries of GS_1 are symmetric with respect to the main diagonal. Besides, the main diagonal of the submatrix GS_1 is pairwise distinct for 5-5-6 and 7-7-8, given by:

$$\begin{gathered} \forall_{i \in I} \forall_{j \in J} \ (G_{i,j} \neq G_{(i+1),(j+1)}) \\ I = \{x \in \mathbb{Z} | \ 0 \leq x < w\}, \ J = \{x \in \mathbb{Z} | \ s \leq x < 2 * s\} \end{gathered} \tag{5.11}$$

Apart from the fixed pattern of GS_1, there is also a fixed pattern among the submatrices of G. Because the second row has already been fixed by Constraint (5.9), we can impose the *allDifferent* constraints on the subsequent rows for those players who have played together in the second week since any two columns of G can only have identical values in exactly one row (e.g., *allDifferent*$(G_{3,5}, G_{3,10}, G_{3,15}, G_{3,20})$ in Table 5.3). These constraints are implied constraints and can be expressed as:

$$\begin{gathered} \forall_{i \in I} \forall_{(j \in J) \wedge (j' \in J) \wedge (j \% s = j' \% s) \wedge (j \neq j')} \ (G_{i,j} \neq G_{i,j'}) \\ I = \{x \in \mathbb{Z} | \ 1 \leq x < w\}, \ J = \{x \in \mathbb{Z} | \ s \leq x < n\} \end{gathered} \tag{5.12}$$

We also notice that for 5-5-6 and 7-7-8, there is always a type solution in which the second row of GS_1 is fixed by the array $[2, s, 1, 3, 4, \ldots, s-1]$ (cf. the second row of GS_1 Table 5.3). We, therefore, assume that for 9-9-10 also exists such solution and solve 9-9-10 by fixing the second row of GS_1 with $[2, 9, 1, 3, 4, 5, 6, 7, 8]$.

In conclusion, we solve 9-9-10 by adding Constraints (5.10), (5.11), and (5.12) to the model of 7-7-8, as well as the fixed values for the second row of GS_1.

5.1.4.3 13-13-14 etc.

We have discovered some common features of the instances of the form *s-s-(s+1)*, particularly for the instances of the form *o-o-(o+1)* when expressing a solution by groups; and these common features are mostly focused on the second submatrix GS_1 of G. It is also interesting to observe that the submatrix GS_i, $1 < i < s$, consists of s s-tuples that are derived from the second submatrix GS_1 on 5-5-6 and 7-7-9 but not from 9-9-10 (cf. Table 5.3). Simply put, the rest of submatrices can be obtained by interchanging rows of GS_1 on these instances. Thus, we expect to solve larger instances of the form *p-p-(p+1)* by restricting row space of the submatrix GS_i $(1 < i < s)$ to the rows of the submatrix GS_1. Formally:

$$PT = \{(G_{i,j}, G_{i,j+1}, ..., G_{i,j+s-1})|\ s \leq j < 2*s\ \wedge\ 2 \leq i < w\ \wedge\ i,j \in \mathbb{Z}\} \quad (5.13)$$

$$(G_{i,j}, G_{i,j+1}, ..., G_{i,j+s-1}) \in PT,\ 2 \leq i < w,\ 2*s \leq j < n,\ j\%s = 0,\ i,j \in \mathbb{Z} \quad (5.14)$$

where Constraint (5.13) defines the potential combination of values of columns of GS_1 as PT. Then we can limit the row space of the submatrices except GS_0 and GS_1 to PT by Constraint 5.14, which can be implemented by the *table* constraint. So the question then is, how to find the submatrix GS_1 that can lead to a solution of the instance (e.g., 13-13-14).

To find the correct GS_1, we create a separate model defined on an $s \times s$ matrix (s must be a prime number), which comprises Constraints (5.10) and (5.11), and the *allDifferent* constraint imposed on each row and each column of the matrix. We also fix the first row and the second row with $[1 \dots s]$ and $[2, s, 1, 3, 4, \dots, s-1]$ respectively, as we did for the 9-9-10 instance. Incidentally, GS_1 is a *Latin square* since it is a $s \times s$ matrix filled with s distinct numbers and every row and column of the matrix is all different. Moreover, for the last row ($i = s-1$) of GS_1, starting with the third element ($j = 2$) to the last element is fixed with the array $[2, 1, 3, 4, 5, \dots, s-2]$. Along with decrementing the row (i--), the element at the tail of the array is removed and the starting position of the first element of the array incrementing (j++) until the array is reduced to containing exactly one element $\{2\}$, as illustrated in Table 5.4.

Having this observed pattern and aforementioned separated model, we can obtain exactly one GS_1 for the instance of the form *p-p-(p+1)*, and then utilize it as an input for Constraint (5.14) with the model of 7-7-8 to solve 11-11-12 and 13-13-14. Note that since GS_1 has already initialized before solving process, we do not use the model of 9-9-10 because it is redundant to impose Constraints (5.10), (5.11), and (5.12) on the model.

1	2	3	4	5	6	7	8	9	10	11	12	13
2	13	1	3	4	5	6	7	8	9	10	11	12
3	1	4	5	6	7	8	9	10	11	12	13	**2**
4	3	5	6	7	8	9	10	11	12	13	**2**	**1**
5	4	6	7	8	9	10	11	12	13	**2**	**1**	**3**
6	5	7	8	9	10	11	12	13	**2**	**1**	**3**	**4**
7	6	8	9	10	11	12	13	**2**	**1**	**3**	**4**	**5**
8	7	9	10	11	12	13	**2**	**1**	**3**	**4**	**5**	**6**
9	8	10	11	12	13	**2**	**1**	**3**	**4**	**5**	**6**	**7**
10	9	11	12	13	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**
11	10	12	13	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
12	11	13	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
13	12	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Table 5.4: The second matrix GS_1 for the instance 13-13-14. (Table reproduced from [125, 124].)

5.1.4.4 8-8-9

So far, all the instances we have discussed conform to the form of *o*-*o*-(*o+1*). We now consider the form of instances *e*-*e*-(*e+1*). The 8-8-9 is solved by the following conjectures derived from 4-4-5 with the model for 7-7-8:

$$\forall_{i \in I}\ (G_{i,(i+s-1)} = 1),\ I = \{x \in \mathbb{Z} |\ 0 < x < w\} \tag{5.15}$$

$$\forall_{(j \neq j') \wedge (i \neq i') \wedge (j/s = j'/s) \wedge (j\%s+1=i) \wedge (j'\%s+1=i')}\ (G_{i,j} \neq G_{i',j'})$$
$$i, i' \in \{x \in \mathbb{Z} |\ 0 < x < w\},\ j, j' \in \{x \in \mathbb{Z} |\ 2 * s < x < n\} \tag{5.16}$$

Constraint (5.15) states that the main diagonal of the matrix GS_1 consists of the fixed values [1,1,...,1]; and the rest of submatrices have the main diagonal whose values must be pairwise distinct (Constraint (5.16)).

Table 5.5 depicts the submatrix GS_1 of the solution of 8-8-9 we solved. It is interesting to observe that the GS_1 matrix of 4-4-5 and 8-8-9 are composed of four symmetric matrices. Moreover, we discover that their solutions also satisfy the Constraints (5.13) and (5.14), and it is still unclear whether or not the 10-10-11, 12-12-13, and 16-16-17 instances share these common features with 4-4-5 and 8-8-9.[7]

[7] In Section 5.1.6.3, we will discuss the larger instances of the form *e*-*e*-(*e+1*), such as 10-10-11, 12-12-13, etc.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	1	4	**3**	**6**	5	8	**7**
3	4	1	**2**	**7**	8	5	**6**
4	**3**	**2**	**1**	**8**	**7**	**6**	**5**
5	**6**	**7**	**8**	**1**	**2**	**3**	**4**
6	5	8	**7**	**2**	1	4	**3**
7	8	5	**6**	**3**	4	1	**2**
8	**7**	**6**	**5**	**4**	**3**	**2**	**1**

Table 5.5: The second matrix GS_1 for a solution 8-8-9. (Table reproduced from [125, 124].)

5.1.5 Instances Solved in Parallel

In the previous section, we have presented the instances that can be solved sequentially via our modeling approach. We now turn to more difficult instances that must deal with by way of parallel processing to obtain one solution. The difficult instances (e.g., 7-3-10) refer to no fixed pattern discovered so far, which implies no instance-specific constraints to shrink search space for these instances and hence there are large search spaces even for relatively small size.

Our idea is to partition the search tree of the SGP into independent subtrees; then each worker that is associated with a thread works on distinct subtrees using the same CP model. Thus, this approach can be classified as *data-level parallelism* based on the taxonomy for parallelism in applications from [93]. Furthermore, since no communication is required during the solving process, to some extent, our parallel approach can also be seen as Embarrassingly Parallel Search (EPS) [170]. The original EPS approach is defined as decomposing the problem into many subproblems and assigning the subproblems to workers dynamically [155]. By contrast, our parallel approach differs from EPS due to the use of a separate model that is used to generate the subproblems instead of Depth-bounded Depth First Search [170]. The generic procedure can be summarized as follows:

1. A subset of the decision variables of the model is selected.

2. A separate model generates all the partial assignments over the selected variables in the subset before the search process.

3. The partial assignments are mapped to the workers so that each worker can work on its own independent search space by using its constraint solver.

4. Once a solution is found, the worker that finds the solution notifies the other workers to stop.

Step 1 is crucial to the search space splitting because it determines the subtrees explored by each worker. The selection of the subset of the decision variables adhere to the following rules: First, they should be easy to generate by a separate model. Second, each worker should not be assigned too many partial assignments because one partial assignment might take a long time to evaluate for a large instance. Because of the usage of the separate model, the partial assignments are consistent with the propagation (i.e., running the propagation mechanism on them does not detect any inconsistency). Besides, the number of solutions of the separate model can help us decide the workload of each worker and workload distribution. In the following sections, we will gradually describe CP models for generating partial assignments for search-space splitting and the constraints imposed on the basic model for the 6-3-8, 6-4-7, and 7-3-10 instances in detail.

5.1.5.1 6-3-8

The 6-3-8 instance is a representative example to illustrate the effectiveness of our parallel approach for the SGP since the instances smaller than it can be solved quickly and the instances bigger than it are difficult to be solved sequentially by constraint solving (cf. the experimental results given in Section 5.1.6). When switching the target instance from 5-3-7 to 6-3-8, the number of decision variables grows from $5*3*7 = 105$ to $6*3*8 = 144$, and the domain size of each variable is incremented by one for our modeling approach, which indicates the overall underlying search space significantly increased from 5^{105} to $6^{144} \approx 4.6e38$ if we do not take account of the search space pruned by constraint propagation.

The idea behind the parallel approach is to freeze a part of the decision variables so that the size of the subproblem is shrunk to solvable, thereby solving the original problem. For the 6-3-8 instance, we select the second row of the matrix G for the search space splitting since the first row of the matrix G is fixed by Constraint (5.1). A separate model is used to generate the solutions for the second row of the matrix G as the partial assignments for the search space splitting, which is composed of the following constraints:

$$\forall_{j \in J}\ (F_j = j+1),\ J = \{x \in \mathbb{Z} |\ 0 \leq x < s\} \tag{5.17}$$

$$GCC(F, V, O),\ V = [1, \ldots, g]$$

$$\forall_{k \in K}\ (o_k \in O = [s, s]),\ K = \{k \in \mathbb{Z} |\ 0 \leq k < g\} \tag{5.18}$$

$$\forall_{(j \neq j') \wedge (j/s = j'/s)} \ (F_j \neq F_{j'}), \ j' \in J \tag{5.19}$$

$$\forall_{j \% s = 0} \ (F_j \leq F_{(j+s)}) \tag{5.20}$$

$$\forall_{j/s = (j+1)/s} \ (F_j < F_{(j+1)}) \tag{5.21}$$

where F is an array of decision variables for the separate model used to generate the second row of the matrix G, and the domain of each variable is also $\{1 \dots g\}$. Constraints (5.17), (5.18), and (5.19) are identical to Constraints (5.1), (5.3), and (5.7) stated in the basic model (see Sec. 5.1.3) respectively. Constraints (5.20) and (5.21), which are not included in the basic model, are static symmetry breaking constraints. Constraint (5.20) removes the symmetries caused by interchangeable submatrices GS_i, $0 < i < s$. We eliminate these symmetries by arranging the values assigned to the first column of the first row of all the submatrices GS_i in non-decreasing order. Please refer to the numbers with superscript a in the second row of Table 5.6. Additionally,

Week \ Player	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	***1***	***1***	***1***	***2***	***2***	***2***	***3***	***3***	***3***	***4***	***4***	***4***	***5***	***5***	***5***	***6***	***6***	***6***
1	*1*^{a}	*2*	*3*	1^a	4	5	1^a	3	6	2^a	5	6	2^a	4	5	3^a	4	6
2	*1*	*2*	*3*	2^a	1	3	3	4	6	5	6	4	6	2	5	1	4	5
3	*1*	*2*	*3*	3^a	2	1	5	6	3	1	6	4	5	6	4	4	5	2
4	*1*	*2*	*3*	4^a	3	5	6	5	2	6	3	1	1	6	4	2	4	5
5	*1*	*2*	*3*	4^a	5	2	6	4	5	3	6	2	4	1	5	6	3	1
6	*1*	*2*	*3*	4^a	5	6	2	5	1	5	4	3	6	3	2	6	1	4
7	*1*	*2*	*3*	4^a	5	6	5	1	6	4	2	5	3	6	1	4	2	3

Table 5.6: A solution of 6-3-8 expressed by groups. (Table adapted from [125, 124].)

interchanging any two columns of a submatrix GS_i generates a solution symmetrical with the original one, which entails Constraint (5.21) to remove these symmetries. Because of Constraint (5.21), the players played together in the first week must be in ascending order of groups in the second week (cf. the second row of Table 5.6).

In addition to the constraints of the separate model, we also place the constraints to break the symmetries caused by interchangeable weeks partially. The idea is to restrict the groups of the 4th player in non-decreasing order from week two, given by:

$$\forall_{i \in I} \ (G_{i,s} \leq G_{(i+1),s}), \ I = \{x \in \mathbb{Z} | \ 0 < x < w - 1\} \tag{5.22}$$

Please note that Constraint (5.22) cannot fully remove the symmetries among weeks because there are still symmetries whenever $G_{i,s} = G_{(i+1),s}$. For example in Table 5.6, interchanging the 6th week with 7th week results in a symmetrical solution.

Finally, 424 solutions for the second row of the matrix G generated by the above model are equally distributed to each worker that runs the basic model.

5.1.5.2 6-4-7

The separate model described in the previous section produces 351 solutions for the second row of 6-4-7, compared to 424 for 6-3-8. However, 6-4-7 is much harder than 6-3-8 due to increased search space ($\frac{6^{168}}{6^{144}} = 6^{24} \approx 4.7e18$). Thus, we add the following constraints based on the separate model for 6-3-8 to produce less number of solutions for the second row of 6-4-7:

$$\forall_{j \in J} \left(F_{j+1} \leq F_{j+s+1}\right)$$
$$J = \left\{ j \in \mathbb{Z} |\ 0 \leq j < n - s \wedge j\%s = 0 \wedge j = (s-1) * s \Rightarrow j + s \neq s^2 \right\} \qquad (5.23)$$

$$\forall_{j \in J} \left(F_{j+1} = F_{j+s+1} \Rightarrow F_{j+2} \leq F_{j+s+2}\right) \qquad (5.24)$$

$$\forall_{j \in J} \left(F_{j+1} = F_{j+s+1} \wedge F_{j+2} = F_{j+s+2} \Rightarrow F_{j+3} \leq F_{j+s+3}\right) \qquad (5.25)$$

In short, Constraints (5.23)-(5.25) ensure that the values occupying the same positions in the first row of the first s submatrices (GS_0, GS_1, GS_2, GS_3) and the last two submatrices (GS_4, GS_5) are in non-decreasing order respectively (see Table 5.7). The reason why the submatrices are divided into two groups is that numeral 1 always takes up the first row of the first column in the first s submatrices due to the restrictions from Constraint (5.20) and (5.21). Thus, if a constraint enforces $G_{1,13} \leq G_{1,17}$, the solution shown in Table 5.7 will not be obtained. These additional constraints also reduce search space by removing symmetries. For example, if we do not impose Constraint (5.23) on the separate model, a second row such as [1 2 3 4 1 4 5 6 1 2 3 4 1 4 5 6 2 3 5 6 2 3 5 6] will be generated. In that case, we will require more workers to work on these symmetrical search spaces.

Week \ Player	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	***1***	***1***	***1***	***1***	***2***	***2***	***2***	***2***	***3***	***3***	***3***	***3***	***4***	***4***	***4***	***4***	***5***	***5***	***5***	***5***	***6***	***6***	***6***	***6***
1	1^{a}	2^{b}	3^{c}	4^{d}	1^{a}	2^{b}	3^{c}	4^{d}	1^{a}	4^{b}	5^{c}	6^{d}	1^{a}	4^{b}	5^{c}	6^{d}	2^{e}	3^{f}	5^{g}	6^{h}	2^{e}	3^{f}	5^{g}	6^{h}
2	*1*	*2*	*3*	*4*	2	4	6	5	5	3	6	1	6	2	4	5	3	4	1	2	5	1	3	6
3	*1*	*2*	*3*	*4*	3	6	4	5	4	2	1	3	6	1	5	2	5	1	2	6	3	6	4	5
4	*1*	*2*	*3*	*4*	4	6	1	2	3	5	2	6	5	6	3	1	1	5	4	3	5	2	6	4
5	*1*	*2*	*3*	*4*	6	3	5	1	2	6	3	4	4	5	6	3	4	2	5	1	5	6	1	2
6	*1*	*2*	*3*	*4*	6	1	2	3	5	1	4	2	3	5	2	6	5	6	3	4	4	5	6	1

Table 5.7: A solution of 6-4-7. The numbers with the same superscript are in non-decreasing order in the second row. (Table adapted from [125, 124].)

Having executed the separate model for obtaining the possible second row, we can evenly distribute the 48 solutions of it to different workers before solving process. Then, to solve the 6-4-7 instance, we further reduce the search space by adding

the following constraints onto the basic model:

$$\begin{aligned} &\forall_{j\in J}\ (GCC(G_{*,j}, V, O)) \\ J = \{j \in \mathbb{Z}|\ s \le j < 3s\}\ ,\ &V = \{v \in \mathbb{Z}|\ 1 \le v \le s\} \\ O = \{(o_0, o_1, \ldots, o_i)|i < s, &\forall_{o_i} = [1, 1]\}\ ,\ 0 < * < w \end{aligned} \tag{5.26}$$

where $G_{*,j}$ denotes the columns from the sth column to the *(3s-1)*$^{\text{th}}$ column of the matrix G with removed first element. More particularly, every value in the set $\{1, 2, 3, 4\}$ can appear only once in all the columns of the submatrices GS_1 and GS_2. We impose Constraint (5.26) on only the columns of GS_1 and GS_2 because each player only plays with other 21 players since $(24-1)\%(4-1) = 2$; thus not every column contains the set $\{1, 2, 3, 4\}$. Though Constraint (5.26) does not enforce all columns containing the values {1,2,3,4}, it reduces much search space; our experiments show that we cannot solve 6-4-7 without these constraints.

5.1.5.3 7-3-10

The 7-3-10 instance is much more difficult than 6-4-7 and 6-3-8 due to more variables and larger domain size, we must harness more instance-specific constraints, which are given by:

$$\forall_{i\in I}\ (G_{i,s} = i+1),\ I = \{i \in \mathbb{Z}|\ 0 < i \le s\} \tag{5.27}$$

$$\forall_{i\in I'}\ (G_{i,s} = s+1),\ I' = \{i \in \mathbb{Z}|\ s+1 < i < 2\} \tag{5.28}$$

$$\begin{aligned} GCC(G_{*,(s+1)}, V, O),\ 0 < * < w,\ V = [1, 2, 3, 6, 7],\ OCC = [1, 1, 1, 0, 0] \\ \forall_{j\in J}\ (o_j \in O = [OCC_j, OCC_j]),\ J = \{j \in \mathbb{Z}|\ 0 \le j < 5 = |V|\} \end{aligned} \tag{5.29}$$

$$\begin{aligned} GCC(G_{*,(s+2)}, V', O'),\ V' = [1, 2, 3, 6],\ OCC' = [1, 1, 1, 0] \\ \forall_{j'\in J'}\ (o'_{j'} \in O' = [OCC'_{j'}, OCC'_{j'}]),\ J' = \{j' \in \mathbb{Z}|\ 0 \le j' < 4 = |V'|\} \end{aligned} \tag{5.30}$$

$$\begin{aligned} \forall_{s+s\le j<n}\ (GCC(G_{*,j}, V'', O'')),\ V'' = [1, \ldots, s] \\ \forall_{j''\in J''}\ (o''_{j''} \in O'' = [1, 1]),\ J'' = \{j'' \in \mathbb{Z}|\ 0 \le j < s = |V''|\} \end{aligned} \tag{5.31}$$

We strictly limit the positions of player 3 so that he/she will never be assigned to groups 5, 6, and 7. The reason is that player 3 must meet players 0, 1, and 2 in the first week, and the groups of the first three (s) players are frozen by Equation 5.2 after the first week, which implies that player 3 must stay in the first three groups from week 2 to week 4. Hence, player 3 can only play in the groups that are greater than or equal to 4 (s+1) from week 4. Moreover, player 3 is always the smallest player number starting from the 9$^{\text{th}}$ column of a solution (cf. Table 5.1). Therefore,

player 3 cannot appear in groups 5, 6, and 7, and only stay in group 4 from week 4. Consequently, the 4th column of Table 5.2 is the result by imposing Constraint (5.27) and (5.28). These two constraints not only shrink the search space but also remove the symmetries caused by swapping the group containing player 3 with other groups after week 3.

Since player 3 can only play in group 4 after week 3, player 4 is impossible to stay in groups 6 and 7, because then there will be no player assigned in group 5. Similarly, player 5 cannot appear in group 7 and can only appear in group 6 once. Thus, we use Constraints (5.29) and (5.30) to limit the number of occurrences of the values 6 and 7.

Furthermore, because (21-1)%(3-1) = 0, each player must play with other players exactly once. Hence, we guarantee the first s players must meet the rest of players once, which are ensured by Constraints (5.29), (5.30), and (5.31). Incidentally, Constraint (5.31) can be applied to any full instance that satisfies $(n\text{-}1)\%(s\text{-}1) = 0$ in our modeling approach (e.g., 7-4-9).

In the implementation of parallelism for 7-3-10, we also use the same separate model as the model for 6-4-7 and we can generate 5,953 solutions of the second row.

5.1.6 Experiments

In this section, we report the experimental results on instances discussed in Section 5.1.4 and Section 5.1.5 separately since different hardware were used.

5.1.6.1 Experimental Results on Instance Solved Sequentially

To confirm our theoretical discussion and the conjecture for the instances discussed in Section 5.1.4, we implemented the basic model as described in Section 5.1.3 and the instance-specific constraints in Section 5.1.4 via the Choco Solver 4.0.6 [165] with JDK version 10.0.1. All experiments were performed on a laptop with an Intel i7-3720QM CPU, 2.60GHz with 4 physical and 8 logical cores, and 8 GB DDR3 memory running Linux Mint 18.3.

Table 5.8 summarizes the experimental results on the instances solved sequentially, including the total CPU time, the number of visited nodes, backtracks, and fails. It also provides search strategies we used. By using our approach, we were able to prove the nonexistence of the solution of 6-6-7 and solved six open instances for constraint satisfaction approach but not for metaheuristic approach [46].

Instance	Time(s)	Nodes	Backtracks	Fails	Strategy
5-3-7	0.095	111	179	94	dom
5-5-6	0.069	7	1	0	min
6-6-7[c]	25	1.38e5	2.77e5	1.38e5	min
7-7-8[c]	111	3.62e5	723e5	3.62e5	min
8-8-9[c]	12	15,370	30,680	15,350	min
9-9-10[c]	2559	2.08e6	4.16e6	2.08e6	min
11-11-12[c]	62	3,150	6,279	3,144	min
13-13-14[c]	2563	5.80e4	1.16e5	5.79e4	min

Table 5.8: Results on the *s-s-(s+1)* Instances. A superscript "c" means that the instance was open for the constraint satisfaction approach; "dom" and "min" denote the predefined search strategies domOverWDegSearch and minDomLBSearch in Choco Solver, respectively. (Table reproduced from [125, 124].)

5.1.6.2 Experimental Results on Instance Solved in Parallel

To validate our parallel approach for the SGP, we switch to a computer with 250 GB DDR3 1066 memory and 4 Intel Xeon CPU E7-4830 2.13GHz processors running on Linux CentOS 6.5, where each processor has 8 physical cores. The versions of Choco Solver and the JDK are unchanged. Table 5.9 reports the experimental results

Instance	Workers	Time(s)	Nodes	Backtracks	Fails	Strategy
6-3-8[c]	1	2.95e4	2.91e8	5.83e8	2.91e8	min
	8	50.2	2.09e5	4.18e5	2.09e5	min
	16	2.62e4	2.50e8	5.13e8	2.31e8	min
6-4-7[f]	1	-	-	-	-	min
	48	8.59e3	1.66e7	3.32e7	1.66e7	min
7-3-10[f]	1	-	-	-	-	dom
	32	7.61e4	1.86e8	3.73e8	1.86e8	dom

Table 5.9: Results on the Instances solved in parallel. A superscript "f" means that the instance is solved by computer for the first time. A "-" sign means the program was still running after a period which is equal to the number of workers multiplied by the execution time in parallel. (Table adapted from [125, 124].)

for comparing parallel and sequential execution when using the same model to solve the same instance. For parallel execution, the number of workers we used varies from instance to instance. For 6-3-8, we specified 8, 16 and 32 workers to execute in parallel, but super-liner speedup was only observed when using 8 workers, because the partial assignment that can lead to a solution does not happen to be evaluated first.

Then, for 6-4-7, we used 48 workers because there are only 48 solutions generated by the separate model. Finally, the result of 7-3-10 is given by selecting the first 8

solutions of the separate model, and every solution is allocated to 4 different workers, each of which employs their respective search strategies that are predefined in Choco Solver, including *minDomUBSearch*, *minDomLBSearch*, *defaultSearch* and *domOverWDegSearch*. Besides, we also performed three more experiments in which the separate model was specified with above mentioned search strategies. As a consequence, the first 8 solutions are different from the first experiment, and we obtained three more non-isomorphic solutions for the 7-3-10 instance. The solutions of the instances in Table 5.9, which are not given in the main body of this thesis, are provided in Appendix A.

5.1.6.3 Discussion

It is interesting to observe the results for the instances of the form s-s-(s+1) ($s = \{5, 7, 8, 11, 13\}$) consisting of s-1 mutually orthogonal $s \times s$ Latin squares (cf. GS_1, GS_2, GS_3, and GS_4 of Table 5.3).[8] The results of these instances are consistent with the basic correspondence of *affine planes* and Latin squares, which proves that there exist n-1 mutually orthogonal Latin squares (MOLS) of order n iff there exists an affine plane of order n [157, 12], i.e., there are affine planes of order 5, 7, 8, 11, and 13. It is also not difficult to relate no solution for 6-6-7 to no MOLS of order 6 [14]. And we argue that the solution of 10-10-11 is nonexistent because there is no set of 7 or more MOLS of order 10 [135] and thereby no affine plane of order 10 [111]. More generally, we speculate that the solutions for the form np-np-(np+1) (e.g., np = 14, 21, 22, 30, 33) do not exist because of the nonexistence of *projective planes*[9] for them according to the *Bruck-Ryser-Chowla theorem* [12], where $np \equiv 1$ *or* 2 (mod 4) and the square-free part of np contain at least one prime $p \equiv 3$ (mod 4).[10] Moreover, the 12-12-13 instance is hard for the CP approach, which corresponds to searching an affine plane of order 12 — an unsettled case [3].

In addition to the results of the instances, we also show that more instance-specific constraints can shorten the execution time even if the size of instances increases. For example, 11-11-12 took much less time than 9-9-10 since more constraints are posted. The experimental results also show that parallel constraint solving through search space splitting is a very effective means to prevent backtrack search from getting stuck

[8] Two Latin squares are mutually orthogonal if, they have the same order n and when superimposed, each of the possible n^2 ordered pairs occur exactly once. Leonhard Euler (1707-1783) famously conjectured that there does not exist two orthogonal Latin squares for any oddly even number $n \equiv 2$ (mod 4). However, in 1959, Bose & Shrikhande disproved Euler's conjecture by showing that there exist at least two orthogonal Latin squares for all $n > 6$ [24, 23].

[9] An affine plane of order n exists iff a projective plane of order n exists.

[10] For instance, $14 = 2 * 7 \equiv 2$ (mod 4), and the primes in the square-free part are 2 and 7.

into a fruitless search area. Without surprise, the superlinear speedup was observed since only one invalid partial assignment is enough to cause instances such as 6-4-7 to be unsolvable for sequential solving and one valid partial solution can easily lead to backtrack search into a search area with a solution. Note that observed superlinear speedups are not in contradiction with Amdahl's law since our goal is to obtain a first solution instead of all solutions.

5.1.7 Related Work

There is a substantial body of work available on symmetry breaking for the SGP from the constraints community, including model reformulation, static symmetry breaking constraints, and dynamic symmetry breaking.

5.1.7.1 Methods from the CSP Literature

Smith [193] presented the integer set model with extra auxiliary variables that automatically eliminates the symmetries inside of groups, which is probably one of the first works that break the symmetry of the SGP via model reformulation. Besides, symmetry breaking during search (SBDS) with symmetry breaking constraints is employed to break renumbering symmetry but not entirely, where SBDS is essentially a search space reduction technique that adds constraints to remove symmetrical search space during search. Law & Lee [113] developed the *Precedence* constraint to break the symmetries of groups inside of weeks for the integer model and the symmetries caused by renumbering players for the set model. Symmetry breaking via dominance detection (SBDD), another dynamic symmetry breaking technique, was developed separately by Focaci & Milano [53] and by Fahle *et al.* [49, 63]. The main idea of SBDD is to utilize nogood learning to avoid exploring search space that is symmetrical of previously explored nodes recorded on the nogoods. By using SBDD, Fahle & Milano discovered seven non-symmetric solutions for the 5-3-7 instance in less than two hours on a computer with an UltraSparc-slowromancapii@ 400 MHz processor.

Barnier & Brisset [13] proposed SBDD+ for the SGP, which computes isomorphisms not only for leaves of the search tree but also on current non-leaves nodes. The experimental results showed that SBDD+ only took around eight seconds to compute all the seven non-symmetric solutions for 5-3-7, which is a significant improvement compared with [49]. However, they also pointed out that SBDD+ has to tackle the explosion of node store and the time overhead due to nodes dominance checking for a larger instance. Puget [166] combined SBDD with symmetry breaking using stabilizers (STAB) to obtain a solution of 5-5-6 in 38 seconds on a laptop with a Pentium M

1.4 GHz processor, where STAB is a variant of SBDS that adds symmetry breaking constraints without changing specified partial assignment.

All of the above mentioned works aim at eliminating the symmetries of the SGP, which is the first difficulty mentioned in Section 5.1.2.1. To tackle the second difficulty, Sellmann & Harvey [187] developed the vertical constraints and horizontal constraints for propagation, which can check whether a given partial assignment is extensible to a solution. They obtained all unique solutions of the 5-3-7 instance in 393.96 seconds on a computer with Pentium slowromancapiii@ 933 MHz processor by using the dedicated constraints. However, the dedicated constraints are developed for the original naive model, and no efficient algorithm for finding the players who have conflicting residual graphs is given.

5.1.7.2 Methods from the Metaheuristic Literature

Despite having elegant and sophisticated search space reduction techniques such as SBDS, SBDD, etc., the constraint satisfaction approach, a systematic search method, cannot compete with the metaheuristic approaches on the SGP when the goal is to obtain one solution instead of all non-symmetric solutions. Dotú & Van Hentenryck [46] employed tabu search with a constructive seeding heuristic and good starting points to achieve significant results on the instances of the form p-p-$(p+1)$ (e.g. 43-43-44, 47-47-48). Dotú & Van Hentenryck also solved 9-9-10 and 6-3-8 by using tabu search with a good starting point in 0.01 second and 51.93 seconds on a computer with Pentium slowromancapiv@ 3.06 GHz processor [47]. Besides, the 6-3-8 instance was also solved by the evolutionary approach on a Pentium slowromancapiv@ 3.06 GHz processor [36]. Unfortunately, the total CPU time is not reported in [47].

Triska & Musliu [197] are the first to solve the 8-4-10 instance reported in the literature, although one solution of 8-4-10 had already been published before [1] but without any explanation. The idea behind their metaheuristic approach is to employ a greedy heuristic for tabu search with the well-designed greedy initial configuration. The first solution of 8-4-10 instance was obtained in 11 minutes on a computer with an Intel Core 2 Duo 2.16 processor. Moreover, after varying the randomization factor of the greedy heuristic, they obtained two new non-isomorphic solutions for 8-4-10. In addition to the metaheuristic approach, they also explored a SAT encoding for the SGP [198]. Unfortunately, their SAT encoding is not competitive with other approaches.

Generally, solving the q-q-$(w+2)$ instance of the SGP amounts to finding w MOLSs. In addition to these approaches mentioned above that address the SGP head-on, Har-

vey & Winterer [90] exploited MOLS (in practice, mutually orthogonal Latin rectangles (MOLR)) solutions found to construct solutions to the SGP. The most notable instance they solved is 20-16-6, which indicates that this is probably the most efficient method so far. However, no full instance g-s-w^* was resolved since this method heavily relies on the construction of MOLR.

5.1.7.3 Summary

Most of the research from the constraints community focus on search space reduction techniques, mainly dynamic symmetry breaking. The metaheuristic approach, by contrast, aims at finding a first solution as quickly as possible. For example, the 6-3-8 instance could be solved within reasonable time via the metaheuristic approach but not the constraint satisfaction approach. Note that the problem grows much faster even from 5-3-7 to 6-3-8 than the performance boost out of the processors. Table 5.10 summarizes the main accomplishments in the SPG from the computer-science community, including both the constraints and metaheuristics communities.

Instance	Year	Authors	Method	Description
4-3-4	2001	Smith [193]	SBDS	42 solutions found
5-3-7	2001	Fahle & Milnano [49]	SBDD	7 unique solutions in 2 h
5-3-7	2001	Barnier & Brisset [13]	SBDD+	7 unique solutions in 8 s
5-3-7	2002	Sellmann &Harvey [187]	Specific Constraints	7 unique solutions in 394 s
5-5-6	2005	Puget [166]	SBDD & STAB	a solution in 38 s
20-16-6	2005	Harvey & Winterer [90]	MOLR	tabu search for MOLR
47-47-48	2005	Dotú *et al.* [46]	Tabu-search	efficient for p-p-$(p+1)$
6-3-8	2007	Dotú *et al.* [47]	Tabu-search	a solution in 52 s
8-4-10	2011	Triska & Musliu [197]	Tabu-search	2 new unique solutions found

Table 5.10: The summary of the most significant results on the SGP from the computer-science community. (Table adapted from [124].)

Finally, some instances that have not been solved by computer at present have already been constructed by combinatorics (e.g., 7-4-9, 9-3-13). For a detailed introduction, please refer to [39, 167].

5.1.8 Conclusion

In this section, we have presented a combination of techniques which allows us to find solutions for nine open instances, where six of these instances are solved sequentially,

and three of these instances are solved in parallel. In particular, we have shown that the constraints derived from the relatively small instances can be used to solve larger instances that are in the same form as the smaller ones. In other words, we explore the properties of the instances of the form s-s-(s+1) from the perspective of constraint programming. Besides, we have also shown that it is not uncommon for solving the SGP in parallel via search space splitting to gain superlinear speedups and parallel solving the SGP can be an effective method to address the instances that cannot be solved sequentially. The results show that our method is much more successful, even if we consider that the computers used for the other methods are up to 10 times slower than ours.

Unlike the earlier researches on the SGP which mainly focus on dynamic symmetry breaking, we attribute the success of our approach to the effectiveness of the instance-specific constraints and parallelism due to mitigating the two problems of solving the SGP mentioned in Section 5.1.2.1. Specifically, the instance-specific constraints imposed on the second submatrix of the decision variables matrix prune a large number of the sub-search trees near the root, including some symmetries. And since many partial assignments are extended simultaneously, fruitless partial assignments have no impact on overall execution time. Not only that, but search space splitting can result in the partial assignments that can lead to a solution to be proceeded much earlier than the sequential search, which is the reason for superlinear speedup. Furthermore, we can conclude that early diversity brought by search space splitting before search can effectively alleviate the strong commitment due to the early decisions made by search strategy. Besides, we also remove the symmetries of the second row in the decision variables matrix when generating the partial assignments, which is helpful because nodes near the root contain much more symmetries than the nodes near the leaves of the search tree [166]. Therefore, with mainstream computers turning into parallel architectures, we believe that parallel constraint solving through search space splitting is a promising approach to solving more significant instances of the SGP.

Indeed, there is still a lot of potential to improve the performance of our approach. In particular, Constraint (5.22) is unable to eliminate the symmetries among weeks after week s when solving 6-3-8, 6-4-7, and 7-3-10. In fact, we can resolve it by enforcing the indices of the second "1" of all the weeks in ascending order, which means that the second golfer assigned in the first group are in ascending order. Unfortunately the performance is not satisfactory due to the use of the *IfThen* constraints or the *Reified* constraints. Besides, Constraints (5.5) and (5.6) introduce too many auxiliary

variables that inevitably slow down the resolution process; thus, we have also implemented a specialized constraint to replace them. However, our constraint increases the difficulty of variable-selection since the constraint requires an additional variable to record the equality relationship among rows of the matrix G. To solve larger instances, in addition to using more processors and discovering more instance-specific constraints, we would like to consider combining the dynamic symmetry breaking and parallel constraint solving for the SGP.

In the end, we must regretfully admit that even if we have made some progress, some interesting instances are are still open (e.g. 7-4-9, 8-3-11, and 9-3-13); notably, the original SGP 8-4-10 [89] is still unsolved for the CP approach, despite many efforts from the constraint programming community. Constraint technology should solve these instances to demonstrate itself as the first choice for solving combinatorial problems.

5.2 Traveling Tournament Problem with Predefined Venues

The Traveling Tournament Problem with Predefined Venues (TTPPV) is a practical problem arising from sports scheduling. In this section, we describe two different modeling approaches for this problem, each of which is suitable for different sizes of instance. Besides, we present the details of how to exploit the EPS approach to discover a feasible solution with much better objective value.

This section is a revised version of our previous work [122] and is organized as follows: Section 5.2.1 gives a short introduction to the TTPPV, Section 5.2.2 gradually describes the first CP model and its parallelization with empirical results. Afterwards, in Section 5.2.3, we present the second model, how to run this model in parallel and the experimental results by comparing models. Next, we discuss and analyze the experimental results in Section 5.2.4. Finally, we conclude in Section 5.2.5.

5.2.1 Introduction to the TTPPV

The Traveling Tournament Problem with Predefined Venues (TTPPV), i.e., problem 068 in CSPLib [160], was originally presented in [137] and seeks a compact single round-robin schedule for a sports tournament that minimizes the total distance traveled by all teams participating in the tournament. The Traveling Tournament Problem (TTP) and the TTPPV have been studied in the constraint programming and integer programming communities [107], where TTPPV is a special case of TTP by adding

predefined venues for each particular game. The predefined venues denote that the home-away assignment of each game is known beforehand. Specifically, i.e., team A plays against team B at team A's home or B's home is already determined before the scheduling. The problem of scheduling TTP usually consists of two subproblems, the construction of the timetable, which schedules that each team plays against other teams in which round, and the home-away pattern (HAP) table that determines home and away games for each team in each round. Hence, a complete scheduling of TTP and TTPPV is composed of the timetable and the HAP. Moreover, since home-away assignment rules out some HAPs that are incompatible with predefined venues, the overall search space of TTPPV is much smaller than TTP for the same number of teams.

Table 5.11 depicts a feasible solution of the TTPPV problem for eight teams, consisting of a timetable and a HAP table. The timetable in Table 5.11 is an 8×8 square matrix in which each row denotes a schedule for a team (cf. the square matrix beneath the horizontal line of Table 5.11). For instance, the entry in the 4th row and 3th column of the matrix stands for team 4 must play against team 7 in round 3. Meanwhile, this game should also be indicated on the entry in the 7th row and 3th column of the matrix, i.e., team 7 must play against team 4 in round 3. A HAP table in the TTPPV problem is a (0,1) matrix in which every entry is either 0 or 1. A home game and an away game are denoted by 1 and 0. The HAP table of the feasible solution, namely an 8×7 (0,1) matrix, is superimposed on a submatrix of the timetable consisting of the columns from the second column to the last column.[11] For convenience, we let a "$*$" prefix denote an away game in Table 5.11, and thus no prefix means a home game. For the example mentioned previously, team 4 is playing away against team 7 in the third round, which implies that team 7 is playing home against team 4 in the third round.

As already mentioned, the predefined venues consisting of home-away assignments for a tournament have been determined before scheduling, which can also be expressed as a (0,1) matrix. As an example, the home-away assignments for the problem described above are defined in Table 5.12, the values *1*, *0*, and *-1* of which denote a home game, an away game, and no game, respectively.

We now formally define the problem. For a tournament with n teams, the timetable in a feasible solution of TTPPV is an $n\times n$ matrix T with a fixed first column

[11] In the given example, the first and the second column are [1,2,3,4,5,6,7,8] and [7,4,5,2,3,8,1,6], respectively.

Team \ Round	1	2	3	4	5	6	7
1	7	*6	*5	4	3	*2	*8
2	*4	3	*8	*6	5	1	*7
3	*5	*2	6	8	*1	*7	4
4	2	5	*7	*1	8	6	*3
5	3	*4	1	7	*2	*8	6
6	8	1	*3	2	7	*4	*5
7	*1	*8	4	*5	*6	3	2
8	*6	7	2	*3	*4	5	1

Table 5.11: A feasible solution for a TTPPV problem with 8 teams.

Team	1	2	3	4	5	6	7	8
1	-1	0	1	1	0	0	1	0
2	1	-1	1	0	1	0	0	0
3	0	0	-1	1	0	1	0	1
4	0	1	0	-1	1	1	0	1
5	1	0	1	0	-1	1	1	0
6	1	1	0	0	0	-1	1	1
7	0	1	1	1	0	0	-1	0
8	1	1	0	0	1	0	1	-1

Table 5.12: The predefined venue data for the problem shown in Table 5.11.

$[1, \ldots, n]$. An element of the matrix T must satisfy the following property[12]:

$$\forall_{i,j,k \in \mathbb{Z}},\ 1 \leq i \leq n,\ 2 \leq j \leq n,\ 1 \leq k \leq n,\ T_{ij} = k \Longleftrightarrow T_{kj} = i \qquad (5.32)$$

where T_{ij} is an opponent variable which means that team i has to play against the team assigned to variable T_{ij} in round j. Index j starts at 2 because the first column ($j = 1$) denotes teams $[1, \ldots, n]$. Property (5.32) states that if the opponent of i in round j is k then the opponent of k in round j must be i. Furthermore, each row of T takes on distinct values from $\{1, \ldots, n\}$ because no team would play against another team more than once, given by:

$$\forall_{i,j,j' \in \mathbb{Z}},\ 1 \leq i \leq n,\ 1 \leq j < j' \leq n,\ T_{ij} \neq T_{ij'} \qquad (5.33)$$

We denote the HAP table of a timetable as an $n \times (n-1)$ matrix H. An element of HAP is denoted by H_{ij}, the matrix H has the following property:

$$\forall_{i,j' \in \mathbb{Z}},\ 1 \leq i \leq n,\ 1 \leq j' \leq n-1,\ H_{ij'} \in \{0, 1\},\ H_{ij'} \oplus H_{T_{i(j'+1)}j'} = 1 \qquad (5.34)$$

[12]For brevity's sake, the index of an array starts from 1 in Section 5.2.

The above property means that if team i has to play against team $T_{i(j'+1)}$ away in round $j'+1$, then team $T_{i(j'+1)}$ is playing against team i at home in the same round, and vice versa. Please note that in a feasible solution the HAP table must overlap the timetable by starting at its second column; therefore, the number of columns of HAP is $n-1$. The HAP table of a feasible solution must meet two conditions, given by the original problem description in [137]: First, the number of home games and the number of away games must be equal or differ by one for each team, which is the *balance condition* for TTPPV. Second, the *consecutive condition* is that the number of consecutive away games or home games for each team cannot exceed a fixed value.

The traveling distance of a team is calculated in the following way: First, for a single away game, the team travels to and from the venue of the opponent. Second, for a sequence of consecutive away games, the team travels from the venue of one opponent to that of the next, without returning home. Following [160], the distance between any two team i and j is defined as:[13]

$$\forall_{i,j \in \mathbb{Z}},\ i \geq j,\ d_{ij} = d_{ji} = min(i-j, j-i+n) \tag{5.35}$$

5.2.2 Modeling the TTPPV Based on Perfect Matching (The First Model)

In this section, we present and compare our first model with another model from the literature [161] in an empirical approach. A CP model for TTPPV problem can be directly derived from its problem definition as we have stated in the introduction. Therefore, in [161, 212], Properties (5.32) and (5.34) are guaranteed by the *element(v,T,i)* constraint, which ensures that value v is assigned to the ith variable in an array of variables T. Because the value of T_{ij} cannot be determined when modeling, one must employ the *ifThen* constraint or the *reified* constraint combined with the *element* constraint to express Property (5.32). Similarly, to ensure Property (5.34), the *element* constraint and the *reified* constraint may be used together again since Property (5.34) depends on Property (5.32). However, our observations on Choco solver show that these constraints are likely to slow down the resolution process of CSPs. Thus, we present an alternative modeling approach to avoid using these constraints.

[13]Note that we could use a matrix that predefines the real distances between teams. Equation (5.35) is just a possible way to define the distances between teams and has no particular meaning.

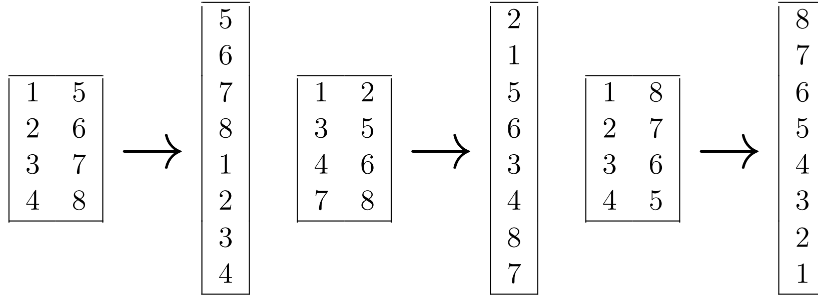


Figure 5.1: A conversion from solutions generated by the model to the potential values of columns of T, the number of teams is 8. (Figure reproduced from [122].)

5.2.2.1 A Model for Perfect Matching

In order to avoid using the *ifThen* constraint and the *element* constraint, we use another model to generate the potential combination of values of columns of the matrix T. The idea behind the model is to generate all possible matches of the n teams directly. Then, we convert the results of the model to possible columns of the matrix T. The variables of this model are defined as an $\frac{n}{2}\times 2$ matrix P in which each row denotes a match between two teams. The model can be expressed as:

$$\forall_{i,i',j,j'\in\mathbb{Z}},\ 1 \le i \le i' \le \frac{n}{2},\ 1 \le j \le j' \le 2$$

$$\forall_{(i\neq i')\vee(j\neq j')},\ P_{ij} \neq P_{i'j'} \tag{5.36}$$

$$\forall_{((i<i')\wedge(j=j'=1))\vee((i=i')\wedge(j<j'))},\ P_{ij} < P_{i'j'} \tag{5.37}$$

Constraint (5.36) guarantees all elements in P are pairwise distinct, which can be implemented by the *allDifferent* constraint. Then, Constraint (5.37) ensures that both the first column and each row of P must be in ascending order, which can be enforced by the *arithm* constraint. With 8 teams (i.e., $n = 8$) we exemplarily show 3 solutions generated by this model depicted on the left side of every arrow in Figure 5.1 (the overall number of solutions generated by this model for 8 teams is 105).

The above model generates all possible games ensured by Constraint (5.36), and rules out all the isomorphisms realized through Constraint (5.37), which is a *static symmetry breaking constraint*. If we treat every row of a solution of the model as an edge of a graph and the two values in each row as two vertices, each solution of the model can be viewed as a perfect matching for the complete graph with n vertices. Thus, the number of solutions of the model is equal to the number of perfect matchings for a complete graph with n vertices, which can be calculated $N_p = \Pi_{k=1}^{n/2}\binom{2k}{2}/(n/2!)$, where n must be even. Nevertheless, the solutions of the model

cannot be used directly as the potential values of columns of T. Therefore, we must convert each solution of the matrix P for the model to an array A with length n, as shown in Figure 5.1. The transformation rule for a solution of the model to the array can be stated as:

$$\forall_{i\in\mathbb{Z}},\ 1 \leq i \leq \frac{n}{2},\ A_{P_{i1}} = P_{i2},\ A_{P_{i2}} = P_{i1} \tag{5.38}$$

where A is the array starting at 1 to n. For each solution of the model, the transformation assigns P_{i2} (P_{i1}) to the element with index P_{i1} (P_{i2}) in the array A that is a potential solution of columns of timetable T. For example, for the leftmost solution in Figure 5.1, A[1]=5, A[5]=1, and A[2]=6 etc. Afterwards, an element of array A is the opponent of the team which is the index of the element. Finally, all solutions of the model (e.g., 105 for $n = 8$) are stored into tuples denoted with TP.

5.2.2.2 A Model for the Timetable

After obtaining the tuples (TP) filled with all arrays converted from the perfect matchings, the constraint imposed on each column of the matrix T can be stated as:

$$\{(T_{1j}, T_{2j}, ..., T_{nj}) \mid j \in \mathbb{Z},\ 2 \leq j \leq n\}\ \subseteq\ TP \tag{5.39}$$

Constraint (5.39) and Property (5.33) can codetermine the feasible solutions of the timetable without the involvement of constraints imposed by Property (5.32). The implementation of the model utilizes the *table* constraint specified with TP to limit possible combinations of values for each column of T other than the first column.

5.2.2.3 Experimental Results

Before elaborating the entire model for TTPPV, we would like to first compare the part of our model completed so far with the corresponding parts of the model presented in [161, 212]. In this section, all the models were implemented in Choco Solver 4.0.6 [165] with JDK version 9.0.4 and all experiments were performed on a computer with an Intel i7-3720QM CPU, 2.60GHz and 8 GB DDR3 memory running Ubuntu 17.10. Both models are used to generate timetables of n teams satisfying Properties (5.32) and (5.33). Note that there is no HAP table being generated in this comparison; therefore, it is not problem-specific and requires neither predefined venues nor distances between these predefined venues.

Instances	Solutions	Time (s)	Nodes/s
n=8	15,724,800	(147, 2062)	(214521, 15671)
n=10	15,724,800	(193, 3863)	(163849, 9037)
n=12	15,724,800	(238, 7326)	(133194, 4928)
n=14	15,724,800	(314, 17470)	(101161, 2035)

Table 5.13: The comparison between our timetable model and the timetable model presented in [161]. The data in parentheses separated by commas were calculated by our model (left) and the model of [161] (right) respectively. (Table reproduced from [122].)

Table 5.13 reports the execution times for generating 15,724,800 timetables from $n = 8$ to $n = 14$.[14] Our model outperformed the model presented in [161] for all the listed instances in the table in terms of execution time (s) and node processing speed (nodes/s). Moreover, the advantage becomes more obvious with each increase in the size of the instance.

5.2.2.4 A Complete Model

We are now going to present a complete model for TTPPV. Having calculated the number of perfect matchings N_p, we can derive the upper bound of the overall search space for the complete model, given by $\frac{\binom{N_p}{n-1}}{2}$. As mentioned, the first column of the matrix T is always fixed with $[1, \ldots, n]$. Thus, the number of combinations of the rest of $n-1$ columns from N_p possible columns is $\binom{N_p}{n-1}$ and can be reduced by half due to the symmetries reflected in the horizontal axis. Nevertheless, this upper bound is not tight enough because each row of the timetable must take on the distinct values from $\{1, \ldots, n\}$, required by Property (5.33). Moreover, if the variables representing timetable and HAP table are tied together and evaluated simultaneously, restrictions imposed on the HAP table can also rule out some unqualified timetables. For instance, the timetable which results in a HAP table with more than 3 consecutive away games are filtered out. The predefined venues and the round information (values assigned to a column of the timetable) together determine the home-away pattern for the round, and both of them are available after executing the model that generates all perfect matchings presented in Section 5.2.2.1. Therefore, we are able to construct the potential values of columns composed of the round information and its home-away pattern together for the complete model.

[14]The total number of solutions for instance n=8 is 15,724,800. For the larger instances, the total number of instances are much greater than 15,724,800, we still use it for the convenience of comparing two models.

In light of these considerations, we define the decision variables of the complete model as an $2n \times (n-1)$ matrix C with integer variables, where the first n rows of the matrix represent the timetable, each of which has domain $\{1, \ldots, n\}$ and the last n rows of the matrix represent the HAP table, each of which has domain $\{0, 1\}$.

We have elaborated how to generate the potential values of columns for timetable matrix T in Section 5.2.2.1. For the complete model, we also generate the potential values of columns for the matrix C. For instance, potential values of a column for the matrix C for 8 teams could be [2 1 6 5 4 3 8 7 0 1 1 1 0 0 0 1] in which the first 8 elements are the opponents of teams $[1, \ldots, 8]$ in a round and the last 8 elements are the home-away assignments for the corresponding games. Since the first n rows of the matrix C stand for a timetable, we use the solutions generated by the perfect matching model and the transformation rule defined in Constraint (5.38) to obtain the potential values of the first n rows of the columns for the matrix C. We rewrite Constraint (5.38) since the length of the array is changed, given by:

$$\forall_{i \in \mathbb{Z}},\ 1 \leq i \leq \frac{n}{2},\ A'_{P_{i1}} = P_{i2},\ A'_{P_{i2}} = P_{i1} \tag{5.40}$$

where the length of the array A' is $2n$. In addition, the last n rows of the columns for the matrix C are home-away assignments with the corresponding teams decided by Constraint (5.40) and predefined venues. The last n elements of the array A' are defined by:

$$\forall_{j \in \mathbb{Z}},\ n \leq j \leq 2n,\ A'_j = PV_{(j-n)A'_{j-n}} \tag{5.41}$$

where PV is the predefined venue table (see for example Table 5.12) in which each element at row i' and column j' is a home-away assignment ($\{0, 1\}$) for team i' and team j'. Hence, any last n element A'_j is the home-away assignment for the team A'_{j-n} and team $j - n$ which are calculated by Constraint (5.40).

Having defined Constraints (5.40) and (5.41), all solutions of the perfect matching model are converted to arrays of length $2n$ with round information and home-away assignments and these arrays are stored into tuples denoted with *TPC*. As with the Constraint (5.39) for the timetable model, the constraint imposed on each column of the matrix C can be stated as:

$$\{(C_{1j}, C_{2j}, ..., C_{nj}) \mid j \in \mathbb{Z},\ 1 \leq j \leq n-1\} \ \subseteq \ TPC \tag{5.42}$$

As a result of Constraint (5.42), Properties (5.32) and (5.34) can be satisfied simultaneously and the *ifThen* constraint and the *element* constraint are avoided.

Additionally, since a feasible solution and its reversed solution have the same cost function value, we can shrink the search space through static symmetry breaking by adding a simple constraint, given by:

$$C_{11} < C_{1(n-1)} \tag{5.43}$$

After applying Constraint (5.43), the search space is reduced by half.

For the consecutive condition required by the HAP table (i.e., a fixed upper bound for the number of consecutive away games or home games for every team), the regular language membership (*regular*) constraint is used here to impose on the last n rows of G. We also use the small DFA presented in [161] as the input of the *regular* constraint to filter out the set of bit strings that contain more than two consecutive 0 or 1 for the last n rows of C.

In summary, the complete model is composed of Constraints (5.42), (5.43), the *regular* constraint mentioned in this section, and the constraints imposed by Property (5.33).

5.2.2.5 Executing the Complete Model in Parallel

The EPS approach is well-suited for solving the TTPPV problem in parallel since disjoint partial solutions can be easily obtained before constraint solving and then mapped to different workers. There are two basic kinds of EPS in terms of the mapping method for parallel computing. The static decomposition method implies that a few subproblems for EPS are generated. In contrast, the dynamic decomposition method splits the problem into a large number of subproblems during evaluation, which ensures each worker has equivalent activity time. We use EPS with static decomposition to accelerate the solving process of TTPPV because our goal is to obtain a feasible solution with a better objective value and the problems are too large to search exhaustively. The generic procedure can be summarized as follows:

1. A subset of the decision variables of the model is selected.
2. All the partial assignments over selected variables in the subset are generated, which can be extended to a feasible solution of the TTPPV problem.
3. The partial assignments are mapped to the workers so that each worker can work on its own independent search space by using its own constraint solver.
4. The last step is to merge the results calculated by each worker.

In order to run this model in parallel, we can obtain all candidate partial assignments before the problem-solving process. For n teams, we generate all possible permutations of the set $\{2, \ldots, n\}$ in which the first element is less than the last element due to the static symmetry breaking, as the partial assignments for the first row of its decision variable matrix C. Then each worker receives the same number of partial assignments, and also utilizes the *table* constraint with the received partial assignments as its input tuples. By doing so, each worker can work on its own search space by using the same model presented in Section 5.2.2.4, and therefore *data-level parallelism* (see Section 2.2 for the definition of data-level parallelism) is achieved.

5.2.2.6 Experimental Results

In this section, we give results of our experiments on the complete model described in Section 5.2.2.4 with different number of workers. The comparison between the complete model and other models will be presented in Section 5.2.3.3.

Before running the models in parallel, we investigated the most suitable filtering algorithms for the *table* constraints used to partition the search space and the complete model (i.e., Constraint 5.42). The result shows that (FC,CT+) was the best among all candidates algorithms on instance n=8, 10, and 12 for all numbers of workers, where FC and CT+ stand for the forward checking [11] and compact-table algorithm [44] for the *table* constraint.

Workers / Time(m)	1	2	4	8
1	(7.48e4,20982,166)	(1.65e5,22409,166)	(3.78e5,20390,162)	(5.08e5,12511,162)
10	(1.16e6,20512,162)	(2.68e6,21166,156)	(5.19e6,17719,156)	(6.48e6,10458,156)
100	(1.46e7,20665,156)	(2.72e7,19333,154)	(4.96e7,16582,154)	(6.25e7,10081,152)
1000	(1.61e8,20324,154)	(3.14e8,19231,154)	(5.51e8,16044,152)	(7.31e8,10438,152)

Table 5.14: The experimental results on instance n=10. (Table reproduced from [122].)

We summarize the experimental results in Table 5.14 in which the parentheses of each cell presents the number of feasible solutions, average node processing speed, and optimal value when using different numbers of workers and execution time. Table 5.14 shows that the theoretical speedup can be achieved when running two or four complete models in parallel. Moreover, the optimal values for the travel distance were improved with additional workers involved.

5.2.3 An Advanced Modeling Approach for Larger Instances (A Second Model)

In this section, we first present another modeling approach dedicated for larger instance ($n \geq 14$). Afterwards, we also exploit parallelism in solving these instances for gaining speedup. The experiment results are also given.

5.2.3.1 An Advanced Model

The model presented in Section 5.2.2.4 (the first model) works well for small instances ($n < 12$) since it sacrifices memory space to improve efficiency. However, it suffers from memory usage explosion for large instances such as $n \geq 20$. Thus, we cannot bypass the *ifThen* constraint by using the first model. We improve the model presented in [161, 212] that still involves the *ifThen* constraint and the *element* constraint. Therefore, due to the use of the *ifThen* constraint, the model presented in this section (the second model) performs worse than the model presented in Section 5.2.2.4 (the first model) when solving the small instances ($n \leq 14$). However, our model achieves better performance by more effective search space reduction. The basic idea behind our improved model is to utilize the predefined venues when constructing timetables, that is, we filter out the timetables whose corresponding HAP table does not meet balance condition and consecutive condition. Roughly speaking, this model is also logically equivalent to the Properties (5.32), (5.33), and (5.34), where Property (5.34) must rely on Property (5.32) as discussed previously.

Since we know predefined venue tables before modeling (e.g., Table 5.12), a set of k consecutive opponent teams (tuples) whose corresponding HAP table does not violate the consecutive condition can be generated directly for each team. Then, we restrict that every k consecutive variables in the row of the timetable for a team must take k consecutive values from its corresponding consecutive opponent teams. Any allowed k consecutive opponent teams are composed of $k - i$ values taken from *away set* and i values taken from *home set*, where $1 \leq i \leq k - 1$, and away set consists of all away games for the team, otherwise home set. If every k consecutive variables in each row (team) of matrix T (timetable) is assigned to allowed values, the timetable satisfies the consecutive condition automatically. And the search space is reduced by avoiding the timetables that do not satisfy the consecutive condition.

Example 1. Let us consider a predefined venue table as shown in Table 5.12. Team 1 has to play against teams $\{2, 5, 6, 8\}$ away, $\{3, 4, 7\}$ at home. Thus the sets $\{2, 5, 3\}$,

Algorithm 3: Generate an array of allowed tuples

```
   Input  : n, k, arr2_PredefinedVenue
   Output: Tuples[] arr_Tuples
1  Create lists list_AwaySet, list_HomeSet, sets_Permutations and
    sets_AllowedSet;
2  for i ← 1 to n do
3  |  for j ← 1 to n do
4  |  |  if arr2_PredefinedVenue[i][j]==0 then
5  |  |  |  add j to list_AwaySet;
6  |  |  if arr2_PredefinedVenue[i][j]==1 then
7  |  |  |  add j to list_HomeSet;
8  |  end
9  |  getAllAllowedSets(k, 1, list_AwaySet, list_HomeSet, sets_AllowedSet);
10 |  getPermutations(sets_AllowedSet, sets_Permutations);
11 |  forall permutation ∈ sets_Permutations do
12 |  |  add the permutation to arr_Tuples[i];
13 |  end
14 |  clear list_AwaySet, list_HomeSet, sets_AllowedSet,
   |   sets_Permutations;
15 end
16 return arr_Tuples;
```

Algorithm 4: getAllAllowedSet(k,i,list_AwaySet,list_HomeSet,sets_AllowedSet)

```
   Input  : i, k, list_AwaySet, list_HomeSet, sets_AllowedSet
1  if i==k then
2  |  return;
3  sets_SubsetAway = getAllSubsets(list_AwaySet, k − i);
4  sets_SubsetHome = getAllSubsets(list_HomeSet, i);
5  forall set_Away ∈ sets_SubsetAway do
6  |  forall set_Home ∈ sets_SubsetHome do
7  |  |  add set_Away ∪ set_Home to sets_AllowedSet;
8  |  end
9  end
10 getAllowedSet(k, i + 1, list_AwaySet, list_HomeSet, sets_AllowedSet);
```

$\{2, 5, 4\}$, $\{2, 5, 7\}$, $\{5, 6, 3\}$, $\{6, 8, 7\}$, and $\{8, 4, 7\}$ etc. are the allowed values if the maximal number of consecutive away games is 3 (i.e., $k = 3$).

We now present how to generate allowed consecutive opponent teams for each team. Algorithms 3 and 4 depict how k consecutive allowed values for each team are generated and added to an array of tuples when given a predefined venue table. In lines 3–7 of Algorithm 3, *list_AwaySet* and *list_HomeSet* are created and added

the away set and home set for each team. In line 9, Algorithm 3 invokes the method described in Algorithm 4 that recursively adds the sets into $sets_AllowedSet$, each of which contains exactly k elements that are allowed for k consecutive variables of the timetable. Afterwards, all permutations of each element in $sets_AllowedSet$ are obtained and stored by $sets_Permutations$ in line 10. Finally, these permutations are added to the tuple corresponding to the row (team) in line 12.

Algorithm 4 can only be invoked by Algorithm 3 and is a recursive method that generates sets consisting of $k - i$ values taken from a given away set and i values taken from a given home set. The result sets for a given team, which is similar to the sets shown in Example 1, are added to $sets_AllowedSet$ on each recursive call in line 7. When the base case is reached in line 1, $sets_AllowedSet$ includes all allowed set containing exactly k elements taken from both away set and home set.

Having the set of allowed values for each team generated by Algorithm 3, the constraints imposed by the consecutive condition can be expressed as:

$$\{(T_{ij}, T_{i(j+1)}, ..., T_{i(j+k-1)}) \mid i, j \in \mathbb{Z},\ 1 \leq i \leq n,\ 2 \leq j \leq n-k\} \ \subseteq\ AL_i \tag{5.44}$$

where AL_i stands for all the sets of k allowed consecutive values for team i, storing in $arr_Tuples[i]$ (cf. Algorithm 3). Note that in the above constraint j starts at 2 because the first column of any feasible solution of TTPPV always contains the fixed values $\{1, \ldots, n\}$ in our model.

As already pointed out in [161], we also add *implied constraints* [192] to accelerate resolution process without changing the set of solutions of the model. The implied constraints for the model can be stated as:

$$\forall_{i,i',j \in \mathbb{Z}},\ 1 \leq i < i' \leq n,\ 2 \leq j \leq n,\ T_{ij}\ \neq T_{i'j} \tag{5.45}$$

These implied constraints are the instances of the *allDifferent* constraint. As with the first model (Section 5.2.2.4), the static symmetry breaking constraint is also introduced:

$$T_{12} < T_{1n} \tag{5.46}$$

In summary, Constraints (5.44), (5.45) and (5.46), together with constraints entailed by Properties (5.32) and (5.33), form the model used to tackle large instances ($n \geq 12$). With a balanced predefined venue table, a solution of the model is a feasible solution of TTPPV.

5.2.3.2 Solving the Model in Parallel for Larger Instances

In Section 5.2.2.5, we have presented the approach to partition the overall search space for the TTPPV problem. However, that approach cannot be applied to the large instances model because of the following factors. First, it is impossible to generate all the possible first rows (e.g., $(18-1)!$ when $n = 18$) for a large instance in a reasonable execution time. Second, it is impossible to store all the possible first rows in memory because out of memory exceptions would occur. Thus, we generate all possible assignments for elements starting at index 2 to $k+1$ $(0 < k \leq n)$ in the first row of T instead of the entire first row. As mentioned, we generate k consecutive allowed values for each team and store them in *sets_Permutations* in Algorithm 3. Thus, we decide to use the k consecutive allowed values of the first team to partition the search space. For a worker with a unique ID, id, the k elements starting at index 2 to $k+1$ in the first row (team 1) can be calculated as:

$$\begin{aligned} \{(T_{12}, T_{13}, ..., T_{1(k+1)}) \mid \forall_{i \in I},\ (P_i),\ T_{12} \neq n\} \\ I = \{i \in \mathbb{Z} \mid i \bmod \#W = id\} \end{aligned} \tag{5.47}$$

where $\#W$ stands for the number of workers to be used, and id is the unique ID of each worker with domains $\{0, \ldots, \#W-1\}$. And P is an array of tuples generated in Algorithm 4 and storing k consecutive allowed values for team 1. To satisfy the static symmetry breaking constraint, the variable T_{12} cannot be equal to n.

5.2.3.3 Experimental Results on the Large Instance Model

We first conducted a comparison of our first model (the model presented in Section 5.2.2.4), the large instances model (the second model) and the model presented in [161].

Instances	Time(s)	Solutions	Optimum value	Speed(n/s)	Nodes
n=8	(**9**,42,115)	(80822,80822,80822)	(76,76,76)	(**36946**,8739,4820)	(**3.56e5**,3.69e5,5.59e5)
n=10	(**300**,300,300)	(**4.55e5**,3.45e5,5.98e4)	(**162**,164,168)	(**20675**,6734,3796)	(**6.20e6**,2.02e6,1.14e6)
n=12	(300,**300**,300)	(7.21e4,**3.57e5**,1.00e5)	(312,**296**,292)	(15718,**4758**,1964)	(4.72e6,**7.13e5**,5.89e5)
n=14	(300,**300**,300)	(2.85e3,**1.01e5**,1.93e4)	(508,**490**,502)	(9883,**3316**,1795)	(2.96e6,**9.95e5**,5.38e5)
n=16	(N/A,**300**,300)	(N/A,**3.3e4**,2.79e3)	(N/A,**752**,780)	(N/A,**2855**,1658)	(N/A,**8.56e5**,4.98e5)
n=18	(N/A,**300**,300)	(N/A,**4.05e4**,8.18e3)	(N/A,**1140**,1162)	(N/A,**1643**,977)	(N/A,**4.93e5**,2.93e5)
n=20	(N/A,**300**,300)	(N/A,**874**,777)	(N/A,**1640**,1566)	(N/A,**1085**,929)	(N/A,**3.25e5**,2.79e5)

Table 5.15: Comparison of the three models on different size of instances. The data in parentheses separated by commas were calculated by the first model, the second model, and the model of [161] respectively. N/A indicates that our first model cannot handle the instance. (Table reproduced from [122].)

Times(m)	# workers	# feasible solutions	processing speed	objective value
1	1	7545	1711	1154
	2	8026	1576	1130
	4	8856	1131	1130
	8	4652	685	1116
10	1	7.60e4	1717	1138
	2	9983	2114	1128
	4	70702	1229	1104
	8	34213	595	1100
100	1	65e5	1562	1128
	2	2.05e5	1866	1104
	4	2.51e5	1248	1102
	8	7.03e5	661	1084
1000	1	8.90e6	1636	1110
	2	3.3e6	1700	1098
	4	6.41e6	1153	1096
	8	3.89e6	652	1078

Table 5.16: The experimental results for 18 teams using the second model.(Table adapted from [122].)

Table 5.15 shows the experimental results calculated in 5 minutes by three models from instances size $n = 8$ to $n = 20$, where the exceptional instance $n = 8$ could be finished in a shorter time by all models. For the small instances ($n = 8.10$), our first model outperforms the other two models in terms of node processing speed and the number of feasible solutions. By contrast, the second model can find a better objective value and process nodes faster for instances larger than 10. Indeed, for instance n =20, the objective value obtained by our second model is worse than that of the model presented in [161]. We argue this is because the solution distribution in the models and the experimental results of parallel runs will support our argument.

We also carried out the experiment that selects the best filtering algorithms for the *table* constraints that are used for partitioning search space and k consecutive variables. The best results were selected based on the sum of the number of feasible solutions of all workers. We obtained (MDD+, GAC3rm), (FC, GAC3rm), and (FC, GAC3rm) for 2, 4, and 8 workers respectively, where the first algorithm was specified in the *table* constraint for partitioning the search space and the second one is for the *table* constraint used for k consecutive allowed values.

Table 5.16 reports the results of parallel runs using 2, 4, and 8 workers as well as the sequential runs. With more workers, a better objective value is attained. We observed that the number of solutions and the speed of processing nodes declined

Times(m)	# workers	# feasible solutions	processing speed	objective value
	2	2024	2279	1132
1	4	9021	2310	1130
	8	23648	2187	1082
	2	17961	2292	1122
10	4	89174	2188	1118
	8	2.07e5	2273	1080
	2	5.40e5	2068	1120
100	4	5.68e5	2222	1104
	8	1.75e6	2260	1066
	2	1.64e6	2328	1096
1000	4	8.14e6	2300	1096
	8	2.03e7	2216	1066

Table 5.17: The results of simulation parallel executions by executing in sequential way. (Table adapted from [122].)

approximately by half, with the number of workers doubled from 4 to 8 (cf. Table 5.16). To confirm the effectiveness of the search space approach for the EPS approach, we simulated the parallel execution by executing the tasks received by parallel workers sequentially. Table 5.17 shows that the number of solutions and the speed of processing nodes remained stable as the number of workers increases, which means that the performance degradation observed in Table 5.16 is not due to our parallel approach. And given the same number of workers, we experienced the improvement in objective values when comparing the simulation and the previous parallel runs.

5.2.4 Discussion

As shown in Table 5.15, the first model achieved the best performance on the instance $n = 8$ due to its shortest execution time. And the first model has the least total number of search nodes among the three models. The use of the *ifThen* constraint introduces extra internal variables created by constraint solvers, thereby increasing the search space and degrading the performance of constraint solving. We argue that one should avoid using the *ifThen* constraint by reformulating CSP models. Moreover, we should develop new techniques to deal with the situation that requires the *ifThen* constraint.

For the parallel execution, the parallel efficiency decreases with the number of workers increased from four to eight. We believe that hyperthreading on a single processor causes decreased parallel efficiency due to limited cache size. Thus, we simulated the execution of parallel runs on a single core. The experimental results

reported in Table 5.17 indicates that the EPS approach itself scales efficiently with the number of processors.

5.2.5 Conclusion

We have presented two distinct models for different sizes of TTPPV problems, as well as utilizing data-level parallelism to execute models in parallel. On the same instances of realistic size, our models outperformed the previous model [161] under their own best search strategy. The advantage of our models is decided by the strategy that trades time for space. Specifically, the search space is reduced by predefined tuples with the possible combinations of values. We also observed that concerning the object value, the search space splitting with more parallel processors can lead to a feasible solution with better object value.

5.3 Talent Scheduling Problem

The Talent Scheduling problem (TS) is a practical problem entailed by devising a schedule for shooting a film, which is a typical Constraint Optimization Problem (COP). As we did for the SGP problem and the TTPPV problem, we first present a concise and efficient modeling approach for the problem, and then exploit TS as a case study to explore how to utilize the EPS approach to speedup constraint solving.

This section is a revised version of our previous work [126], and the remaining part of this section is structured as follows: In Section 5.3.1, we briefly introduce the definition of the problem. Then, in Section 5.3.2, we gradually describe the modeling approach for the problem in detail. Next, in Section 5.3.3, we examine how the EPS approach solves this problem in parallel, experimental results are given in Section 5.3.4. Finally, we conclude in Section 5.3.5.

5.3.1 The Introduction of the TS

The talent scheduling problem (TS) is an interesting NP-hard problem originally presented in [30] and is problem No.039 in CSPLib [190]. The problem can be described as follows: the process of making a film is partitioned into n individual pieces, each of which may require a different subset of the resources such as actors, props and costumes, etc. that can be viewed as a set whose members are m different resources. Besides, the duration of pieces varies according to the requirement of the film shoot; the cost of different resources is paid at different rates. For a given piece, the

Piece	4	1	11	10	13	12	3	2	6	8	7	9	5	20	15	14	17	18	16	19	Cost/100
resource 1	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	10
resource 2	0	1	0	1	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0	4
resource 3	0	0	0	0	1	0	1	1	0	1	1	0	1	0	1	1	0	0	0	0	5
resource 4	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
resource 5	0	0	0	0	0	1	0	1	0	1	1	0	0	1	0	1	0	1	0	1	5
resource 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	40
resource 7	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	4
resource 8	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	20
Duration	1	2	1	2	2	1	1	1	3	1	1	1	1	1	2	1	1	2	1	1	

Table 5.18: A feasible solution for a given TS in [191]. The Cost and Duration in the last column and the last row stands for the cost per time unit and the duration of the pieces, respectively. The overall cost of this solution is 14,600. See below for calculating the cost of a solution. (Table reproduced from [126].)

cost incurred by one resource is equal to the product of the duration of the piece and the cost of the resource. A feasible solution of a TS problem can be represented as a table (see for example Table 5.18), in which each column stands for a fixed set of resources required by a piece, while each row represents the demand for the resource for all the pieces of the film.

One feasible solution of the TS problem differs from another only due to their different permutations of pieces (i.e., columns) because interchanging any two columns of the solution might alter the overall cost of the film. By contrast, the order of resources (i.e., rows) can always be fixed since interchanging any two rows does not lead to a different feasible solution. Thus, the order of the columns (pieces) in a feasible solution is vital to the cost of a film. A cell of a feasible solution is assigned to one if the resource is required by the piece, otherwise zero. For example, in Table 5.18, piece 4 requires resource 1 and does not require resource 2; hence, the corresponding cells are 1 and 0.

The cost of a resource only depends on the interval between the first piece in which it is involved and the last piece in which it is involved, which implies that the idle times of the resource in the interval also need to be paid. Here, if a resource lies idle in such an interval, we call it *idle resource*. For instance, resource 1 required from piece 4 to piece 8 lies idle for pieces 13 and 12 in the feasible solution shown in Table 5.18. In this case, resource 1 still needs to be paid even if not required by pieces 13 and 12. Thus, the additional expense incurred by resource 1 is calculated by $10 \cdot 2 + 10 \cdot 1$, where 10 is the cost of resource 1, and 2 and 1 are the durations of pieces 13 and 12 respectively. The objective of the TS problem is to find a feasible solution that has

the lowest cost incurred by all the idle resources of the feasible solution. Given the above, the TS problem is a typical constraint optimization problem (COP).

Consider the making process of a film composed of n pieces and m kinds of resources, d_j ($j \in \{1, \ldots, n\}$) denotes the duration for piece j, while c_i ($i \in \{1, \ldots, m\}$) denotes the costs of resource i. Thus, we denote a feasible solution to the TS problem by an $m \times n$ (0,1) matrix T. To calculate the overall film cost, we define a function $\tau(i, j)$ as follows:

$$\tau(i, j) = 1 - T_{i,j} \tag{5.48}$$

where $T_{i,j}$ denotes the value in row i and column j of the given feasible solution (i.e., $T_{i,j}$ indicates whether resource i is required by piece j or not.), and the domain of j is over the interval between the first occurrence of "1" (f_i) and the last occurrence of "1" (l_i) in row i. Therefore, the total cost function for all the idle resources in a feasible solution s is:

$$cost(s) = \sum_{i=1}^{m} \sum_{j=f_i}^{l_i} \tau(i, j) \cdot d_j \cdot c_i \tag{5.49}$$

where the feasible solution s determines f_i, l_i, and $\tau(i, j)$. Therefore, the objective of the TS problem can be stated as finding a feasible solution that has the minimum value of the *cost* function, which is given by:

$$\underset{s \in P}{Minimize} \;\; cost(t) \tag{5.50}$$

where P is all the permutations of the columns of the feasible solution s, namely the total solution space of the given TS problem.

5.3.2 The CSP Model of the TS

We are now going to introduce our model in detail. Any feasible solution for the TS problem is a permutation of $\{1, 2, \ldots, n\}$, where n is the number of pieces involved in film shooting. Hence, the problem can be viewed as to assign n values to n slots. We define the decision variables as $X = \{x_1, x_2, ..., x_n\}$, each of which has domain $\{1, 2, \ldots, n\}$, where $x_i = j$ if slot i is assigned to piece j in the sequence. Therefore, the basic constraint of the model can be described as:

$$\forall_{i,j} \; (x_i \neq x_j) \tag{5.51}$$

where $1 \leq i < j \leq n$. Constraint (5.51) can be realized by the *allDifferent* constraint [112], which is implemented in almost all constraint solvers.

Although a TS problem can be solved by only using the *allDifferent* constraint, the search space would be immense even for a small number of pieces in practice, and consequently, we would not obtain a feasible solution in a reasonable time. For example, given a problem with 20 pieces (cf. Table 5.18), 20! permutations on the sequence $\{1, \ldots, 20\}$ causes that iterating over all the possible permutations is impossible in reasonable execution time. Thus, we have to reduce the search space by imposing additional constraints.

A feasible solution and the solution in its reverse order have the same cost function value because the expenses incurred by the idle resources in both solutions are the same. Thus, it is unnecessary to re-explore the symmetrical search region. The static symmetry breaking constraint in our model is relatively simple and can be stated as:

$$x_1 < x_n \tag{5.52}$$

The *arithm* constraint can be used to express Constraint (5.52), and the overall search space is reduced by half.

We can also take advantage of the intrinsic characteristics of the data to shrink the search space further. Specifically, if there is a fixed pattern for the optimal solutions, a set of *optimality constraints* can restrict the search space to the solutions containing such fixed pattern. An optimal solution remains unchanged by interchanging two pieces that require the same set of resources. However, as we can see from Table 5.18, there are some pairs of pieces in which two pieces request almost the same set of resources except for one difference. For example, piece 1 and piece 3 require almost the same resources apart from resource 3. As a result, only resource 3 affects the positions of pieces 1, 3 with other pieces in an optimal solution since any two pieces requiring the same set of resources are interchangeable in an optimal solution. Moreover, piece 3 must be closer to the pieces (e.g., piece 13) requiring resource 3 than piece 1 in an optimal solution because this arrangement of pieces will incur a lower overall cost. Based on this observation, we first find all the pairs of pieces (pieces i and j) requiring a set of resources with only one different resource and then find a piece (we call this resource benchmark) that requires that different resource as well (piece bm). The relationship between these pieces is restricted as:

$$|idx_i - idx_{bm}| \; < \; |idx_j - idx_{bm}| \tag{5.53}$$

where idx_i, idx_j, and idx_{bm} denote the index of pieces i, j, and bm, respectively. All

the possible pieces satisfying this relationship can be defined as follows:

$$\{(i,j,bm) \mid i \neq j \neq bm, |R_i \cup R_j| - |R_i \cap R_j| = 1, \\ (R_i \cup R_j) \setminus (R_i \cap R_j) \in R_{bm}\} \tag{5.54}$$

where R_i, R_j, and R_{bm} represent the set of resources required by pieces i, j, and bm, respectively (e.g., $R_1 = \{1,2\}, R_3 = \{1,2,3\}, R_{13} = \{2,3,4\}$ in the given problem shown in Figure 5.18). The equation $|R_i \cup R_j| - |R_i \cap R_j| = 1$ ensures the resources required by idx_i and idx_j with only one difference. Piece bm entails the different resource between resources required by piece i and piece j, guaranteed by $(R_i \cup R_j) \setminus (R_i \cap R_j) \in R_{bm}$. For example, piece 13 can be the benchmark of pieces 3 and 1, and piece 3 must be closer to the benchmark than piece 1 is. Thus, we have the constraint: $|idx_3 - idx_{13}| < |idx_1 - idx_{13}|$. Note that piece 13 is not the only choice of the benchmark. All the pieces requiring resource 3 (e.g., pieces 2, 8 etc.) can be the benchmark for piece 3 and piece 1.[15]

Constraint (5.53) can be realized by the *inverseChanneling*, the *distance* constraint, and the *arithm* constraint. For a specific example of Constraint (5.53), we first use the *inverseChanneling* constraint to record the indices of the decision variables X by introducing new auxiliary variables *IDX*, given by:

$$\forall_{i,j \in \{1,\ldots,n\}} \; (X[i] = j \Leftrightarrow IDX[j] = i) \tag{5.55}$$

Having auxiliary variables IDX, we can restrict the distance between given values (pieces) in a output of a sequence easily. For given pieces i, j, and bm, Constraint (5.53) can be converted to:

$$distance \mid IDX[i] - IDX[bm] \mid < distance \mid IDX[j] - IDX[bm] \mid \tag{5.56}$$

For the implementations using Choco solver, two extra auxiliary variables (aux_1, aux_2) must be introduced to store the *resulting variables* of the distances since Choco cannot directly represent Constraint (5.56). Thus, we have

$$distance \mid IDX[i] - IDX[bm] \mid = aux_1 \tag{5.57}$$

$$distance \mid IDX[j] - IDX[bm] \mid = aux_2 \tag{5.58}$$

[15] Indeed, applying Constraints 5.53 to a CSP model of the TS problem entails the given problem having the characteristic that requires the same resources except for one difference. It is common for TS problems to have such a characteristic since pieces usually share most of resources during a film shoot. Thus, even without two pieces sharing the same resources with one difference, we can relax the restriction by allowing more differences between the two pieces.

Then, the *arithm* constraint can be used to restrict the relation between the two auxiliary variables, given by:

$$arithm\ (aux_1\ <\ aux_2) \tag{5.59}$$

An instance of Constraint (5.53) might reduce more than half of the entire search space because the solutions satisfying $|idx_i - idx_{bm}| \geq |idx_j - idx_{bm}|$ are ruled out. Before implementation, we should find all the pairs of pieces that only have one different resource, then impose the instances of Constraint (5.53) for these pairs on the model.

Local search (LS), an incomplete search method for finding an optimal solution, is often the method of choice to solve COPs because it can obtain feasible solutions with better objective value efficiently compared to complete systematic search [96]. Several ways to combine CP and local search have been proposed in the literature [52, 161, 96]. One way to utilize LS for CP is to freeze a fragment of the variables specified with fixed values and to solve the subproblem defined by the uninstantiated variables. This type of local search is essentially the same as the EPS approach we used to solve the SGP and TTPPV problems since both approaches require a subset of variables to be fixed. The only difference is that parallelism is exploited by the EPS approach. Hence, analogous to the SGP and TTPPV problems, we should carefully decide which variables should be frozen. For the problem given in Table 5.18, resources 6 and 8 cost much more than other resources, and there is no intersection between pieces requiring resources 6 and the pieces requiring resources 8; therefore, we presume that pieces involving resource 6 or resource 8 are more likely to arrange together within an optimal solution. Note that this method cannot ensure that no optimal solution will be ruled out, but the experimental result shows that the best solution we obtained is the same as that one provided in [191].

In our implementation, we treat pieces 6, 7, 8, and 9 as an *entirety*, say, piece 21, and also treat pieces 14, 15, 16, 17, and 18 as another entirety, say, piece 22. Consequently, the entire search space is reduced from 20! to $13! \cdot 5! \cdot 4!$ by this way. Indeed, the more freezing variables are, the more search space reduction is. But, there is a trade-off between search space reduction and the loss of optimal solutions. In principle, when applying LS to solve TS problem, we should consider freezing the pieces entailing the highest cost of the resource, the second-highest cost of the resources, and so on. Then, we treat these pieces as entireties. We select variables for LS using an automated way in three steps:

1. We sort resources based on their cost in descending order.

2. We freeze pieces requiring the current highest cost resource.
3. If the number of frozen pieces can lead the computation ends in predefined execution time, say, 20 mins, then the selection procedure stops. Otherwise, we set the next resource in the sorted resources as the current resource and then repeat step 2.

In summary, Constraints (5.51) and (5.52), together with constraints entailed by Constraint (5.53) and LS form the model used to tackle the TS problem.

5.3.3 Solving the TS in Parallel

As with the SGP and the TTPPV problems, the EPS approach is also well suited for solving the TS problem in parallel since disjoint partial solutions can be easily obtained before the solving process and then mapped to workers. Hence, we use EPS with static decomposition to accelerate the solving process of TS problem.

To parallelize the constraint solving for TS, we first select a subset of decision variables. One viable way is to select the pieces that do not belong to any entirety, i.e., pieces frozen by LS are excluded. For the given problem in Table 5.18, we chose pieces $PS = \{1, 2, 3, 4, 5, 10, 11, 12, 13, 19, 20\}$ to generate around billion partial solutions.

Then, the model presented in Section 5.3.2 is applied to the decision variables for PS, where the model used to generate the partial solutions includes Constraints (5.51), (5.52), and constraints entailed by Constraint (5.53). Please recall that all the pairs of pieces that only have one different resource are identified, and then we impose the instances of Constraint (5.53) for these pairs on the model in the sequential version. In the parallel version, to generate partial solutions, the instances of Constraint (5.53) would not involve the decision variables entailed by the pieces that are not being contained in PS. There are two advantages by doing so: First, the partial solutions can be generated in a reasonable time. Second, solutions that cannot be extended to an optimal solution will not be generated.

After generating the partial solutions, each worker receives the same number of partial solutions and works on its own independent partial solutions in parallel. Embarrassingly parallel execution works as follows on each worker:

1. We replace the entireties on all possible positions of a partial solution to obtain a feasible solution (i.e., no piece would be left out).

2. The cost function (Equation 5.49) is evaluated for each permutation. The constraint-based branch-and-bound approach is used here to reduce the nodes that are impossible to be extended as an optimal solution.

Note that PS does not include all the possible pairs of pieces that only have one different resource. Thus, we should impose Constraint (5.53) on the remaining part of such pairs of pieces to further shrink the search space of each worker. Due to no communication between workers, a new lower bound discovered by a worker cannot be used by other workers for improving their current resolution. The final step is to obtain the permutation with the smallest value of the cost function.

5.3.4 Numerical Results

In order to confirm our theoretical discussion, we implemented the model as described in Section 5.3.2 and Section 5.3.3 and its parallel version in the Choco Solver 4.0.6 [165] with JDK version 9.0.1. All experiments were performed on a computer with an Intel i7-3720QM CPU, 2.60GHz with 4 physical and 8 logical cores, and 8 GB DDR3 memory running Ubuntu 17.10.

All the techniques of solving the TS problem in reasonable execution time are essentially to reduce search space. The constraints regarding search space reduction allowed us only to evaluate 1,027,403,520 out of 20! feasible solutions, reducing approximately 99.99% of the total search space. The following experiments were carried out to test the effectiveness of the search space reduction sequentially and its parallel version.

Number of Workers	1	2	4	8
Execution time (s)	451.68	250.56	137.64	119.82
Speedup	1	1.8	3.28	3.77
Efficiency	1	0.9	0.82	0.47125

Table 5.19: Solving the TS on a multi-core computer. (Table reproduced from [126].)

As can be seen from Table 5.19, the efficiency dropped rapidly as the number of workers increasing from 4 to 8. Theoretically, we would not have experienced this result since no communication is required. Thus, to eliminate the factors such as the limit number of physical cores and the limited size of the cache, we partitioned the partial solutions into two parts, as did we for the TTPPV problem. Then we separately solved the first part of the partial solutions by parallel execution, and the second part of the partial solution. As given in Table 5.20, the execution times for

the first and the second part of partial solutions are 83.34s and 81.42s. A speedup of 5.42 ($\frac{451.68}{83.34}$) with 8 physical cores was obtained, which is higher than the speedup of 8 threads with hyperthreading. The parallel efficiency is improved from 0.47 to 0.68 ($\frac{5.42}{8}$).

Four Workers	First Part	Second Part
Execution time (s)	83.34	81.42

Table 5.20: Using 4 workers to calculate the first part and second part in turn.

The execution time of the sequential part for generating the partial solutions was 25 seconds in the parallel version; therefore, by using Amdahl's law, we can calculate that the theoretical speedup of our approach is around 18.1 for the given problem shown in Table 5.18.[16] Additionally, one benefit of parallel constraint solving for the TS is that we may gain more optimal solutions in a shorter time. For the problem shown in Table 5.18, we obtained the following optimal solutions:

4	1	11	10	13	3	12	2	6	8	7	9	5	20	15	14	17	18	16	19
4	1	11	10	13	12	3	2	6	8	7	9	20	5	15	14	17	18	16	19
4	1	11	10	3	13	12	2	6	8	7	9	20	5	15	14	17	18	16	19
4	1	11	10	13	3	12	2	6	8	7	9	20	5	15	14	17	18	16	19

Table 5.21: Optimal solutions with cost 14,600. (Table reproduced from [126].)

5.3.5 Conclusion

We have presented a model for the TS problem, as well as utilizing data-level parallelism to speed up the execution. Besides, our approach also employs local search to reduce the search space. The experimental results indicate that the EPS approach is an appropriate choice for solving such COPs. But we believe there is still the potential to improve the performance of our approach. Although it has performed well, Constraint (5.53) and the dual variables could be replaced by a customized constraint. Besides, the theoretical speedup can still be improved when solving larger instances because the non-parallelizable part is negligible for these instances.

[16] We can predict the theoretical speedup of a given parallel algorithm by using Equation 2.7 (page 24). For this question, the theoretical speedup is $1/1-\frac{451.68-25}{451.68}$.

5.4 Conclusion

In this chapter, we have presented detailed solving processes for three computationally hard problems in which the SGP problem is a constraint satisfaction problem, and the TTPPV and TS problems are constraint optimization problems. The improvements in node processing speed and solvable instances size can be attributed to the two aspects: the well-designed customized CSP models and the EPS approach. We have demonstrated that reformulating CSP with a better model can reduce the number of variables and or the domain size of variables, thereby shrinking search space. Furthermore, the EPS approach also brings benefits to constraint solving, i.e., more nodes processed simultaneously and early diversity. For the SGP problem, the obtained solutions of new instances and the observed superlinear speedups confirm our theoretical analysis conducted in Section 4.2.4. That is, by exploiting parallelism to search different subspaces simultaneously, we introduce early diversity into the backtracking search to avoid a strong commitment due to early mistakes made by search strategies at early stages in search. For the COPs, higher-quality feasible solutions could be obtained with linear speedup, although superlinear speedup was not achieved since parallelism cannot enhance the efficiency of sequential solving.

Although we can solve new instances or solve existing instances faster by means of reformulating and modeling constraint models as well as exploiting parallelism, the inadequacies of our approaches are clearly evident and reveal two research topics. First, reformulating and modeling a problem as a CSP still relies on the constraint programmer's insight, skill, and ingenuity. As pointed out by Freuder & Mackworth (2005), "constraint programming is still somewhat of an art [55]." Now, more than a decade later, the challenge remains though much progress has been made on providing more general and powerful modeling tools for constraint programming. There exist many ways of automatic enhancing constraint models, such as aggregation algorithms (see, *e.g.*, Frisch *et al.* [57]), flattening and identical common subexpression elimination (see, *e.g.*, Nightingale *et al.* [151, 150], Rendl *et al.* [172]), automated symmetry breaking (see, *e.g.*, Mears *et al.* [136]), etc. However, to the best of our knowledge, the state of art constraint solvers provide little support for automated reformulating and modeling, and research on automated reformulating and modeling for constraint solving is still in its infancy. Second, the choice of variables for search space splitting still depends on the constraint programmer's insight into a given problem as well. Indeed, the original EPS approach provides an automated decomposition method,

namely DBDFS (cf. page 72). However, we are likely to generate many invalid subproblems (i.e., partial assignments that cannot lead to a solution) if the variables for problem decomposition are selected randomly or chronologically. For instance, for the SGP problem, if we select a column instead of a row for the search space splitting, more invalid subproblems would be generated because a column in our model is less restricted. In short, future work will be required to develop more generalized techniques to utilize the EPS approach.

Chapter 6

Parallel Stochastic Portfolio

It is not uncommon to observe that the performance of constraint solving on a particular problem can be easily influenced by altering the search strategy, restart policy and their parameter settings, etc. In the multicore era, this lack of robustness can be exploited to speed up the constraint solving by devising a parallel portfolio search that simultaneously executes different incarnations of a sequential solver on the same problem. In this chapter, we first investigate the techniques of existing single-solver-based portfolio approach in detail. On this basis, we gain insight into how to improve the portfolio approach. We then present the parallel stochastic portfolio search that benefits from the explicit early diversity resulted from randomization and parallelism.

This chapter is a revised version of our previous work [123]. We organize the rest of the chapter as follows: We start by briefly reviewing the parallel portfolio search and the proposed approach. We then provide an overview of the components of the current single-solver-based portfolio approach in Section 6.2. From this overview, we sum up the limitations of the current portfolio approach in Section 6.3. Then, Section 6.4 introduces and justifies our new parallel approach in detail. Experimental results and a discussion are presented in Section 6.5. Before concluding, related work is discussed in Section 6.6. Finally, conclusion is given in Section 6.7.

6.1 Introduction

The shift in processor design from increasing the clock speed to parallel architectures has become an irreversible trend. The constraints community has developed various techniques to embrace this change. As we have surveyed in Chapter 3, the techniques for parallel constraint solving can be categorized under four main types: parallel consistency and propagation, parallelizing the search process, parallel portfolio, and

hybrid approaches. In recent years, the majority of the researches on parallel constraint solving focus on parallelizing the search process such as search space splitting, or local search. The most notable feature of these parallel approaches is that workers are *collectively orchestrated to be part of the same overall search process* [61]. Thus, the main challenge of this type of approach is to balance the workload distribution between workers.

By contrast, parallel portfolio search [99] does not need to consider this issue. The idea of parallel portfolio search is to solve the same constraint model with different combinations of search strategies, restart strategies, nogood recording, and their parameter setting. Hence, communication is not required during resolution process. By using portfolio search, the likelihood of finding a first solution within a timeout is increased. The enhancement is attributable to the diversification of exploration of the search tree in comparison with the sequential solver. But, as discussed in Section 3.3, there is also another broader way to define portfolio search. For example, Amadini *et al.* [6, 5] developed a portfolio solver consisting in 12 different types of constraint solvers, which tries to predict the most suitable solver for a given problem by exploiting machine learning techniques.

Nevertheless, we are concerned with improving the problem-solving ability of parallel portfolio approach on hard computational problems using only one type of solver, i.e., single-solver-based parallel portfolio approach. We propose a new parallel portfolio approach that aims at exploring the effective use of massively parallel architectures for constraint solving. Its key idea is to utilize explicit early diversity to avoid strong commitment to unfortunate variable choices due to the search strategies. Our portfolio approach has three advantages. First, it is easy to be implemented and solver-independent. Second, it can obtain excellent scalability due to the mechanism of the parallelism. Third, it can solve harder instances than the previous single-solver-based portfolio approach (e.g., the parallel portfolio implemented in Choco [165]).

6.2 The Components of the Current Single-Solver-Based Portfolio Approach

In this section, we give an introduction about the details of the parallel portfolio search in the state of the art constraint solvers, such as Choco [165] and Gecode [186]. Generally, portfolio means to exploit some different algorithms to obtain an overall better algorithm. In the context of CSP, a typical parallel portfolio might consist of different combinations of search strategies, abstract search strategies, nogoods, and

restart strategies. We are now going to review the essential components of a parallel portfolio search implemented in Choco and Gecode in more detail.

Adaptive search strategies

Adaptive search strategies compute the score for each candidate variable during resolution process on the basis of a specific criterion that takes account of information concerning the search space already explored, and then decide the next variable according to the score. The basic principle of these adaptive search strategies is to choose variables occurring in the difficult parts of the constraint network, which is called the fail-first principle [88]. The main difference among adaptive search strategies is the way of computing the scores. Now, we review the adaptive search strategies related to our approach.

1. WDEG [25] is widely implemented in most of the competitive constraint solvers [165, 186, 110].[1] The WDEG algorithm maintains a counter to record the weight degree for every variable. For a variable x_i, its counter (i.e., weight, denoted by $wdeg[x_i]$) increments whenever a constraint c_j covering x_i leads to a domain wipe-out in the scope of c_j. The score of each variable is computed by $\frac{|D(x_i)|}{wdeg[x_i]}$, where $|D(x_i)|$ denotes the domain size of variable x_i; $wdeg[x_i]$ denotes the weighted degree of variable x_i, defined by:

$$wdeg[x_i] = \sum_{c_j \in C} weight[c_j] \ s.t. \ x_i \in Scope(c_j) \wedge |FutScp(c_j)| > 1 \qquad (6.1)$$

where $weight[c_j]$ is associated with each constraint c_j whose scope covers variable x_i and $|FutScp(c_j)| > 1$ restricts that the number of uninstantiated variable in the scope of c_j must be greater than one.

2. Activity-Based Search (ABS) [139] chooses the variable based on the activities in which it is involved. In the context of ABS, activity refers to the removal of inconsistent values of a variable domain affected by constraint propagation. The activity of variable x_i, denoted by $A(x_i)$, is updated at each node of the backtracking search tree by the following two equations:

$$\begin{aligned} \forall_{x_i \in X \ s.t. |D(x_i)|>1} : \ A(x_i) &\coloneqq \gamma * A(x_i) \\ \forall_{x_i \in X'} : \ A(x_i) &\coloneqq 1 + A(x_i) \end{aligned} \qquad (6.2)$$

[1] In [25], the full name of WDEG was not given. We believe it means weighted degree.

where $\gamma \in [0,1]$ represents a decay parameter that helps forget the oldest statistics progressively and $X' \subseteq X$ denotes the subset of affected variables defined by:

$$\begin{aligned} \forall_{x_i \in X'} &: D'(x_i) \subset D(x_i) \\ \forall_{x_i \in X \setminus X'} &: D'(x_i) = D(x_i) \end{aligned} \tag{6.3}$$

For the value selection, ABS selects the value with the least activity, where the activity of assigning a value (a) to a variable (x_i) is counted by the number of variables ($|X'|$) affected by constraint propagation, defined by:

$$A(x_i = a) = |X'| \tag{6.4}$$

ABS has two phases. The first phase is the probe execution during which ABS calculates and accumulates activities for each variable without aging. The number of probs (probe executions) guarantees that the 95% confidence interval of the mean value of activities constructed by the Student's t-distribution for all variables is sufficiently small. Having the accumulated activities for all variables after the probs, the ABS selects the variable x_i with the greatest ratio $\frac{A(x_i)}{|D(x_i)|}$ (the score of the ABS).

3. For Impact-Based Search (IBS), we refer to [168].

The following two search strategies do not belong to adaptive search strategies: *First fail* picks the variable with the smallest number of remained values. *Occurrence* chooses the variable with the largest number of attached propagators.[2]

Abstract search strategy

Abstract search strategy is a class of auxiliary search strategies that cannot work alone. That is to say, they have to be applied on top of other search strategies (e.g., ABS, WDEG, etc.). The purpose of abstract search strategies is to help underlying search strategy reduce *thrashing* [40],[3] while still being a look-ahead algorithm. More precisely, it simulates a backjumping effect by a type of lazy identification of culprit variables, instead of using a look-back algorithm such as conflict-directed backjumping. Last conflicts (LC) [116] and Conflict ordering search (COS) [59] are effective and widely implemented abstract search strategies.

[2]Occurrence is the naming in Choco, it is called *largest degree* in Gecode.

[3]Thrashing is a phenomenon that repeats the exploration of failing subtrees of the backtracking search tree.

Restarts

Restarts is one of the major techniques to introduce the randomization into complete, systematic, backtrack search procedures.[4] The effectiveness of restart is due to the fact that the variance of the runtime distribution of the different variable orderings for the same model of some problems is considerable large [201, 73, 114], which is called *the heavy-tailed phenomenon*. Harvey [91] observed that restarting a backtracking search periodically with different variable orderings could mitigate the adverse effect caused by poor choices of variables at the early stage of the search. When using restart, we must consider a restart strategy that defines how we restart the backtrack search. A restart strategy $R = (r_1, r_2, r_3, ...)$ is an infinite sequence where each r_i is a *cutoff*, which means if no solution is found within cutoff r_i, the backtrack search starts again for the next cutoff r_{i+1}. Luby *et al.* [127] show that the fixed cutoff strategy $R = (r^*, r^*, r^*, ...)$ is optimal when knowing the complete knowledge of the runtime distribution. In addition to this, they presented a universal restart strategy (Luby's restart strategy) that minimizes the expected overheads when the runtime distribution is unknown.

Nogoods

A nogood is a partial instantiation of variables for a given CSP P that cannot occur in any solution of P. Moreover, nogoods can also be *branching constraints* [201] since an instantiation can be expressed in constraints. nogoods have a variety of uses for enhancing constraint solving. One approach to using nogoods is to obtain the backjump point (node) for the conflict-directed backjumping to reduce thrashing, where nogoods are discovered during the search process [178, 114]. Besides, for a CSP P containing symmetry, a symmetry mapping applied to nogoods can effectively prune the redundant search space [56]. Another important application of nogoods is to avoid exploring the same search space on restart. In competitive CSP solvers such as Choco solver [165] and Gecode [186], the nogoods are learned from failures during search and encoded as constraints. However, the two solvers follow two different nogood learning mechanisms. Gecode employs the *reduced nld-nogoods* for restarts that is originally proposed by Lecoutre *et al.* [115]. By contrast, Choco solver borrows the idea of the deductive proofs from the Proof-Producing CSP solver (PCS solver) [202].

[4] We have provided a more comprehensive introduction about the rationale and techniques of restarts in Section 4.2.1.

No.	Configuration	No.	Configuration
1	Default	17	dom/wdeg+nogood+restart+COS
2	dom/wdeg	18	dom/wdeg+nogood+restart+LC
3	ABS	19	ABS+nogood+restart
4	ABS + restart	20	First fail+COS
5	IBS	21	ABS+nogood+restart+LC
6	First fail	22	ABS+nogood+restart+COS
7	IBS + restart	23	IBS+nogood+restart+LC
8	First fail+LC	24	IBS+nogood+restart+COS
9	Occurrence+LC	25	Occurrence+COS
10	Input order+LC	26	Input order+LDC
11	Random Selection+LC	27	Random Selection+COS
12	Random Selection	28	dom/wdeg+restart+LC
13	dom/wdeg+restart+COS	29	ABS+restart+LC
14	ABS+restart+COS	30	IBS+restart+LS
15	IBS+restart+COS	31	IBS+nogood+restart
16	dom/wdeg+nogood+restart	32	dom/wdeg+restart

Table 6.1: A possible configuration of portfolio search (PPS).

Summary

We have reviewed how a parallel portfolio search is constituted and the technical details of its components. Table 6.1 gives a configuration of parallel portfolio search (PPS), which includes the default parallel portfolio search of Choco solver.[5] Furthermore, one can devise a portfolio search with more design options, such as different consistency levels of the constraint network, different filtering algorithms for the same constraint, different parameter settings for restart, and different restart strategies. Any changes to these options might significantly alter the potential search tree, thereby affecting the total number of searched nodes for finding one solution. Moreover, as far as we know, no generic algorithm can optimize these design options (e.g., search strategy, restart strategy, consistency level, etc.) before resolution process for CSP until now. In light of this, Gecode advocates for "always use parallel portfolios" [186], so long as the hardware supports multithreading.

[5]Gecode might have a different name for the same search strategy in Choco. This thesis follows the naming convention of Choco solver.

6.3 The Limitations of the Current Parallel Portfolio Search

In this section, we will investigate the potential limitations of the current parallel portfolio search implemented in Choco solver whose components have been discussed in Section 6.2.

In most cases, adaptive search strategies are deterministic. This means that the variable ordering determined by an adaptive search strategy, given the model of a CSP P with the same parameter setting (e.g., filtering algorithm, consistency level etc.), is always fixed. The only non-determinism comes from two or more than two variables with the same score, where the adaptive search strategy must choose one variable out of them at random. The determinism of adaptive search strategies implies that a problem, which is unmanageable because of its size, still cannot be solved even after many attempts. Nevertheless, if the score of all variables is not reset after restart, all adaptive search strategies exhibit learning ability when combined with restart (e.g., the configurations 4, 7, and 14, etc. in Table 6.1). In practice, Choco solver [165] and Gecode [186], encourage the use of restart combined with adaptive search strategies, because the resolutions are improved by sometimes restarting the search exploration from the root node. The basic principle of this technique is to utilize accumulated scores from adaptive search strategies in the previous round before restarting to increase the likelihood of the better choice of variables in the current or future rounds after restarting. However, since the constraint network structure, the filtering algorithm, and the consistency level remain unchanged after restart, the accumulated score of variables at each round keeps constant. Consequently, the combination of restart and adaptive search strategies may not be able to alter the variable ordering after a certain number of times restarts. Given a restart strategy, our empirical study proves that the variable ordering tends to be stable after some times of restart, for both ABS and dom/wdeg.[6]

For a restart strategy, it is generally hard to predict in advance which restart strategy and its parameter setting (e.g., scale factor) is the most suitable for a given problem [98, 87]. One possible remedy to this problem is to increase the number of workers for the parallel portfolio in which different restart strategies are executed simultaneously. In [87], Hamadi *et al.* examine this approach in the context of a SAT solver, which shows that using four different restart strategies can achieve about a

[6]The experiment was conducted by outputting and recording the scores of variables during restart-based searches with search strategies such as ABS and dom/wdeg.

factor 2 speed-up on eight processors. The result indicates a low parallel efficiency of this approach. Besides, we could also apply the dynamic restart strategy to different adaptive search strategies in parallel. The dynamic restart strategy aims at finding an optimal restart search strategy since the performance might be boosted over several orders magnitude by tuning the cutoff parameter [73]. The existing dynamic restart strategies [180, 106] utilize a Bayesian model to predict the runtime behavior of the algorithm on a problem instance. Unfortunately, this method is not widely implemented in the popular constraint solvers. We believe that the reason is the prior probability corresponding to a heterogeneous ensemble of instances is required, and it might be problematic to find such a training set for an arbitrary CSP P.

In addition to the above mentioned two factors, another restriction is the adaptive search strategy. It is true that determining the optimal search strategy for a given CSP P is a non-trivial task. Furthermore, even obtaining the first variable of an optimal variable ordering is at least as difficult as proving whether the CSP P has a solution [119, 178]; even so, the black-box search strategies, such as ABS and IBS, can achieve better performance than their previous search strategies on many problems. But it might be far from the optimal search strategies in most cases. One direct evidence is that black-box strategies can be augmented with restarts, and we will show the stochastic mixed search strategy can outperform these black-box search strategies on parallelism. Indeed, some hyper-heuristics can select the best strategy from a set of strategies by applying given properties for a given problem on the fly (e.g., machine-learning-based method of [9, 153], evolutionary-computation based method of [196, 154]), and recombination multiple strategies such as [26, 118]). However, we believe that no state of the art constraint solvers have implemented these methods; thus, it takes time to assess their effectiveness.

To sum up, we suppose that the following four main reasons hinder the current single-solver-based portfolio approach to solve hard computational problems in parallel:

1) As an essential constituent part of a single-solver-based portfolio approach, the combination of restarts and adaptive search strategy renders the variable ordering converging after a certain number of restarts.

2) A more sophisticated solution is required to find an optimized restart strategy for a given problem, and the result of parallelism in restart strategies is not encouraging.

3) The current search strategies, including static and dynamic search strategies, have difficulties handling complex and diverse CSPs.

4) Beyond all that, the power of parallel machines is not fully utilized because the search effort made by one worker cannot support other workers due to lack of a comprehensive set of orthogonal yet complementary search strategies without communication. Furthermore, it is hard to scale to an arbitrarily high number of processors because of the limited number of search strategies, let alone effective use of the processors.

6.4 A Novel Parallel Stochastic Portfolio Approach

So far, we have concluded four reasons which impede the current parallel portfolio search towards solving hard computational problems. In this section we present our novel parallel stochastic portfolio approach and carry out theoretical analysis for the approach.

Our parallel portfolio approach is simple to implement. It is composed of two search strategies in conjunction with nested restart strategies. Every worker executes the same constraint model with the same brancher consisting of two different search strategies and the same restart strategies. Roughly speaking, given a constraint model with n variables, k variables ($k < n$) are selected uniformly at random. The first search strategy is used to compute the candidate variable from the k variables. After branching on the k variables, the second search strategy guides the search by selecting from the rest of the $n - k$ variables. Concerning the restart strategies, the inner restart strategy is used to augment the search guided by the second search strategy, which means the inner restart strategy is only for learning purpose. Meanwhile, the outer restart strategy is responsible for terminating the current tree search, then commencing a new tree search from the root node, where the randomization is introduced by the first k randomly selected nodes (variables) in the search tree.

According to the heavy-tailed phenomena in CSPs, some unfortunate choices of variables early on are to blame for a long-running search process [74, 73, 201]. Because no consistent assignment to the early "wrongly chosen variables" can be found, the resolution process is led into a barren part of the search space, especially for hard search problems. Various researches have already confirmed that restarting a depth-first backtracking search with randomized search strategies (e.g., variable orderings) can eliminate heavy-tailed behavior to boost the performance of constraint solving [73, 74, 92, 201, 114]. As for random variable orders, complete random variable

ordering is ineffective for tackling structured instances [114]. However, it may suffice to begin the search with a different variable on each run to have separate search spaces when employing dynamic variable ordering heuristics [92, 91, 114]. Thus, it is natural to think of utilizing parallel computing to diversify at the top of a search tree by generating a random set of variables with cardinality k for the first search strategy on each worker so that the poor early variable choices can be offset. A main feature of the approach is that there is no need for communication when solving a large instance of a hard computational problem because each worker processes an *almost* independent search space exploration by setting k.

We now look in more detail at our work-splitting mechanism to justify our claim about zero-communication cost. Formally, let n_w be the number of workers and assuming that each worker restarts exactly n_r times. Besides, the first search strategy always determines the same variable ordering for given k variables, which implies no need to consider the permutations of k variables. For a given model with n variables and the first k variables of the first search strategy on each worker, we let $P(E_o)$ denote the probability of generating $n_w * n_r$ pairwise distinct sets, each of which has k variables. Thus

$$\begin{aligned} P(E_o) &= \frac{\binom{n}{k} * (\binom{n}{k} - 1) * (\binom{n}{k} - 2) * \ldots * (\binom{n}{k} - n_w * n_r + 1)}{\binom{n}{k}^{n_w * n_r}} \\ &= \prod_{w=1}^{n_w * n_r} \frac{\binom{n}{k} - w + 1}{\binom{n}{k}} \end{aligned} \tag{6.5}$$

where $k < n$, and $0 < n_w * n_r < 1 + \binom{n}{k}$. If we specify a probability threshold value T (e.g., 99%) for $P(E_o)$, we can calculate the minimum integer value of k for the inequality $P(E_o) > T$ for given n_w and n_r. In reality, hard CSPs often have hundreds variables or even more, thus, it does not need a big k to guarantee $P(E_o) > 99\%$. For instance, a model of a CSP P has 800 variables, then $k = 6$ is sufficient to ensure $P(E_o) > 99\%$ for 320 workers within 1000 restarts.

We are now in the position to explain why we employ the combination of two search strategies and the nested restart strategies. As mentioned before, the variables in our approach are divided into two groups. Indeed, both of them could use the same search strategy. In that case, then, it might be inappropriate for a small number of k variables to use sophisticated adaptive search strategies (e.g., ABS, IBS etc.) because these strategies need costly probe execution before search. Hence, we can employ a relatively simple search strategy (e.g., dom/wdeg). On the other hand, keeping k as small as possible means the vast majority of the variables are selected by the second

search strategy, which offers us an opportunity to take advantage of complex adaptive search strategies, such as ABS and IBS, since they are more robust and often produce performance improvements. In addition to this, the second search strategy can work hand in hand with the inner restart strategy to augment the search.

Finally, because of the way of exploiting the variability, we are able to generate a portfolio of arbitrary size on demand when applied to an arbitrarily high number of processors. Moreover, since the state of the art constraint solvers support the use of combining two search strategies [7] and the restart mechanism, one can readily integrate our approach into most of the constraint solvers.

6.5 Experimental Results

To confirm our theoretical discussion, we implemented our single-solver-based portfolio approach as described in Section 6.4 via the Choco solver 4.10.0 [165], then we compared our proposed approach with the parallel portfolio search described in Table 6.1 (PPS) on three distinct problems in the CSPLib [38]: Magic Squares [205], Sports Tournament Scheduling [206], and Costas Arrays [104]. The reason we chose these problems is that they do not require any third party benchmark (e.g., the Travelling Salesman Problem needs maps of instances). Thus, it is more convenient for other authors to make comparisons. Besides, we used the CSP models of these problems given by the examples of Gecode. The environment is JDK 10 under CentOS 6.5 (a Linux distribution) with four Intel Xeon CPU E7-4830 2.13GHz processors (32 cores total) and 250 GB DDR3 1066 memory.

	Magic Squares						Sports Tournament Scheduling			Costas Arrays	
Order	24	25	26	27	28	29	20	22	24	21	22
#vars.	577	626	677	730	785	842	442	485	783	949	1131
pps(s)	8.36e3	3.00e4	3.27e4	5.52e4	-	-	5.83e2	-	-	1.18e4	1.63e5
avg.(s)	4.78e3	5.04e3	1.10e4	4.20e3	2.42e4	2.28e4	4.79e2	4.06e3	2.10e4	1.38e4	9.41e3
short.(s)	2.80e3	1.73e2	6.34e2	9.60e1	1.42e4	1.47e3	6.79e1	1.57e3	2.10e4	4.51e3	1.28e3
long.(s)	6.98e3	1.15e4	1.44e4	1.65e4	3.42e4	4.42e4	1.87e3	6.54e3	2.10e4	1.11e4	3.38e4
%sol.(s)	100	100	100	100	40	40	100	40	20	100	100
%imp.(s)	43	83	66	92	-	-	18	-	-	-6	94

Table 6.2: The comparison between our approach and parallel portfolio search shown in Table 6.1 (PPS) on 32 cores.

[7]For instance, Gecode [186] supports executing several branchers in order of creation, and these branchers represent a branching. More specifically, search branches first on a group of variables are assigned to the first brancher, and then on the other group of variables assigned to the second brancher, etc.

Table 6.2 summarizes the results of the experiments which compares our single-solver-based portfolio approach with the portfolio search shown in Table 6.1 on 32 processors. We denote the number of variables of the CSP model of the instance by (#vars.) and execution time of PPS by (pps). Besides, the results of our approach are also given in the table, including the average time (avg.), shortest time (short.), longest time (long.), success rate of our approach for the problems (%sol.) in five consecutive runs, and improvement rate of execution time on average (%imp.) with respect to PPS. The CPU time are in seconds and a "-" sign indicates that the program was still running without solution after about 6.05e5 seconds (one week). The parameter settings of the solver are as follows: The first search strategy for k variables and the second search strategy for the rest of the variables are dom/wdeg and ABS, respectively. The inner restart strategy is set to Luby's restart strategy with scale factor 500 and base 1000, while the outer restart strategy employs periodically restarting with the constant cutoff which is a 1800-second time limit. (One may use the number of backtracks as well.)

For Magic Squares, we start with instance order 24 since the instances smaller than it can be easily solved by the PPS. We see that our technique on average outperformed the PPS for all the instances of Magic Squares in Table 6.2. The worst-case running times of our technique for order 25 and 26 are much shorter than the PPS. For order 27, the best-case of our technique shortened the runtime by three orders of magnitude, and we were able to obtain a solution for the order 28 and 29, which was not able to be solved by the PPS, in reasonable execution time. However, with a timeout set to one day, we also had three failures not to solve the order 28 and 29. The results for Sports Tournament Scheduling confirm the observations made on magic squares problem. Our technique provided a possibility to solve the unsolvable instances of sports tournament scheduling and to reduce the runtime on the solvable problem. Please note that the number of teams in sports tournament scheduling must be even. We next considered the Costas Arrays, the runtime of order 22 was reduced by orders of magnitude on average. Interestingly, our technique performed worse for order 21 and could not solve the larger instance than the PPS.

Discussion. Several possible reasons might cause that order 23 of the Costas Arrays remains unsolved to us and the performance degradation of order 22. First, though order 23 is not an open problem, it is still a computationally hard problem for constraint approach. In [29], the authors employed constraint-based local search on 8,192 cores of the IBM Bluegene/P supercomputer at Jülich to obtain a solution in approximately three minutes and could not solve it when using 1,024 cores. Second,

the model we used contains a large percentage of auxiliary variables, which slows down the resolution process. It is unclear whether or not auxiliary variables have an adverse effect on our approach. Third, the parameter settings for our nested restart strategies are likely not to be optimized. Fourth, according to [29], the runtime distribution of the large order instances of the Costas Arrays exhibits the exponential behavior (cf. [200]) instead of the heavy-tailed behavior. Thus, it is hard to say whether or not our parallel approach performs better on problems with the heavy-tailed behavior than the exponential behavior. Clearly, more work is needed to clarify these issues.

6.6 Related Work

As far as we are aware, the approach of Bordeaux *et al.* [22] is the most up-to-date work related to our proposed technique. The authors also aim at developing a massively parallel portfolio approach for CSP. Interestingly, however, all the techniques are presented in the context of SAT, and they conducted experiments on the SAT solver MiniSat. Their study shows that the most effective way to introduce randomization is to fix the variable ordering partially, where a small quantity q of variables are randomly selected, and then the variables are branched on first. But the way to set q is not given, and restart mechanism is also not discussed.

In the context of parallel branch-and-bound algorithm, which is also based on backtracking search, McCreesh *et al.* [133] pointed out that early diversity is critical to avoid a strong commitment to the most vulnerable heuristic advice. They observed that parallelism by introducing early diversity can often lead to superlinear speedups, and in this case, the workload balance is not the determining factor for performance.

Cire *et al.* [32] presented parallel restart search that executes Luby's restart as a whole in parallel. In their approach, the way of adding randomization is also to select a few variables uniformly at random. Similar to [22], the number of random variables is fixed for all different problems. For Magic Squares, the approach cannot ensure that order 16 can be solved every time by using Gecode and Intel Xeon 2.4 GHz computers with 32 cores, which cannot even compete with PPS.

6.7 Discussion and Conclusion

We have presented a simple technique for parallelizing portfolio search in CSP, which is based on diversifying the search near the root of search tree. The experimental results confirm that early diversity can effectively alleviate the strong commitment due

to the early decisions made by a search strategy, which is concluded by researches from different areas [22, 133, 32, 114, 91, 74, 73, 178]. And not only that, they demonstrate that our novel parallel stochastic portfolio is a promising approach to exploit large-scale parallel processing for CSP, verifying our theoretical expectations about its performance that we argued in Section 4.2.3. In addition, the design choices of our parallel stochastic portfolio search adds credence to the argument of [32] that "parallel algorithm design should not be performed independently of the underlying sequential algorithm".

Finally, we believe that we can do much better by tuning the parameter settings of the restart strategies and the restart strategies themselves. However, this may not lead to general improvements in our parallel stochastic portfolio approach, and we believe this will be improved when introducing a new and better adaptive restart policy in the future.

Chapter 7

Towards Parallel Constraint Solving by Hypertree Decomposition

As shown in the previous chapters, parallelism is a promising way to enhance the performance of constraint solving. In this chapter we explore the use of hypergraph decomposition to distribute constraints to parallel processors for exploiting parallel constraint solving. Besides, we explain why and how the hypergraph decomposition can be employed to relatively evenly distribute workload for parallel constraint solving.

This chapter is a revised version of our previous works [120, 121] and organized as follows. The introduction of decomposition methods and related notions are presented in Section 7.1 and Section 7.2, respectively. Section 7.3 describes the new method *det-k-CP* in detail, and then analyzes its time complexity. In Section 7.4, we present our experimental results. Finally, we conclude in Section 7.5.

7.1 Introduction

Many *NP-complete* and *NP-hard* problems can be solved in polynomial time if the corresponding hypergraph has the bounded hypertree-width, which indicates that the original intractable problem can be divided into a number of tractable subproblems [81]. In addition, the tree structure for a constraint network implies that each node of the tree decomposition can be solved simultaneously, which makes us naturally think of utilizing parallel computing to solve constraint sanctification problem (CSP). In other words, the acyclic structure of constraint networks implies that the given CSP $\mathcal{P}$ is tractable and parallelizable [75, 76, 78]. Several decomposition methods have been developed to transform cyclic constraint networks to acyclic ones, although these methods apply to different types of graph for the given constraint

network. For example, *join-tree-clustering* transforms the primal graph of the given constraint network into the equivalent acyclic network [40]. *Cycle-cutset decomposition* [42] also works on the primal graph by removing the vertexes that prevent the hypergraph to be acyclic. Some decomposition methods (e.g., *hinge decomposition* [85], *hypertree decomposition* [79, 81]), on the other hand, use the hypergraph as its input, and the output of these methods is at least in accord with the conditions for hypertree decomposition defined in [79].

Nevertheless, the decomposition methods, which have been proposed in the literature during the last decades, aim at obtaining as small hypertree width as possible for the hypergraph, because the smaller width of a hypertree decomposition we obtained, the faster the original CSP problem can be solved [81]. Moreover, previous structural decomposition methods, such as *det-k-decomp*, which is the most generalized decomposition method so far [77], cannot ensure a relatively even distribution of constraints based on our observation of results after running *det-k-decomp*. The algorithm *det-k-decomp* only guarantees the greatest node width of the decomposition tree is *k*, and fairly often, the width of most nodes is far less than *k*. This characteristic of *det-k-decomp* impedes its application in parallel constraint solving.

In the following sections, we will present a new decomposition *det-k-CP* method with stochastic search procedure for parallel constraint solving. The goal is to provide a mapping algorithm for parallel constraint solving. The idea behind *det-k-CP* is to utilize the property of the dual graph that a redundant arc can be removed between two nodes of the graph if there is an alternative path that ensures the two nodes still being connected. After applying *det-k-CP*, the hypertree structure of a constraint network can be used to parallelize the constraint solving.

7.2 Preliminaries

Any constraint network can be graphically represented by a *hypergraph*. A hypergraph $\mathcal{H}$ is a tuple (V, E), where V is a set of vertexes and E is a set of *hyperedges*. A hyperedge of a hypergraph is composed of two or more vertexes, which makes hyperedges fundamentally different from normal edges in a graph. Any constraint in a given constraint network corresponds to a hyperedge in a hypergraph, and the variables of a constraint can be seen as vertexes of a hyperedge.

A *hypertree* of a hypergraph $\mathcal{H}$ is a triple (T, χ, λ), where $T = (V_T, E_T)$ is a tree, χ and λ are labeling functions. We denote a set of variables for a given node ($node_i$) in a hypertree by v_i. Therefore, $v_i = \chi(node_i)$ and $v_i \subseteq 2^{vertexes(\mathcal{H})}$, where $vertexes(\mathcal{H})$ are

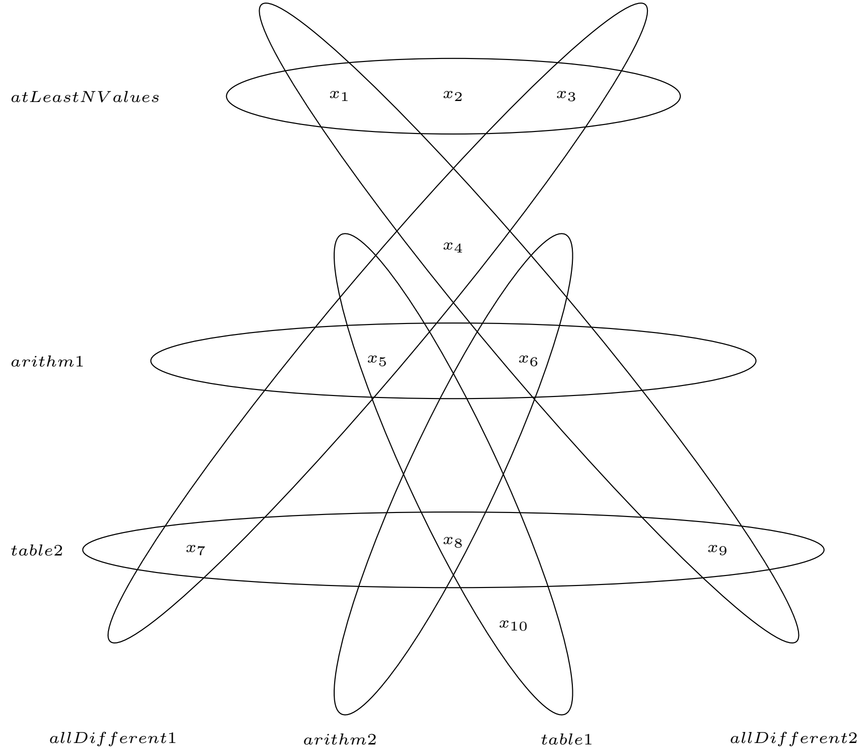


Figure 7.1: The hypergraph for the constraint network. (Figure adapted from [80] and reproduced from [120].)

vertexes of hypergraph $\mathcal{H}$. Similarly, we denote a set of edges of $node_i$ by e_i. Therefore, $e_i = \lambda(node_i)$ and $e_i \subseteq 2^{edges(\mathcal{H})}$, where $edges(\mathcal{H})$ are the hyperedges of hypergraph $\mathcal{H}$. By $root(T)$ we denote the root of a tree T, for every $p \in V_T$, let T_p denote the subtree of T with root p.

The *width of a hypertree* is the maximum number of hyperedges among the nodes of it, which is given by $hw(T) = max \mid \lambda(node_i) \mid$. *Hypertree decomposition* is a procedure that converts a hypergraph into a hypertree. In order to demonstrate hypertree decomposition on a given constraint network, assume we have a simple problem over a set of variables $\{x_1, \ldots, x_{10}\} \subseteq X$ modeled by the following constraints:

- *allDifferent1*(x_3, x_4, x_5, x_7)
- *allDifferent2*(x_1, x_4, x_6, x_9)
- $atLeastNvalues(x_1, x_2, x_3)$
- $arithm2(x_6, x_8)$
- $table1(x_5, x_8, x_{10})$
- $table2(x_7, x_8, x_9)$
- $arithm1(x_5, x_6)$

The hypergraph for this constraint network is depicted in Figure 7.1, where the variables x_i, $i \in \{1, \ldots, 10\}$ are the vertexes, while the edges are represented by the

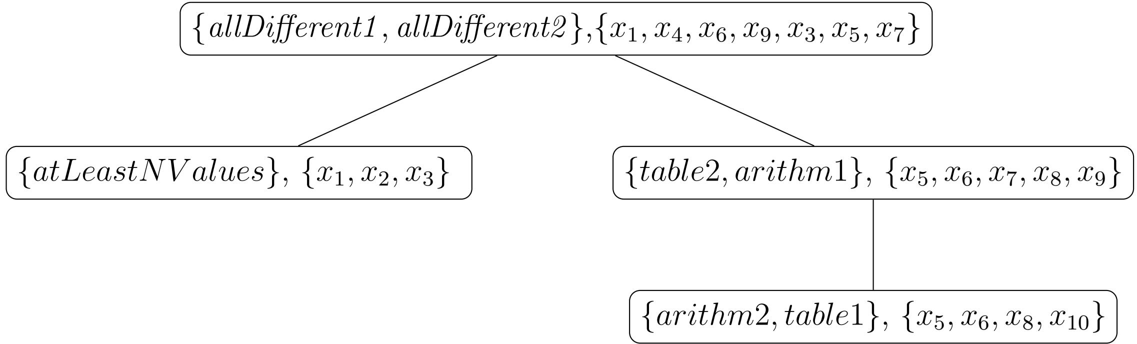


Figure 7.2: Hypertree decomposition for the hypergraph of Figure 7.1. (Figure adapted from [80] and reproduced from [120].)

enclosing ellipses. Figure 7.2 illustrates a possible hypertree decomposition of this hypergraph. Note that the hypertree decomposition (Figure 7.2) can also be viewed as a *dual graph* for the hypergraph (Figure 7.1). The nodes of a dual graph consist of a set of hyperedges of the corresponding hypergraph, and an edge of the dual graph is due to existing shared variables between two nodes of the dual graph. Formally, $\mathcal{H}^{dual}$ for a given $\mathcal{H}$ can be represented as a tuple (S, E) in which $S = \{s_1, ..., s_i, ..., s_j\} \subseteq$ $edges(\mathcal{H})$ and $\forall e \in E = edges(\mathcal{H}^{dual}) : s_i \cap s_j = e \Leftrightarrow var(s_i) \cap var(s_j) \neq \emptyset$.

Gottlob *et al.* [79, 81] defined four conditions, which must be satisfied by a hypertree after hypertree decomposition:

1. Every hyperedge of the hypergraph is contained in at least one node of the hypertree, which can be mathematically expressed as:

$$\forall_{e \in edges(H)}, \exists p \in vertexes(T) : e \subseteq \chi(p) \tag{7.1}$$

2. Nodes of a hypertree that contain the same vertex in the hypergraph form a subtree of the hypertree. For all $v \in vertices(H)$, the set

$$\{p \in vertexes(T) \mid v \in \chi(p)\} \tag{7.2}$$

induces a connected subtree of T (This is also called connectedness property [40]).

3. For any node of the hypertree, the vertexes of χ are included in the vertexes of λ, given by:

$$\forall_{p \in vertexes(T)} : \chi(p) \subseteq vertexes(\cup\lambda(p)) \tag{7.3}$$

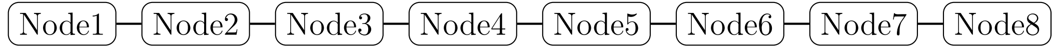


Figure 7.3: A constraint network is divided into eight parts. An edge between two nodes is due to the shared variables. (Figure reproduced from [120].)

4. For a vertex of a hypergraph in node p of the hypertree, if the vertex is included in both $\lambda(p)$ and $\chi(T_p)$, where $\chi(T_p)$ stands for the subtree of T rooted at p, this vertex must also be included in the $\chi(p)$, which can be expressed as:

$$\forall_{p \in vertexes(T)} : vertexes(\cup\lambda(p)) \cap \chi(T_p) \subseteq \chi(p) \tag{7.4}$$

7.3 The algorithm *det-k-CP*

In this section, we present our new algorithm *det-k-CP* that is designed to decompose a hypergraph for parallel constraint solving. The k of *det-k-CP* denotes the number of nodes in the decomposition tree. Roughly speaking, the goal of *det-k-CP* is to decompose a given constraint network $\mathcal{N}$ to a degenerate tree in which each internal node has exactly one child. The solutions of $\mathcal{N}$ can be found in time linear after each node is solved independently because *an acyclic constraint network can be solved efficiently* [40]. For example, Figure 7.3 depicts a target degenerate decomposition tree with eight nodes decomposed by *det-k-CP* for a multi-core processor with eight cores.

For a given ordering of nodes of a degenerate decomposition tree T generated by *det-k-CP*, there is an edge between two nodes because there exist shared variables between two nodes. Additionally, we only keep the edges between two adjacent nodes and eliminate the edges between any pair of non-adjacent nodes.[1] The mechanism of elimination of *det-k-CP*, which guarantees the decomposition tree is equivalent to the original one, is based on the property of which any edge on a circuit formed by common shared variables of a dual graph can be removed without changing the set of all solutions for the constraint network [40]. For instance, in Figure 7.2, the edge between the root node and the right leaf node caused by shared variables (x_5, x_6) is inexistent because there are shared variables (x_5, x_6) between the root node and its child node, as well as the right leaf node and its parent node respectively. The positional relationship between two nodes in T is either adjacent or non-adjacent; thus a pair of nodes in T can be denoted as (N_i, N_j) for non-adjacent or (N_p, N_{p+1})

[1]Please note that the verb "eliminate" does not mean an edge is deleted, it means that we can ignore the join selection for the nodes connected by this edge.

for adjacent respectively, where $|i-j| \geq 2$, $0 \leq i < j \leq k$ and $p \in \{i, ..., j-1\}$. A decomposed graph T after decomposition by *det-k-CP* must meet the following two conditions:

$$\forall_{p \in \{i,\ldots,j-1\}}: \ \chi(N_i) \cap \ \chi(N_j) \subseteq \chi(N_p) \cap \chi(N_{p+1}) \tag{7.5}$$

$$\forall_{N_i \in T}: \ \cup\, \chi(N_i) = \chi(\mathcal{H}), and \ \cup\, \lambda(N_i) = \lambda(\mathcal{H}) \tag{7.6}$$

where the first condition means that the shared variables between any pair of nodes (N_p, N_{p+1}) must contain the shared variables between (N_i, N_j), $i \leq p \leq j$, whereas the second condition ensures that the union of vertexes on each node of T is equal to the set of the vertexes of the original hypergraph $\mathcal{H}$, and the union of edges on each node of T is equal to the set of the edges of $\mathcal{H}$. In other words, the decomposition method *det-k-CP* does not lose any constraint or variable.

Having these two conditions we can easily validate whether a given decomposition tree is successfully decomposed by *det-k-CP*. Besides, if a decomposition tree satisfies the conditions for *det-k-CP*, it must meet the four conditions defined in [79] for hypertree decomposition (see Section 7.2).

Proposition 1. *A degenerate decomposition tree T of a hypergraph H generated by det-k-CP is a hypertree decomposition of the hypergraph H.*

Proof of Proposition 1. To prove this proposition, we are going to check and confirm the four conditions of *hypertree decomposition* one by one.

i Since *det-k-CP* does not remove any constraint, as mentioned in the second condition of *det-k-CP*, for every constraint e in the hypergraph H, we can find a node p in the degenerate decomposition tree T, where $\chi(p)$ contains e.

ii The second condition of *hypertree decomposition* ensures that all nodes that share a common vertex v of H induce a connected subtree of T. To prove it by contradiction, we assume there is a vertex v in T, where all nodes that contain v cannot induce a subtree of T. Therefore, in that case, these nodes result in a circuit which indicates there exist edges induced by a set of vertexes that cannot be eliminated by an alternative edge. This is contradicted by the first condition of *det-k-CP* in which the shared variables between any pair of adjacent nodes must contain the shared variables between non-adjacent nodes.

iii Because *det-k-CP* does not remove any variable (vertex), for any node p in T, $\chi(p) = (vertexes)(\cup\lambda(p))$, which satisfies $\chi(p) \subseteq (vertexes)(\cup\lambda(p))$.

Algorithm 5: *det-k-CP($\mathcal{N}$,k)*

Input: A Constraint Network $\mathcal{N}$, and the desired number of nodes k.
Output: A degenerate hypertree with k nodes

```
1  Set list_LN = a list which contains sorted constraints of N based on weight ;
2  Set i_size = the size of list_LN ;
3  Initialize an array array_Nodes with k nodes ;
4  for i ← 1 to i_size do
5  |  add list_LN[i] into array_Nodes[i%k] ;
6  end
7  while true do
8  |  getPotentialSolution(array_Nodes);
9  |  if array_Nodes pass test conditions (1) and (2) then
10 |  |  break;
11 |  end
12 |  Swap nodes in array_Nodes;
13 end
14 return array_Nodes;
```

iv For a given node p in T, $(vertexes)(\cup\lambda(p))\cap\chi(T_p) = (vertexes)(\cup\lambda(p)) = \chi(p)$, satisfying the fourth condition of *hypertree decomposition*. □

Algorithm 5 We are now going to explain the *det-k-CP* in more detail. In order to relatively evenly distribute workloads, line 1 of Algorithm 5 sorts the constraints based on its computational requirements (weight) that could perhaps depend on many factors (e.g., the time complexity of constraint propagator used by the constraint solver, the number of variables of the constraint, and the range of each of these variables, etc.). Then, after the sorting procedure, the constraints are inserted into an array ($array_Nodes$) with length k in turn in line 5 so that each node of the array contains the same amount of workload. Algorithm 5 runs into the loop in lines 7-13 until a qualified solution is found. Since sometimes a qualified solution cannot be obtained in one iteration, we use the heuristic for a swap procedure in line 12 of Algorithm 5. The heuristic has many choices, for example, random exchange, switching two nodes that have the fewest and the most number of constraints, or exchanging nodes based on the permutation in lexicographic order of the indexes of $array_Nodes$ in turn. For instance, if the length of $array_Nodes$ is 4, we might first use the permutation (0,1,2,3), then (0,1,3,2) and so on.

Algorithm 6 Let us now consider the function *getPotentialSolution* (called in Algorithm 5, line 8) defined by Algorithm 6, which exhaustively invokes Algorithm 7 for all the edges between non-adjacent nodes. To this aim, the starting point of the

Algorithm 6: *getPotentialSolution(array_Nodes)*

```
   Input: array_Nodes
   Output: A potential solution
1  Set i_len = the length of array_Nodes ;
2  Set i_start = i_len-3 ;
3  while i_start ≥ 0 do
4  |   for i_end ← i_start + 2 to i_len − 1 do
5  |   |   eliminateEdge(i_start, i_end, array_Nodes) ;
6  |   end
7  |   Set i_start = i_start − 1 ;
8  end
```

non-adjacent edge (i_start) is specified as the third to last index in line 2 of Algorithm 6, which should be the first disconnected node with the last node (with index $i_len - 1$). Then, it is decremented on each iteration until it reaches the first index of the array in the outer loop, and the ending point of the non-adjacent edge is initialized as $i_end = i_start + 2$, then the inner loop continues to iterate to the end of the array ($i_len - 1$).

Algorithm 7 The function *eliminateEdge* plays an important role in *det-k-CP*. For two non-adjacent nodes N_{i_start} and N_{i_end}, Algorithm 7 might add constraints to the nodes between N_{i_start} and N_{i_end} so that a potential edge between N_{i_start} and N_{i_end} could be covered. In line 2 of Algorithm 7, $list_2Eliminated$ is set to all shared variables between non-adjacent nodes N_{i_start} and N_{i_end}. For each edge, which exists due to shared variables between N_{i_start} and N_{i_end}, *eliminateEdge* checks whether or not every edge between adjacent nodes N_i and N_j, $i_{start} \leq i \leq i_{end} - 1,\ j = i + 1$, contains all variables that are also included in the input edge between nodes N_{i_start} and N_{i_end}, as shown in line 5 of Algorithm 7. If an edge between N_i and N_j contains all the shared variables that are also included in the edge between N_{i_start} and N_{i_end} (line 5), then the *for* loop runs into the next iteration for the next edge between N_{i+1} and N_{j+1}; otherwise, line 6 removes all the shared variables of N_i and N_j on $list_2Eliminated$.

The function $getMinimumSetConstraints4Share$, which will be presented in Algorithm 8, returns the minimum number of constraints that covers all the variables on the list $list_2Eliminated$. In lines 8-12, we loop through all constraints obtained by Algorithm 8, if the constraint is not contained in N_j, the constraint will be added into N_j. By doing so, the edges between non-adjacent nodes N_{i_start} and N_{i_end} can be eliminated since the shared variables between N_{i_start} and N_{i_end} are now covered

Algorithm 7: *eliminateEdge(i_start, i_end, array_Nodes)*

```
   Input: i_start, i_end, array_Nodes
1  for i ← i_start to i_end − 1 do
2  |  Set list_2Eliminated = getSharedVariables(i_start, i_end,
   |   array_Nodes) ;
3  |  Set j = i + 1 ;
4  |  Set list_SharedOnMainPath = getSharedVariables(i, j, array_Nodes);
5  |  if list_SharedOnMainPath.notContainsAll(list_2Eliminated) then
6  |  |  list_2Eliminated.removeAll(list_SharedOnMainPath);
7  |  |  Set list_2beAddedConstraints =
   |  |   getMinimumSetConstraints4Share(i, array_Nodes,
   |  |   list_2Eliminated);
8  |  |  foreach constraint cs ∈ list_2beAddedConstraints do
9  |  |  |  if array_Nodes[j] notContained cs then
10 |  |  |  |  add cs into array_Nodes[j];
11 |  |  |  end
12 |  |  end
13 |  end
14 end
```

by all the edges between adjacent nodes N_i and N_j, $i_{start} \leq i \leq i_{end} - 1,\ j = i + 1$.

Algorithm 8 The idea behind Algorithm 8 is to search the constraints covering as many variables as possible on (*list_2Eliminated*) for each variable in the difference of sets (*list_2Eliminated*) and (*list_SharedOnMainPath*) in each iteration, whereas the constraints themselves contain as few variables as possible. In line 12 of Algorithm 8, the covered variables on (*list_2Eliminated*) are added into the HashSet variable (*HashSet_IsCovred*), which helps avoid rechecking covered variables at the beginning of the *for* loop (lines 5-7).

So far we have discussed the detailed process of *det-k-CP*. The reason why we introduce randomness into the *Swap* method in line 12 of Algorithm 5 is that backtracking to the eliminated edges would cause a large number of loops due to the newly added constraints making the edges appear again. For instance, in Figure 7.3, we added some constraints onto node 4 due to the elimination process for the edge between nodes 3 and 5. Then the edge between nodes 4 and 7, which had been removed before, happened to be generated again. In this case, this generated edge would force us to add new constraints onto nodes 5 and 6. However, an edge between nodes 3 and 5 would be generated again. Consequently, we would fall into a repeated elimination process until the worst case happened, i.e., each node filled up with all the constraints

Algorithm 8: *getMinimumSetConstraints4Share(i,array_ Nodes,list_ 2Eliminated)*

```
   Input: endPoint, array_Nodes, list_2Eliminated
   Output: A set of constraints that has minimal number of variables to cover
           the list_2Eliminated
 1 Set list_SharedConstraints = add all constraints which covers the variables
    in the list_2Eliminated and remove duplicates;
 2 Initialize list_Result ;
 3 Initialize HashSet HashSet_IsCovred ;
 4 foreach Variables v ∈ list_2Eliminated do
 5 |   if HashSet_IsCovred contains v then
 6 |   |   continue;
 7 |   end
 8 |   foreach Constraint c ∈ list_SharedConstraints do
 9 |   |   add the constraint c' into list_Result, which covers the maximum
   |   |    number of variables in the list_2Eliminated and the constraint itself
   |   |    contains minimum number of variables;
10 |   |   foreach Variables v' ∈ c' do
11 |   |   |   if v' ∉ HashSet_IsCovred then
12 |   |   |   |   add v' into HashSet_IsCovred;
13 |   |   |   end
14 |   |   end
15 |   end
16 end
17 return list_Result;
```

of the given constraint network.

Now, we would like to analyze the time complexity of *det-k-CP*. If we combine the loops in Algorithm 6 with the loop in Algorithm 7 to form a triple-nested loop, the total number of iterations for the triple-nested loop can be calculated by the following recurrence relation:

$$n_3 = 2 \tag{7.7}$$

$$n_4 = 3 + 2 + n_3 \tag{7.8}$$

$$n_5 = 4 + 3 + 2 + n_4 \tag{7.9}$$

$$\vdots$$

$$n_k = (k-1) + (k-2) + \cdots + 2 + n_{k-1} = \frac{(k^2-k-2)}{2} + n_{k-1} \tag{7.10}$$

where n_k denotes the number of iterations for the triple-nested loop for k number of nodes of the target decomposition tree. A recurrence relation for $\{n_k\}$ can be obtained by considering the following: Whenever one node is added to a tree with

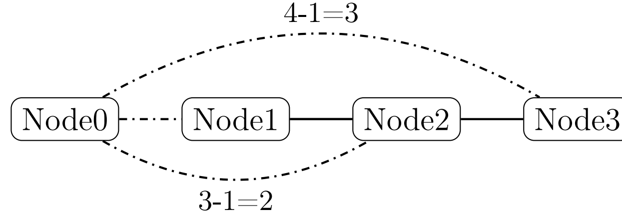


Figure 7.4: A new node is prepended to a degenerated tree with 3 nodes. (Figure reproduced from [120].)

$k-1$ nodes, new non-adjacent nodes are generated, and we have to eliminate these edges. The number of these edges can be summed by $(k-1)+(k-2)+\cdots+2$. For instance, as can be seen in Figure 7.4, when node 0 is added to the tree, the number of edges required to be eliminated is increased by 5. To obtain the explicit formula for this recurrence relation, we solve it with the initial conditions $n_3 = 2$, $n_2 = 0$ and $n_1 = 0$. The solution of the recurrence relation is $n_k = \frac{k^3-7k+6}{6}$, which means the number of loops of the triple-nested loop is exactly $\frac{k^3-7k+6}{6}$.

At each iteration of the triple-nested loop, in lines 2 to 13 of Algorithm 7, the number of executions can be bounded by the number of constraints (N_c) plus the complexity of method *getMinimumSetConstraints4Share*, denoted by $\mathcal{O}(N_c) + \mathcal{O}(getMinimumSetConstraints4Share)$. The implementation of *getMinimumSetConstraints4Shar* (i.e., Algorithm 8) is bounded by $\mathcal{O}(N_v \cdot N_c)$, where N_v is the number of variables. Thus, Algorithm 6 is bounded by $\mathcal{O}(\frac{k^3-7k+6}{6} \cdot (N_v \cdot N_c + N_c))$. Algorithm 5, which can be viewed as the outermost loop of the entire algorithm, can be bounded by $\mathcal{O}(k \cdot (N_c - \frac{N_c}{k}))$. This is because the loop of Algorithm 5 terminates eventually when every node is filled with the whole constraint network. The overall time complexity is, therefore, $\mathcal{O}(k \cdot (N_c - \frac{N_c}{k}) \cdot \frac{k^3-7k+6}{6} \cdot (N_v \cdot N_c + N_c))$. Therefore, the asymptotic time complexity is $\mathcal{O}(k^4 \cdot N_c^2 \cdot N_v)$. Note that the runtime may be significantly smaller in practice since we take into account the worst cases for Algorithm 5 and 7.

7.4 Experimental Results

In this section, we present our experimental results of the algorithm *det-k-CP* when applied to the benchmark suite provided by Gottlob et al. used in [81]. Note that we do not compare *det-k-CP* with *det-k-decomp* because these two algorithms aim at two different decomposition targets, as mentioned before. However, this should not place an obstacle for us since the hypergraphs in the benchmarks are extracted

from practical industrial constraint satisfaction problems. All the experiments are set up on an iMac computer having an Intel i7-3770 CPU, 3.40GHz, with 8 GB 1600 MHz DDR3 and running under macOS Sierra version 10.12.5. The algorithms are implemented in Java under JDK version 1.8.0_131.

Table 7.1: Experimental results for *det-k-CP* of the benchmark suite from [81]. (Table reproduced from [120].)

	$k=4$				$k=8$				$k=16$			
Instance	T	Min	Max	σ^2	T	Min	Max	σ^2	T	Min	Max	σ^2
●adder_75	75	94	201	1501	309	47	180	1832	3.7s	24	162	1874
●adder_75	9	94	173	1053	134	47	160	1734	3.9s	23	162	1881
●adder_75	10	94	208	1815	192	47	207	3201	3.6s	24	217	3608
●adder_99	75	124	262	2341	406	62	236	3062	5s	31	207	3050
●adder_99	10	124	261	4430	242	62	204	2098	12s	31	303	7291
●adder_99	14	124	272	3050	263	62	264	5008	5.9s	31	84	5724
●bridge_50	59	113	211	1323	277	57	209	2517	3s	29	185	2032
●bridge_50	7	113	221	2760	165	56	207	3160	3s	28	149	2633
●bridge_50	9	113	227	1830	207	57	219	2841	3.3s	29	209	3017
●bridge_75	82	170	314	2934	511	85	294	4968	7.8s	43	267	4938
●bridge_75	14	169	303	2729	639	84	347	7046	9s	42	322	5869
●bridge_75	17	170	336	4065	402	85	321	6796	8s	43	332	8396
●bridge_99	172	224	420	5168	13	112	376	7484	16.7s	56	358	8983
●bridge_99	38	223	435	6166	782	111	430	16265	14s	55	387	13914
●bridge_99	33	224	442	6727	737	112	424	11258	17s	56	428	12717
●NewSystem2	54	50	102	341	159	25	88	387	1.13	53	86	450
●NewSystem2	5	50	91	249	81	25	116	723	1.5	12	110	1000
●NewSystem2	3	50	102	341	72	25	111	729	979	13	97	636
●NewSystem3	84	70	133	343	268	35	134	1046	15s	18	157	1709
●NewSystem3	5	69	145	1370	150	34	141	1824	3.9s	12	159	2242
●NewSystem3	7	70	149	899	115	35	141	1240	2m	18	146	1681
●NewSystem4	71	105	211	1467	391	53	185	1610	4.5s	27	166	1717
●NewSystem4	11	104	210	1629	328	52	235	5516	75s	26	264	3585
●NewSystem4	10	105	211	1467	286	53	219	3102	5s	27	228	3778
▲grid2D_40	97	200	332	2630	777	100	311	4949	13S	50	300	6002
▲grid2D_40	28	200	335	2755	695	100	320	8010	15S	50	302	6269
▲grid2D_40	31	200	338	3079	592	100	318	5297	14S	50	337	7957
▲grid2D_75	486	703	1149	29669	7.8s	352	1093	62397	3.7m	176	1057	77453
▲grid2D_75	277	703	1192	35509	9.4s	351	1149	66461	3.7m	175	1078	59942
▲grid2D_75	298	703	1186	35874	7.9s	352	1131	68936	3.5m	176	1154	91808
■s953	71	106	204	1299	313	53	195	2286	3s	27	172	1965
■s953	8	106	211	2756	225	53	217	2776	3.2s	26	194	3226
■s953	6.9	106	204	1298	170	53	205	2724	3.5s	27	218	3638

Continued on next page

Table 7.1 – continued from previous page

	$k = 4$				$k = 8$				$k = 16$			
Instance	T	Min	Max	σ^2	T	Min	Max	σ^2	T	Min	Max	σ^2
■s1494	78	164	335	3859	523	82	291	4489	6.6s	41	249	3938
■s1494	15	163	350	8079	380	81	345	11411	7.6s	40	348	13751
■s1494	17.5	164	348	4832	340	80	332	7630	7.6s	41	339	9316
■s5378	507	740	1461	70147	5.8s	370	1338	101431	3m	185	1220	103192
■s5378	209	739	1347	57433	13.2s	369	1642	153866	5.3m	184	1568	167284
■s5378	183	740	1461	70148	6.8s	370	1614	176602	3.8m	185	1565	178186

Table 7.1 reports the experimental results for *det-k-CP* of the benchmark suite from [81]. The symbols ●, ▲, and ■ denote the benchmark packages *DaimlerChrysler*, *Grid2D*, and *ISCAS89* as used in [81], respectively.[2] The capital letter "T" represents execution time in *ms* (unless other specified), and Min, Max stands for the minimal and maximal number of constraints among all nodes, respectively. The variance of the number of constraints of all nodes is denoted σ^2. We use three different background colors to denote three types of heuristics for the swap procedures used when decomposing the instances, including permutation, random exchange, and switching nodes according to the number of constraints.[3]

As shown in Table 7.1, the execution time of instances is affordable even for the largest instance *s5378* with 2,958 constraints and 2,993 variables. But, for these large-scale instances, the algorithm *det-k-CP* does not achieve one principal goal of decomposition, i.e., each node has relativity balanced workload distribution. The decomposition results for the medium-scale instances, such as *adder_ 75, bridge_ 75, NewSystem3, NewSystem4, and S953* with a number of constraints from around 400 to 700, might be suitable for parallel constraint solving. After applying *det-k-CP*, the node with the maximum number of constraints among these instances is slightly higher than 200, implying that the node for these instances can be solved in reasonable execution time. For example, instances *adder_ 75* and *NewSystem4* have 677 and 418 constrains, respectively. The largest nodes we obtain after decomposition in terms of the number of constraints are 201 and 210 for *adder_ 75* and *NewSystem4*, respectively. Besides, the relatively small variance of these instances indicates that

[2]We do not include small instances such as *adder_ 15, adder_ 25*, etc. from the benchmark suite because it turns out that these small constraint networks are solvable for mainstream constraint solvers and thus do not require parallel solving.

[3]The term *permutation* is explained in Algorithm 5.

the number of constraints distributed to nodes for these instances is not spread out from their mean.

The results also indicate that the method of heuristics has a significant impact on the outcomes of decomposition. In most cases, exchanging nodes by permutation order (as shown in white background color in Table 7.1) gets smaller variances, but with exceptions (e.g., *bridge_ 75* and *NewSystem4*). It should be noted that all instances in the benchmark suite are decomposable by both *det-k-CP* and by *det-k-decomp*. Remember that in the proof of Proposition 1, we have already shown that if a hypertree decomposition meets the two conditions of *det-k-cp*, then it also satisfies the four conditions of *det-k-decomp*. Besides, based on our observation, an instance that can be decomposed by *det-k-CP* implies the instance can also be decomposed by *det-k-decomp* and vice versa.

In summary, we can conclude that *det-k-CP* can decompose a given constraint network within a reasonable execution time except for very large instances (e.g., *s5378* with 2,958 constraints and 2,993 variables) and a big k (e.g., $k = 16$). For the application of parallel constraint solving, the algorithm can be applied to medium scale constraint networks with the number of constraints from around 400 to 700.

7.5 Conclusion and Future Work

We have presented the new algorithm *det-k-CP* to construct a degenerate decomposition tree for parallel constraint solving, and we have also evaluated *det-k-CP* by a benchmark suite from previous research *det-k-decomp*. Our results have shown that *det-k-CP* can evenly distribute a constraint network with around 400 to 700 constraints onto parallel processors. However, we believe that there is a lot potential to improve *det-k-CP*. For instance, the algorithm should take into consideration an estimate of the amount/complexity of computation for the constraints. Furthermore, local search methods such as tabu search can be employed to replace the existing stochastic strategies in Algorithm 5 for better decomposition outcomes. Besides, for balanced workload distribution, we can add constraints from other nodes to one node after decomposition while still preserving the decomposition tree. Finally, the key indicator of the value of this research depends on whether we can obtain speedup or even superlinear speedup when using *det-k-CP* for parallel constraint solving, which is to be researched in detail.

Chapter 8

Conclusions

Moore's Law has powered the information technology revolution since the 1960s, but unfortunately, the age of Moore's Law has almost come to an end. This change indicates that it is becoming more and more difficult to achieve increased uniprocessor performance. Hence we are forced to switch to parallel processors from uniprocessors. At the same time, we still have to cope with the challenge raised by the demand for solving larger and harder constraint satisfaction problems. Indeed, the constraints community endeavors to develop more efficient and sophisticated techniques, such as randomization, restarts, search heuristics, consistency algorithms, nogood learning, etc., to boost the performance of problem-solving for constraint solvers. However, exploiting parallelism is one of the few means that allows constraint solving still to benefit from the performance improvement of hardware. Thus, research on parallel constraint solving is an essential subfield in constraint programming. Much progress has been made on effectively utilizing parallelism for constraint programming, such as work-stealing for constraint solving [31], parallel constraint-based local search [28, 29], embarrassingly parallel search (EPS) method for constraint solving [170, 171, 173, 131], etc.

This dissertation analyzed the effectiveness of parallel constraint solving, with the focus on obtaining a first solution when solving computationally hard combinatorial problems. We have shown that a well-designed search space splitting method and constraint programming model can enable the EPS approach to solve some open instances of the social golfer problem that have not been solved by a sequential algorithm. We also observed superlinear speedups when solving these instances, which confirms our theoretical analysis. Besides, we examined two practical constraint optimization problems, including the traveling tournament problem with predefined venues and the talent scheduling problem. Our proposed constraint models outperformed the existing models on the same instances, and the EPS approach could always

attain better feasible solutions in terms of the optimal objective value by using more parallel processors. To explore the use of massively parallel processing, we proposed the parallel stochastic portfolio search, which is a simple and non-intrusive way to parallelize different incarnations of a sequential solver. When comparing the existing portfolio to our portfolio approach by solving the same constraint satisfaction problems using the same constraint models, our technique could solve harder and larger instances. The successes of our new parallel approaches are attributed to early diversity; i.e., some diversity early in the search introduced by parallelism can offset early mistakes caused by weak heuristic choices. Unlike the other techniques (e.g., limited discrepancy search) used to overcome early mistakes, the studied two parallel constraint solving approaches not only can visit more nodes simultaneously but also does not sacrifice the guarantee of completeness.

In this dissertation, we also attempted to explore a new paradigm to exploit parallelism for constraint programming. We presented a hypertree decomposition method that builds a degenerate decomposition tree for a given constraint network, in which each node of the decomposition tree possesses and executes a subset of constraints of the given constraint network. The usefulness of our proposed parallel technique depends on whether we can find an efficient way to join the results of each node.

Appendix A

The Solutions of Some SGP Instances

Week \ Group	1			2			3			4			5			6		
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	3	6	1	9	12	2	7	15	4	13	16	5	10	14	8	11	17
2	0	4	15	1	3	13	2	5	6	7	11	16	8	10	12	9	14	17
3	0	5	9	1	4	17	2	3	8	6	12	16	7	10	13	11	14	15
4	0	7	14	1	10	16	2	12	17	3	9	15	4	6	11	5	8	13
5	0	8	16	1	6	14	2	11	13	3	10	17	4	7	9	5	12	15
6	0	11	12	1	8	15	2	4	10	3	14	16	5	7	17	6	9	13
7	0	13	17	1	5	11	2	9	16	3	7	12	4	8	14	6	10	15

Table A.1: The solution for 6-3-8 transformed from the solution Shown in Table 5.6. (Table adapted from [124].)

Week \ Group	1			2			3			4			5			6		
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	3	6	1	4	7	2	5	9	8	12	15	10	13	16	11	14	17
2	0	4	15	1	3	14	2	6	11	5	7	13	8	9	16	10	12	17
3	0	5	8	1	9	17	2	3	10	4	11	12	6	13	15	7	14	16
4	0	7	17	1	11	15	2	8	14	3	12	16	4	9	13	5	6	10
5	0	9	12	1	6	16	2	13	17	3	7	11	4	8	10	5	14	15
6	0	10	14	1	8	13	2	7	12	3	9	15	4	6	17	5	11	16
7	0	11	13	1	5	12	2	4	16	3	8	17	6	9	14	7	10	15

Table A.2: A new non-isomorphic solution for the 6-3-8 instance. (Table adapted from [124].)

Group / Week	1				2				3				4				5				6			
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	4	8	12	1	5	16	20	2	6	17	21	3	7	9	13	10	14	18	22	11	15	19	23
2	0	5	9	23	1	6	11	14	2	7	12	18	3	10	19	20	4	15	17	22	8	13	16	21
3	0	6	15	16	1	7	10	21	2	8	14	19	3	4	18	23	5	11	13	22	9	12	17	20
4	0	7	19	22	1	8	17	23	2	5	10	15	3	11	12	16	4	9	14	21	6	13	18	20
5	0	10	13	17	1	9	15	18	2	4	11	20	3	6	8	22	5	12	19	21	7	14	16	23
6	0	11	18	21	1	4	13	19	2	9	16	22	3	5	14	17	6	10	12	23	7	8	15	20

Table A.3: The solution for 6-4-7 transformed from the solution Shown in Table 5.7. (Table adapted from [124].)

Group / Week	1			2			3			4			5			6			7		
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2	1	4	7	2	5	10	3	8	11	6	13	16	9	14	19	12	17	20	15	18	21
3	1	5	18	2	8	15	3	17	19	4	14	20	6	9	11	7	12	13	10	16	21
4	1	6	19	2	7	16	3	5	12	4	13	17	8	14	21	9	10	18	11	15	20
5	1	8	17	2	4	11	3	9	16	5	13	21	6	14	18	7	10	20	12	15	19
6	1	9	15	2	13	19	3	4	21	5	8	20	6	10	17	7	11	18	12	14	16
7	1	10	14	2	12	18	3	6	15	4	8	16	5	7	19	9	13	20	11	17	21
8	1	11	13	2	14	17	3	18	20	4	9	12	5	15	16	6	7	21	8	10	19
9	1	12	21	2	6	20	3	7	14	4	10	15	5	9	17	8	13	18	11	16	19
10	1	16	20	2	9	21	3	10	13	4	18	19	5	11	14	6	8	12	7	15	17

Table A.4: A new non-isomorphic solution for the 7-3-10 instance. (Table reproduced from [124].)

Group / Week	1			2			3			4			5			6			7		
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	3	6	1	4	9	2	7	10	5	12	15	8	13	18	11	16	19	14	17	20
2	0	4	17	1	7	14	2	16	18	3	13	19	5	8	10	6	11	12	9	15	20
3	0	5	18	1	6	15	2	4	11	3	12	16	7	13	20	8	9	17	10	14	19
4	0	7	16	1	3	10	2	8	15	4	12	20	5	13	17	6	9	19	11	14	18
5	0	8	14	1	12	18	2	3	20	4	7	19	5	9	16	6	10	17	11	13	15
6	0	9	13	1	11	17	2	5	14	3	7	15	4	6	18	8	12	19	10	16	20
7	0	10	12	1	13	16	2	17	19	3	8	11	4	14	15	5	6	20	7	9	18
8	0	11	20	1	5	19	2	6	13	3	9	14	4	8	16	7	12	17	10	15	18
9	0	15	19	1	8	20	2	9	12	3	17	18	4	10	13	5	7	11	6	14	16

Table A.5: A new non-isomorphic solution for the 7-3-10 instance. (Table adapted from [124].)

Week \ Group	1			2			3			4			5			6			7		
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	3	6	1	4	9	2	5	10	7	11	12	8	15	18	13	16	19	14	17	20
2	0	4	18	1	12	15	2	9	14	3	17	19	5	7	13	6	10	16	8	11	20
3	0	5	17	1	8	10	2	7	20	3	9	13	4	12	19	6	14	15	11	16	18
4	0	7	14	1	5	16	2	8	17	3	12	18	4	10	20	6	9	19	11	13	15
5	0	8	16	1	14	19	2	3	11	4	7	15	5	9	18	6	13	20	10	12	17
6	0	9	15	1	3	20	2	12	16	4	8	13	5	11	14	6	17	18	7	10	19
7	0	10	13	1	7	18	2	15	19	3	8	14	4	11	17	5	6	12	9	16	20
8	0	11	19	1	13	17	2	4	6	3	7	16	5	15	20	8	9	12	10	14	18
9	0	12	20	1	6	11	2	13	18	3	10	15	4	14	16	5	8	19	7	9	17

Table A.6: A new non-isomorphic solution for the 7-3-10 instance. (Table adapted from [124].)

Week \ Player	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
2	1	2	3	4	5	6	7	8	2	1	4	3	6	5	8	7	3	4	1	2	7	8	5	6
3	1	2	3	4	5	6	7	8	3	4	1	2	7	8	5	6	6	5	8	7	2	1	4	3
4	1	2	3	4	5	6	7	8	4	3	2	1	8	7	6	5	8	7	6	5	4	3	2	1
5	1	2	3	4	5	6	7	8	5	6	7	8	1	2	3	4	2	1	4	3	6	5	8	7
6	1	2	3	4	5	6	7	8	6	5	8	7	2	1	4	3	4	3	2	1	8	7	6	5
7	1	2	3	4	5	6	7	8	7	8	5	6	3	4	1	2	5	6	7	8	1	2	3	4
8	1	2	3	4	5	6	7	8	8	7	6	5	4	3	2	1	7	8	5	6	3	4	1	2

Week \ Player	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
2	4	3	2	1	8	7	6	5	5	6	7	8	1	2	3	4	6	5	8	7	2	1	4	3
3	8	7	6	5	4	3	2	1	2	1	4	3	6	5	8	7	4	3	2	1	8	7	6	5
4	5	6	7	8	1	2	3	4	6	5	8	7	2	1	4	3	7	8	5	6	3	4	1	2
5	6	5	8	7	2	1	4	3	7	8	5	6	3	4	1	2	3	4	1	2	7	8	5	6
6	7	8	5	6	3	4	1	2	3	4	1	2	7	8	5	6	8	7	6	5	4	3	2	1
7	3	4	1	2	7	8	5	6	8	7	6	5	4	3	2	1	2	1	4	3	6	5	8	7
8	2	1	4	3	6	5	8	7	4	3	2	1	8	7	6	5	5	6	7	8	1	2	3	4

Week \ Player	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8
1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
2	7	8	5	6	3	4	1	2	8	7	6	5	4	3	2	1
3	5	6	7	8	1	2	3	4	7	8	5	6	3	4	1	2
4	3	4	1	2	7	8	5	6	2	1	4	3	6	5	8	7
5	8	7	6	5	4	3	2	1	4	3	2	1	8	7	6	5
6	2	1	4	3	6	5	8	7	5	6	7	8	1	2	3	4
7	4	3	2	1	8	7	6	5	6	5	8	7	2	1	4	3
8	6	5	8	7	2	1	4	3	3	4	1	2	7	8	5	6

Table A.7: A solution of 8-8-9 expressed by groups. (Table adapted from [124].)

Bibliography

[1] Alejandro Aguado. A 10 days solution to the social golfer problem. *Math games: Social Golfer problem. MAA Online*, 2004.

[2] Ö Akgün, Ian Miguel, and Chris Jefferson. Refining portfolios of constraint models with conjure. In *Doctoral Programme Proceedings, The 16th International Conference on Principles and Practice of Constraint Programming (CP 2010)*, pages 1–6, 2010.

[3] Kenzi Akiyama and Chihiro Suetake. On projective planes of order 12 with a collineation group of order 9. *Australasian J. Combinatorics*, 43:133–162, 2009.

[4] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a lazy portfolio approach for constraint solving. *TPLP*, 14(4-5):509–524, 2014.

[5] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A multicore tool for constraint solving. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 232–238, 2015.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):229–246, 2016.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, pages 483–485, 1967.

[8] Alejandro Arbelaez and Philippe Codognet. A GPU implementation of parallel constraint-based local search. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 648–655, 2014.

[9] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Online heuristic selection in constraint programming. In *Proceedings of the 4th International Symposium on Combinatorial Search*, 2009.

[10] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, pages 20–38, 2019.

[11] Fahiem Bacchus and Adam J. Grove. On the forward checking algorithm. In *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, pages 292–308, 1995.

[12] Simeon Ball. *Finite geometry and combinatorial applications*, volume 82. Cambridge University Press, 2015.

[13] Nicolas Barnier and Pascal Brisset. Solving the kirkman's schoolgirl problem in a few seconds. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pages 477–491, 2002.

[14] JG Benadé, AP Burger, and JH van Vuuren. The enumeration of k-sets of mutually orthogonal latin squares. In *Proceedings of the 42th Conference of the Operations Research Society of South Africa, Stellenbosch*, pages 40–49, 2013.

[15] Frédéric Benhamou and Laurent Granvilliers. Continuous and interval constraints. In *Handbook of Constraint Programming*, chapter 16, pages 571–603. Elsevier, 2006.

[16] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[17] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3, pages 29–83. Elsevier, 2006.

[18] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993.*, pages 108–113, 1993.

[19] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The tractability of global constraints. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 716–720, 2004.

[20] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 398–404, 1997.

[21] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[22] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 443–448, 2009.

[23] Raj Chandra Bose, Sharadchandra S Shrikhande, and Ernest T Parker. Further results on the construction of mutually orthogonal latin squares and the falsity of euler's conjecture. *Canadian Journal of Mathematics*, 12:189–203, 1960.

[24] Raj Chandra Bose and Sharadchandra Shankar Shrikhande. On the falsity of euler's conjecture about the non-existence of two orthogonal latin squares of order 4t+ 2. *Proceedings of the National Academy of Sciences of the United States of America*, 45(5):734, 1959.

[25] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150, 2004.

[26] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[27] Federico Campeotto, Alessandro Dal Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of gpus in constraint solving. In *Practical Aspects of Declarative Languages - 16th International Symposium,*

PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings, pages 152–167, 2014.

[28] Yves Caniou, Philippe Codognet, Daniel Diaz, and Salvador Abreu. Experiments in parallel constraint-based local search. In *Evolutionary Computation in Combinatorial Optimization - 11th European Conference, EvoCOP 2011, Torino, Italy, April 27-29, 2011. Proceedings*, pages 96–107, 2011.

[29] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-scale parallelism for constraint-based local search: the costas array case study. *Constraints*, 20(1):30–56, 2015.

[30] TCE Cheng, JE Diamond, and BMT Lin. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications*, 79(3):479–492, 1993.

[31] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 226–241, 2009.

[32] André A. Ciré, Serdar Kadioglu, and Meinolf Sellmann. Parallel restarted search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 842–848, 2014.

[33] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *Stochastic Algorithms: Foundations and Applications, International Symposium, SAGA 2001 Berlin, Germany, December 13-14, 2001, Proceedings*, pages 73–90, 2001.

[34] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[35] Paul R. Cooper and Michael J. Swain. Arc consistency: Parallelism and domain dependence. *Artificial Intelligence*, 58(1-3):207–235, 1992.

[36] Carlos Cotta, Iván Dotú, Antonio J. Fernández, and Pascal Van Hentenryck. Scheduling social golfers with memetic evolutionary programming. In *Hybrid*

Metaheuristics, Third International Workshop, HM 2006, Gran Canaria, Spain, October 13-15, 2006, Proceedings, pages 150–161, 2006.

[37] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, pages 1092–1097, 1994.

[38] CSPLib: A problem library for constraints, 1999. `http://www.csplib.org/`. Accessed 9 Jan 2020.

[39] Marialuisa J de Resmini. There exist at least three non-isomorphic s (2, 4, 28)'s. *Journal of Geometry*, 16(1):148–151, 1981.

[40] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.

[41] Rina Dechter. Tractable structures for constraint satisfaction problems. In *Handbook of Constraint Programming*, chapter 7, pages 209–244. Elsevier, 2006.

[42] Rina Dechter and Judea Pearl. The cycle-cutset method for improving search performance in ai applications. In *Third IEEE Conference on AI Applications*, pages 224–230. IEEE, 1987.

[43] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[44] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 207–223, 2016.

[45] Emir Demirovic, Geoffrey Chu, and Peter J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, pages 99–108, 2018.

[46] Iván Dotú and Pascal Van Hentenryck. Scheduling social golfers locally. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005,*

Prague, Czech Republic, May 30 - June 1, 2005, Proceedings, pages 155–167, 2005.

[47] Iván Dotú and Pascal Van Hentenryck. Scheduling social tournaments locally. *AI Communications*, 20(3):151–162, 2007.

[48] Thorsten Ehlers and Peter J. Stuckey. Parallelizing constraint programming with learning. In *Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings*, pages 142–158, 2016.

[49] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, pages 93–107, 2001.

[50] Boi Faltings. Distributed constraint programming. In *Handbook of Constraint Programming*, chapter 20, pages 699–729. Elsevier, 2006.

[51] Matteo Fischetti, Michele Monaci, and Domenico Salvagnin. Self-splitting of workload in parallel computation. In *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, pages 394–404, 2014.

[52] Filippo Focacci, François Laburthe, and Andrea Lodi. Local search and constraint programming. In *Handbook of Metaheuristics*, chapter 13, pages 369–403. Kluwer Academic Publishers, first edition, 2003.

[53] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, pages 77–92, 2001.

[54] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 29(1):24–32, 1982.

[55] Eugene C. Freuder and Alan K. Mackworth. Constraint satisfaction: An emerging paradigm. In *Handbook of Constraint Programming*, chapter 2, pages 13–27. Elsevier, 2006.

[56] Eugene C. Freuder and Richard J. Wallace. Generalizing inconsistency learning for constraint satisfaction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 563–571, 1995.

[57] Alan M. Frisch, Ian Miguel, and Toby Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming, Cork, Ireland, June 19-21, 2002. Selected Papers*, pages 15–30, 2002.

[58] Matteo Gagliolo and Jürgen Schmidhuber. Learning restart strategies. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 792–797, 2007.

[59] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 140–148, 2015.

[60] Fayez Gebali. *Algorithms and parallel computing.* Wiley & Sons, 2011.

[61] Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil C. A. Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *TPLP*, 18(5-6):725–758, 2018.

[62] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, chapter 10, pages 329–376. Elsevier, 2006.

[63] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 10, pages 329–376. Elsevier, 2006.

[64] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, pages 599–603, 2000.

[65] Nebras Gharbi. *On compressing and parallelizing constraint satisfaction problems.* PhD thesis, Artois University, 2015.

[66] Gael Glorian, Frédéric Boussemart, Jean-Marie Lagniez, Christophe Lecoutre, and Bertrand Mazure. Combining nogoods in restart-based search. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, pages 129–138, 2017.

[67] Solomon W Golomb and Leonard D Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[68] Carla P. Gomes, Cèsar Fernández, Bart Selman, and Christian Bessiere. Statistical regimes across constrainedness regions. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 32–46, 2004.

[69] Carla P. Gomes, Cèsar Fernández, Bart Selman, and Christian Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.

[70] Carla P. Gomes, Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, chapter 2, pages 89–134. Elsevier, 2008.

[71] Carla P. Gomes and Ashish Sabharwal. Exploiting runtime variation in complete solvers. In *Handbook of Satisfiability*, chapter 9, pages 271–288. IOS Press, 2009.

[72] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, pages 121–135, 1997.

[73] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[74] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 431–437, 1998.

[75] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 57–74, 2016.

[76] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Advanced parallel algorithms for acyclic conjunctive queries. Technical report, Technical Report DBAI-TR-98/18, `http://www.dbai.tuwien.ac.at/staff/gottlob/parallel.ps`, 1998.

[77] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[78] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[79] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[80] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences*, 66(4):775–808, 2003.

[81] Georg Gottlob and Marko Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[82] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, 2003.

[83] Laurent Granvilliers and Gaétan Hains. A conservative scheme for parallel interval narrowing. *Information Processing Letters*, 74(3-4):141–146, 2000.

[84] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[85] Marc Gyssens and Jan Paredaens. A decomposition methodology for cyclic databases. In *Advances in Data Base Theory, Vol. 2, Based on the Proceedings of the Workshop on Logical Data Bases, December 14-17, 1982, Centre d'études et de recherches de Toulouse, France*, pages 85–122, 1982.

[86] Youssef Hamadi. Optimal distributed arc-consistency. *Constraints*, 7(3-4):367–385, 2002.

[87] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[88] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[89] Warwick Harvey. CSPLib problem 010: Social golfers problem, 2002. `http://www.csplib.org/Problems/prob010`, Accessed 9 Jan 2020.

[90] Warwick Harvey and Thorsten Jan Winterer. Solving the MOLR and social golfers problems. In *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 286–300, 2005.

[91] William D Harvey. *Nonsystematic backtracking search.* PhD thesis, stanford university, 1995.

[92] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 607–615, 1995.

[93] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, sixth edition, 2019.

[94] Ruth Hoffmann, Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, Craig Reilly, Christine Solnon, and James Trimble. Observations from parallelising three maximum common (connected) subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th*

International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings, pages 298–315, 2018.

[95] Petra Hofstedt and Armin Wolf. *Einführung in die Constraint-Programmierung - Grundlagen, Methoden, Sprachen, Anwendungen.* eXamen.press. Springer, 2007.

[96] Holger H. Hoos and Edward P. K. Tsang. Local search methods. In *Handbook of Constraint Programming*, chapter 5, pages 135–167. Elsevier, 2006.

[97] Benny Van Houdt. Randomized work stealing versus sharing in large-scale systems with non-exponential job sizes. *CoRR*, abs/1810.13186, 2018.

[98] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2318–2323, 2007.

[99] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[100] Willem Hulsink, Dick Manuel, and Harry Bouwman. Clustering in ict: From route 128 to silicon valley, from dec to google, from hardware to content. *ERIM Report Series Reference No. ERS-2007-064-ORG*, 2007.

[101] Bilal Syed Hussain. CSPLib problem 054: N-queens. `http://www.csplib.org/Problems/prob054/`, Accessed 19 May 2019.

[102] Joey Hwang and David G. Mitchell. 2-way vs. d-way branching for CSP. In *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 343–357, 2005.

[103] Daisuke Ishii, Kazuki Yoshizoe, and Toyotaro Suzumura. Scalable parallel numerical CSP solver. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 398–406, 2014.

[104] Serdar Kadioğlu and Özgür Akgün. CSPLib problem 076: Costas arrays, 2001. `http://www.csplib.org/Problems/prob076`, Accessed 9 Jan 2020.

[105] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[106] Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman. Dynamic restart policies. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 674–681, 2002.

[107] Graham Kendall, Sigrid Knust, Celso C. Ribeiro, and Sebastián Urrutia. Scheduling in sports: An annotated bibliography. *Computers and Operations Research*, 37(1):1–19, 2010.

[108] Thomas P Kirkman. Note on an unanswered prize question. *Cambridge and Dublin Math. J*, 5:255–262, 1850.

[109] Lars Kotthoff and Neil C. A. Moore. Distributed solving through model splitting. *CoRR*, abs/1008.4328, 2010.

[110] Krzysztof Kuchcinski and Radoslaw Szymanek. *JaCoP Documentation.* Lund University, 2017. Available from `http://www.lth.se/jacop/`.

[111] Clement WH Lam, Larry Thiel, and Stanley Swiercz. The non-existence of finite projective planes of order 10. *Canadian Journal of Mathematics*, 41(6):1117–1123, 1989.

[112] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[113] Yat Chiu Law and Jimmy Ho-Man Lee. Global constraints for integer and set value precedence. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 362–376, 2004.

[114] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms.* John Wiley & Sons, 2009.

[115] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4):147–167, 2007.

[116] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[117] Jimmy H. M. Lee, Christian Schulte, and Zichen Zhu. Increasing nogoods in restart-based search. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 3426–3433, 2016.

[118] Hongbo Li and Zhanshan Li. A novel strategy of combining variable ordering heuristics for constraint satisfaction problems. *IEEE Access*, 6:42750–42756, 2018.

[119] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1-2):315–326, 2000.

[120] Ke Liu, Sven Löffler, and Petra Hofstedt. Hypertree decomposition: The first step towards parallel constraint solving. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2017, Unifying INAP, WFLP, and WLP, Würzburg, Germany, September 19-22, 2017, Revised Selected Papers*, pages 81–94, 2017.

[121] Ke Liu, Sven Löffler, and Petra Hofstedt. Using hypertree decomposition for parallel constraint solving. In Maximilian Eibl and Martin Gaedke, editors, *47. Jahrestagung der Gesellschaft für Informatik, Informatik 2017, Chemnitz, Germany, September 25-29, 2017*, volume P-275 of *LNI*, pages 615–622. GI, 2017.

[122] Ke Liu, Sven Löffler, and Petra Hofstedt. Solving the traveling tournament problem with predefined venues by parallel constraint programming. In *Mining Intelligence and Knowledge Exploration - 6th International Conference, MIKE 2018, Cluj-Napoca, Romania, December 20-22, 2018, Proceedings*, pages 64–79, 2018.

[123] Ke Liu, Sven Löffler, and Petra Hofstedt. Parallel stochastic portfolio search for constraint solving. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom 2019, Xiamen, China, December 16-18, 2019*, pages 697–704. IEEE, 2019.

[124] Ke Liu, Sven Löffler, and Petra Hofstedt. Social golfer problem revisited. In *Agents and Artificial Intelligence - 11th International Conference, ICAART 2019, Prague, Czech Republic, February 19-21, 2019, Revised Selected Papers*, pages 72–99, 2019.

[125] Ke Liu, Sven Löffler, and Petra Hofstedt. Solving the social golfers problems by constraint programming in sequential and parallel. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019*, pages 29–39, 2019.

[126] Ke Liu, Sven Löffler, and Petra Hofstedt. Solving the talent scheduling problem by parallel constraint programming. In *Artificial Intelligence Applications and Innovations - 15th IFIP WG 12.5 International Conference, AIAI 2019, Hersonissos, Crete, Greece, May 24-26, 2019, Proceedings*, pages 236–244, 2019.

[127] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[128] Rui Machado and Carsten Lojewski. The fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - R&D*, 23(3-4):125–132, 2009.

[129] Rui Machado, Vasco Pedro, and Salvador Abreu. On the scalability of constraint programming on hierarchical multiprocessor systems. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 530–535, 2013.

[130] Alan K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[131] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly parallel search in constraint programming. *Journal of Artificial Intelligence Research*, 57:421–464, 2016.

[132] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[133] Ciaran McCreesh and Patrick Prosser. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. *TOPC*, 2(1):8:1–8:27, 2015.

[134] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.

[135] Brendan D McKay, Alison Meynert, and Wendy Myrvold. Small latin squares, quasigroups, and loops. *Journal of Combinatorial Designs*, 15(2):98–119, 2007.

[136] Christopher Mears, Todd Niven, Marcel Jackson, and Mark Wallace. Proving symmetries by model transformation. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, pages 591–605, 2011.

[137] Rafael A. Melo, Sebastián Urrutia, and Celso C. Ribeiro. The traveling tournament problem with predefined venues. *Journal of Scheduling*, 12(6):607–622, 2009.

[138] Tarek Menouer, Mohamed Rezgui, Bertrand Le Cun, and Jean-Charles Régin. Mixing static and dynamic partitioning to parallelize a constraint programming solver. *International Journal of Parallel Programming*, 44(3):486–505, 2016.

[139] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, pages 228–243, 2012.

[140] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Parallelizing constraint programs transparently. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 514–528, 2007.

[141] Ian Miguel. CSPLib problem 038: Steel mill slab design, 2012. `http://www.csplib.org/Problems/prob010`, Accessed 19 May 2019.

[142] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[143] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *ECAI*, pages 651–656, 1988.

[144] Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper. Parallel discrepancy-based search. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 30–46, 2013.

[145] Thierry Moisan, Claude-Guy Quimper, and Jonathan Gaudreault. Parallel depth-bounded discrepancy search. In *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, pages 377–393, 2014.

[146] Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48(2):143–170, 1991.

[147] Danny Munera, Daniel Diaz, and Salvador Abreu. Towards parallel constraint-based local search with the X10 language. In *Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers*, pages 169–184, 2013.

[148] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A parametric framework for cooperative parallel local search. In *Evolutionary Computation in Combinatorial Optimisation - 14th European Conference, EvoCOP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, pages 13–24, 2014.

[149] T. Nguyen and Yves Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[150] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 590–605, 2014.

[151] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

[152] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

[153] José Carlos Ortiz-Bayliss, Hugo Terashima-Marín, and Santiago Enrique Conant-Pablos. A supervised learning approach to construct hyper-heuristics for constraint satisfaction. In *Pattern Recognition - 5th Mexican Conference, MCPR 2013, Querétaro, Mexico, June 26-29, 2013. Proceedings*, pages 284–293, 2013.

[154] José Carlos Ortiz-Bayliss, Hugo Terashima-Marín, and Santiago Enrique Conant-Pablos. Combine and conquer: an evolutionary hyper-heuristic approach for solving constraint satisfaction problems. *Artificial Intelligence Review*, 46(3):327–349, 2016.

[155] Anthony Palmieri, Jean-Charles Régin, and Pierre Schaus. Parallel strategies selection. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 388–404, 2016.

[156] Behrooz Parhami. *Introduction to parallel processing: algorithms and architectures*. Springer Science & Business Media, 2006.

[157] Ernest Tilden Parker. Construction of some sets of mutually orthogonal latin squares. *Proceedings of the American Mathematical Society*, 10(6):946–949, 1959.

[158] Jeff Parkhurst, John A. Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *2006 International Conference on Computer-Aided Design, ICCAD 2006, San Jose, CA, USA, November 5-9, 2006*, pages 67–72, 2006.

[159] Laurent Perron. Search procedures and parallelism in constraint programming. In *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, pages 346–360, 1999.

[160] Gilles Pesant. CSPLib problem 068: Traveling tournament problem with predefined venues (ttppv), 2009. `http://www.csplib.org/Problems/prob068`, Accessed 9 Jan 2020.

[161] Gilles Pesant. A constraint programming approach to the traveling tournament problem with predefined venues. *Practice and Theory of Automated Timetabling*, pages 303–316, 2012.

[162] David Poole and Alan K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents.* Cambridge University Press, second edition, 2017.

[163] Steven Prestwich. CSPLib problem 028: Balanced incomplete block designs, 2001. `http://www.csplib.org/Problems/prob010`, Accessed 9 Jan 2020.

[164] Steven Prestwich. Local search and backtracking vs non-systematic backtracking. In *AAAI 2001 Fall symposium on using uncertainty within computation*, pages 109–115, 2001.

[165] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. Available from `http://www.choco-solver.org`.

[166] Jean-Francois Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[167] Rolf S Rees and WD Wallis. Kirkman triple systems and their generalizations: A survey. In *Designs 2002*, pages 317–368. Springer, 2003.

[168] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 557–571, 2004.

[169] Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In *Handbook of Parallel Constraint Reasoning*, chapter 9, pages 337–379. Springer, 2018.

[170] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 596–610, 2013.

[171] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the embarrassingly parallel search for data centers. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 622–635, 2014.

[172] Andrea Rendl, Ian Miguel, Ian P. Gent, and Christopher Jefferson. Automatically enhancing constraint model instances during tailoring. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*, 2009.

[173] Mohamed Rezgui, Jean-Charles Régin, and Arnaud Malapert. Using cloud computing for solving constraint programming problems. In *First Workshop on Cloud Computing and Optimization, a conference workshop of CP 2014*. Citeseer, 2014.

[174] Christopher K Riesbeck and Roger C Schank. *Inside case-based reasoning*. Lawrence Erlbaum Associates, 1989.

[175] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in HPC systems: why and when? In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016.*, pages 889–898, 2016.

[176] Carl Christian Rolf. *Parallelism in Constraint Programming*. PhD thesis, Lund University, Sweden, 2011.

[177] Carl Christian Rolf and Krzysztof Kuchcinski. Parallel solving in constraint programming. In *MCC 2010: Third Swedish Workshop on Multi-Core Computing*, 2010.

[178] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[179] Olivier Roussel. *Description of ppfolio*. Lens Computer Science Research Lab, 2017. Available from `https://www.cril.univ-artois.fr/~roussel/ppfolio/`.

[180] Yongshao Ruan, Eric Horvitz, and Henry A. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *Principles and*

Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings, pages 573–586, 2002.

[181] Alvaro Ruiz-Andino, Lourdes Araujo, Fernando Sáenz-Pérez, and José J. Ruz. Parallel arc-consistency for functional constraints. In *Proceedings of the International Workshop on Implementation Technology for Programming Languages based on Logic, held in conjunction with the Joint International Conference and Symposium on Logic Programming, Manchester, UK, Saturday 20th June, 1998*, pages 86–100, 1998.

[182] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Pearson Education, third edition, 2010.

[183] Ashok Samal and Tom Henderson. Parallel consistent labeling algorithms. *International Journal of Parallel Programming*, 16(5):341–364, 1987.

[184] Christian Schulte. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, pages 41–57, 2000.

[185] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In *Handbook of Constraint Programming*, chapter 14, pages 495–526. Elsevier, 2006.

[186] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and programming with gecode.* Gecode Team, 2017. Available from `https://www.gecode.org/`.

[187] Meinolf Sellmann and Warwick Harvey. Heuristic constraint propagation–using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem. In *CPAIOR'02*. Citeseer, 2002.

[188] Charles Severance and Kevin Dowd. *High performance computing.* Rice University, 2012.

[189] Yuan Shi. Reevaluating Amdahl's law and Gustafson's law. *Computer Sciences Department, Temple University (MS: 38-24)*, 1996.

[190] Barbara Smith. CSPLib problem 039: The rehearsal problem, 2001. `http://www.csplib.org/Problems/prob039`, Accessed 9 Jan 2020.

[191] Barbara Smith. Constraint programming in practice: Scheduling a rehearsal. *Re-search Report APES-67-2003, APES group*, 2003.

[192] Barbara M. Smith. Modelling. In *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[193] Barbara M. Smith. Reducing symmetry in a combinatorial design problem. In *CPAIOR'01*, pages 351–359, April 2001. http://www.icparc.ic.ac.uk/cpAIOR01.

[194] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[195] Google AI team. *Google's OR-Tools*. Google LLC, 2019. Available from `https://developers.google.com/optimization/`.

[196] Hugo Terashima-Marín, José Carlos Ortiz-Bayliss, Peter Ross, and Manuel Valenzuela-Rendón. Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems. In *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, pages 571–578, 2008.

[197] Markus Triska and Nysret Musliu. An effective greedy heuristic for the social golfer problem. *Annals of Operations Research*, 194(1):413–425, 2012.

[198] Markus Triska and Nysret Musliu. An improved SAT formulation for the social golfer problem. *Annals of Operations Research*, 194(1):427–438, 2012.

[199] Roman Trobec, Bostjan Slivnik, Patricio Bulic, and Borut Robic. *Introduction to Parallel Computing - From Algorithms to Programming on State-of-the-Art Platforms*. Undergraduate Topics in Computer Science. Springer, 2018.

[200] Charlotte Truchet, Florian Richoux, and Philippe Codognet. Prediction of parallel speed-ups for las vegas algorithms. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 160–169, 2013.

[201] Peter van Beek. Backtracking search algorithms. In *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier, 2006.

[202] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[203] Toby Walsh. Depth-bounded discrepancy search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1388–1395, 1997.

[204] Toby Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 1172–1177, 1999.

[205] Toby Walsh. CSPLib problem 019: Magic squares and sequences, 2002. `http://www.csplib.org/Problems/prob019`, Accessed 9 Jan 2020.

[206] Toby Walsh. CSPLib problem 026: Sports tournament scheduling, 2002. `http://www.csplib.org/Problems/prob026`, Accessed 9 Jan 2020.

[207] Toby Walsh. Breaking value symmetry. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1585–1588, 2008.

[208] Wikipedia contributors. Transputer — Wikipedia, the free encyclopedia, 2019. [Online; accessed 11-August-2019].

[209] Feng Xie and Andrew J. Davenport. Massively parallel constraint programming for supercomputers: Challenges and initial results. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, pages 334–338, 2010.

[210] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[211] Xi Yun and Susan L. Epstein. A hybrid paradigm for adaptive parallel search. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 720–734, 2012.

[212] Alessandro Zanarini and Gilles Pesant. More robust counting-based search heuristics with alldifferent constraints. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, pages 354–368, 2010.

[213] Ying Zhang and Alan K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, SPDP 1991, 2-5 December 1991, Dallas, Texas, USA*, pages 394–397, 1991.

[214] Ying Zhang and Alan K. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. Technical Report 92-30, Department of Computer Science, The University of British Columbia, Vancouver, B.C. Canada, November 1992.

[215] Peter Zoeteweij and Farhad Arbab. A component-based parallel constraint solver. In *Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings*, pages 307–322, 2004.