TWO COMBINATORIAL OPTIMIZATION PROBLEMS AT THE INTERFACE OF COMPUTER SCIENCE AND OPERATIONS RESEARCH

BY

GIO K. KAO

B.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Sheldon Jacobson, Chair
Associate Professor Chandra Chekuri
Associate Professor Jeff Erickson
Professor Edward Sewell, Southern Illinois Univeristy Edwardsville
Dr. Laura Swiler, Sandia National Laboratories (non-voting)

Abstract

Solving large combinatorial optimization problems is a ubiquitous task across multiple disciplines. Developing efficient procedures for solving these problems has been of great interest to both researchers and practitioners. Over the last half century, vast amounts of research have been devoted to studying various methods in tackling these problems. These methods can be divided into two categories, heuristic methods and exact algorithms. Heuristic methods can often lead to near optimal solutions in a relatively time efficient manner, but provide no guarantees on optimality. Exact algorithms guarantee optimality, but are often very time consuming.

This dissertation focuses on designing efficient exact algorithms that can solve larger problem instances with faster computational time. A general framework for an exact algorithm, called the Branch, Bound, and Remember algorithm, is proposed in this dissertation. Three variations of single machine scheduling problems are presented and used to evaluate the efficiency of the Branch, Bound, and Remember algorithm. The computational results show that the Branch, Bound, and Remember algorithms outperforms the best known algorithms in the literature.

While the Branch, Bound, and Remember algorithm can be used for solving combinatorial optimization problems, it does not address the subject of post-optimality selection after the combinatorial optimization problem is solved. Post-optimality selection is a common problem in multi-objective combinatorial optimization problems where there exists a set of optimal solutions called Pareto optimal (non-dominated) solutions. Post-optimality selection is the process of selecting the best solutions within

the Pareto optimal solution set. In many real-world applications, a Pareto solution set (either optimal or near-optimal) can be extremely large, and can be very challenging for a decision maker to evaluate and select the best solution.

To address the post-optimality selection problem, this dissertation also proposes a new discrete optimization problem to help the decision-maker to obtain an optimal preferred subset of Pareto optimal solutions. This discrete optimization problem is proven to be NP-hard. To solve this problem, exact algorithms and heuristic methods are presented. Different multi-objective problems with various numbers of objectives and constraints are used to compare the performances of the proposed algorithms and heuristics.

To my parents for teaching me the importance of hard work and higher education

To the memory of my grandfather Prof. S.F. Kao for being my inspiration

To my grandmother Por Por for all her life lessons when I was a child

Acknowledgments

This dissertation would not have been possible without the help and support of countless people. I especially want to thank

- My adviser and mentor, Dr. Sheldon H. Jacobson for not just teaching me how to become a successful student and researcher, but also the many lessons on how to succeed in life. Without his guidance and his constant encouragement, I would not have finished.

- Dr. Edward C. Sewell for all of his guidance in helping me put together many research ideas, and his truly exceptional insight and feedback regarding much of our work. Without his help as well, I would not have finished.

- My committee members, Dr. Chandra Chekuri and Dr. Jeff Erickson for their guidance and feedback.

- Dr. Laura P. Swiler for being my co-mentor at Sandia National Laboratories, for providing me with excellent guidance and feedback on my research, and for giving me many research ideas.

- Dr. James E. Campbell for being my mentor at Sandia National Laboratories, and for continuing to be my mentor even after his retirement. His guidance and encouragement both at work and in life are irreplaceable.

- Dr. Dan Roth for his guidance for my undergraduate work, for his encouragement to pursue a Ph.D., and for his help in getting me into graduate school.

- Dr. Russ D. Skocypec, Dr. Robert M. Cranwell, and Bruce M. Thompson for giving me the research opportunities at Sandia National Laboratories, and for being so understanding and patient with my dissertation.

- Dr. Jean-Paul Watson, Craig Lawton, Dr. Dan Briand, Dr. John Eddy, and all my other co-workers at Sandia National Laboratories for all their help and support throughout all those summers.

- Sandia National Labboratories for awarding me the Sandia National Labboratories Fellowship that provided me with the opportunity to work and gain invaluable experience.

- The Air Force Office of Scientific Research (FA9550-07-1-0232) and Austral Engineering and Software, Inc. that provided funding for this research.

- Dr. Luke Olson for his advice on graduate school, and for his patience on the Ultimate Frisbee field.

- Dr. Gary L. Ragatz and Dr. Antoine Jouglet for their help in part of my research.

- My labmates through the years, Dr. Hemanshu Kaul, Dr. Shane N. Hall, Dr. Laura A. McLay, Ruben Proano, Alex Nikolaev, Adrian Lee, Doug King, JD Robbins, and Alex Duda for their friendship, feedback, and support.

- Calvin K. Wong, Terry H. Wong, Neil Patel, Shaun Law, Kristie Tong, Matt Dryden, Betty Earle, Helen Mao, and Marci Meingast for all their friendship, support, and encouragement since the beginning of this project, and for helping me survive through many of my personal obstacles.

- Tanya Crenshaw, Erin Wolf, Shamsi Iqbal, and Jacob Biehl for all their help, advice and feedback.

- All my friends throughout the years in Champaign-Urbana for their help in keeping me grounded and sane. In particular, Tommy Yun, Steve Long, Annie Cheever, Lauren Jakubowski, Pahn Pataramekin, Natalie Bowerman, Abhi Rao, Travis Dixon, and Maggie Burns for putting up with me.

- The Champaign-Urbana ultimate frisbee community, Prion, and the Illinois Men's Team for giving me an outlet from work, and keeping me in shape.

- Jane C. Bailie for all the cookies, brownies, and sandwiches, and more importantly for helping me get through the hard times.

- Katie Dorry, the cutest beagle in the world, for always waiting at home for me.

- My aunty, Lulu Bagaman for her support, for teaching me many heartfelt lessons, and for her insightful perspective on viewing the world.

- My parents, Christina and Leo Kao for believing in me, and for challenging me to overcome all obstacles.

- My brother, Ron Kao for taking care of home while I was away for so many years.

- Tamary Alvarez for all those unforgettable joyful and happy moments, for the motivation, for all her tangible and intangible encouragement and support, and for telling me to finish!

- God for everything.

Table of Contents

List of Tables . . . . . **x**

List of Figures . . . . . **xii**

Chapter 1 Introduction **1**

Chapter 2 Background **5**
2.1 Branch and Bound 5
2.2 Meta-heuristics 9

Chapter 3 The $1|r_i|\sum U_i$ Scheduling Problem **11**
3.1 Dominance Rules 12
3.2 Memory-based Dominance Rule 17
3.3 Branch, Bound, and Remember Algorithm 19
3.3.1 Bounding Scheme 20
3.3.2 Branching and Dominance Scheme 22
3.3.3 Enhancements to the BB&R Algorithm 26
3.4 Computational Results 28
3.5 Conclusion 35

Chapter 4 The $1|r_i|\sum t_i$ Scheduling Problem **38**
4.1 Background and Notations 39
4.2 Dominance Rules 41
4.3 Bounding Scheme 49
4.4 Branch, Bound, and Remember Algorithm 52
4.5 Computational Results 55
4.6 Conclusion 65

Chapter 5 The $1|ST_{sd}|\sum t_i$ Scheduling Problem **67**
5.1 Notations 69
5.2 Branch, Bound, and Remember Algorithm 70
5.2.1 Dominance Rule 70
5.2.2 Bounding Scheme 71
5.2.3 The Algorithm 73
5.3 Counterexample 75

5.4 Computational Results 77
5.5 Conclusion 83

Chapter 6 Post Optimality Selection 84
6.1 Discrete Optimization Problem Formulation 86
6.2 Complexity 90
6.3 Algorithms and Heuristics 95
6.3.1 Exact Algorithms 95
6.3.2 Constructive Heuristics 99
6.3.3 Local Search Heuristics 100
6.3.4 Greedy Reduction Algorithm 104
6.4 Computational Results 106
6.5 Conclusion 114

Chapter 7 Summary 117

References 120

Author's Biography 129

List of Tables

3.1 $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Average amd Maximum CPU Time (sec.) . . . 30
3.2 $1|r_i|\sum U_i$ BB&R Algorithms: Average CPU Time (sec.) with LA-NDDOR . . . 31
3.3 $1|r_i|\sum U_i$ BB&R Algorithms: Standard Deviation in CPU Time (sec.) with LA-NDDOR . . . 32
3.4 $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Average CPU Time (sec.) with Different Dominance Rules . . . 33
3.5 $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Maximum CPU Time (sec.) with Different Dominance Rules . . . 34
3.6 EDP vs. MLJ Upper Bounds Comparison for the $1|r_i|\sum U_i$ Scheduling Problem . . . 36

4.1 BLB and the Decomp-DP Lower Bounds Comparison for the $1|r_i|\sum t_i$ Scheuduling Problem in CPU Time (sec.) . . . 56
4.2 BLB and Decomp-DP Lower Bounds Comparison for the $1|r_i|\sum t_i$ Scheduling Problem in CPU Time (sec.) and Percentage Solved (Larger Instances) . . . 57
4.3 $1|r_i|\sum t_i$ BB&R-DBFS Algorithm: Average and Maximum CPU Time (sec.) . . . 59
4.4 $1|r_i|\sum t_i$ BB&R-DFS Algorithm: Average and Maximum CPU Time (sec.) . . . 60
4.5 $1|r_i|\sum t_i$ BB&R-DBFS Algorithm: Maximum and Average Number of Stored States ($\alpha = 0.5, \beta = 0.5$) . . . 62
4.6 $1|r_i|\sum t_i$ BB&R-DBFS Algorithm Using Jouglet et al. [50] Test Instances in Average CPU Time (sec.) . . . 63
4.7 $1|r_i|\sum t_i$ JBC Algorithm Using Jouglet et al. [50] Test Instances in Average CPU Time (sec.) . . . 64

5.1 $1|ST_{sd}|\sum t_i$ BB&R Algorithms: Average and Maximum CPU Time (sec.) . . . 80
5.2 $1|ST_{sd}|\sum t_i$ BB&R Algorithms: Fraction Solved By Time Limit with Different Exploration Strategies . . . 80
5.3 Performance of the Lower Bound Algorithms for the $1|ST_{sd}|\sum t_i$ Scheduling Problem . . . 81

5.4 $1|ST_{sd}|\sum t_i$ BB&R Algorithms Comparison with Luo and Chu [72] . 82

6.1 Counter Example for the GR Algorithm. 105
6.2 First Iteration of the GR Algorithm Applied to Table 6.1 Example. . 106
6.3 The PPOSP: Percentile Function Value Ratio 109
6.4 Algorithms and Heuristics for the PPOSP: Average Running Time (CPU Seconds) 113
6.5 Algorithms and Heuristics for the PPOSP: Average Running Time (CPU Seconds) 115

List of Figures

3.1 Upper Bound Extended Dynamic Programming Algorithm for the $1|r_i|\sum U_i$ Scheduling Problem 23
3.2 Outline of DBFS 26
3.3 BB&R-DBFS Pseudo-Code for the $1|r_i|\sum U_i$ Scheduling Problem . . 27
3.4 LA-NDDOR Pseudo-Code for the $1|r_i|\sum U_i$ Scheduling Problem . . . 28

4.1 BB&R Pseudo-Code for the $1|r_i|\sum t_i$ Scheduling Problem 54

6.1 Two Dimensional Example of *DE_table* for the PPOSP. 97
6.2 Two Dimensional Traversal of *DE_table* for the PPOSP. 98
6.3 The PPOSP: Test Problem 1 Percentile Function Value Results. . . . 107
6.4 The PPOSP: Test Problem 2 Percentile Function Value Results. . . . 108
6.5 The PPOSP: Test Problem 3 Percentile Function Value Results. . . . 108
6.6 The PPOSP: Test Problem 4 Percentile Function Value Results. . . . 110
6.7 The PPOSP: Test Problem 5 Percentile Function Value Results. . . . 110

Chapter 1

Introduction

In the past decade, there has been an explosion of work at the border of computer science research and operations research. Traditionally, researchers in both fields have remained separate, but recent research has started a compilation of work among the researchers in both fields. Journals and conferences have been established to explore this boundary between computer science and operations research. Special interest articles and books have been published since; helping researchers in both communities to gain new perspectives and to leverage each others work [3, 12, 41, 42, 43, 83]

Despite the relative independence in the fields of computer science and operations research, these two disciplines share a large number of common problems. Two of the most common overlaps between the two fields are in the area of combinatorial optimization and decision analysis. These areas have emerged as a great challenge both academically and practically. Research and development in the two areas can lead to both practical and theoretical significance.

Some of the classic combinatorial optimization problems that have been of great interest to computer scientists and operations researchers are the traveling salesman problem [23], the quadratic assignment problem [60], various job shop scheduling problems [64], and many other NP-hard combinatorial optimization problems [38]. The area of decision analysis also has a significant overlap between computer science and operations research. Both fields are in pursuit with strategies in decision support as well as autonomous decision making. This is most noticeable in the computer science sub-field of artificial intelligence.

Although researchers in both fields have limited interaction, they do share some key conceptual backgrounds. For example researchers in both fields study and use techniques from computational complexity theory, algorithms, probability theory, graph theory, and game theory. What sets the researchers in the two fields apart is the different perspective on approaching the problems. In the area of combinatorial optimization, computer scientists are recently focusing on approximation algorithms [101] and randomized algorithms [81], while operations researchers study traditional mathematical programming [22], heuristics and meta-heuristics [11].

This dissertation focuses on two main topics at the intersection of computer science and operations research. The first topic of interest is designing efficient exact algorithms for solving large combinatorial optimization problems. A modified branch and bound (B&B) algorithm, called Branch, Bound and Remember (BB&R) algorithm is presented through three different single machine scheduling problems. One objective of the work in this dissertation is to seek out optimal methods that can solve combinatorial optimization problems with larger instances and with faster computational speed.

The second topic of interest is in the area of decision making, namely post optimality selection. That is given a set of optimal solutions, how can decision-makers select the best solution(s) from the optimal set? In a multi-objective combinatorial optimization environment, it is common for an algorithm to return not just a single optimal (nearly optimal) solution, but a set of Pareto optimal (nearly optimal) solutions. In many real-world applications, such Pareto solution sets can be extremely large. A new discrete optimization problem formulation is presented in this dissertation to help the decision-maker obtain an optimal preferred subset of Pareto optimal solutions.

This dissertation is organized as follows. Chapter 2 presents the background on B&B algorithms, and an introduction to the BB&R algorithm that is used for

solving several single machine scheduling problem presented in Chapters 3, 4, and 5. Backgrounds on meta-heuristic methods are also presented. Chapter 3 presents the BB&R algorithm with the Distributed Best First Search (DBFS) exploration strategy for solving the $1|r_i|\sum U_i$ scheduling problem [56]. Several new dominance rules for the $1|r_i|\sum U_i$ scheduling problem are reported. Theoretical results are presented showing that the dominance rules presented in Chapter 3 can be combined to form an exact algorithm. Computational results are also reported that establish the effectiveness of the BB&R algorithm with the DBFS exploration strategy for a broad spectrum of problem instances and sizes for the $1|r_i|\sum U_i$ scheduling problem.

A variation of the BB&R algorithm with the DBFS exploration strategy is presented in Chapter 4 for solving the $1|r_i|\sum t_i$ scheduling problem [55]. Several memory-based dominance rules for the $1|r_i|\sum t_i$ scheduling problem are incorporated to the BB&R algorithm. A new modified dynamic programming algorithm is also presented to efficiently compute lower bounds for the $1|r_i|\sum t_i$ scheduling problem. Computational results are reported, which show that the BB&R algorithm with the DBFS exploration strategy outperforms the best known algorithms reported in the literature [4, 27, 71, 72, 76].

Chapter 5 also presents a BB&R algorithm for solving the $1|\sum ST_{sd}|\sum t_i$ scheduling problem [54]. The Best First Search (BFS) exploration strategy and a new memory-based dominance rule are incorporated into the BB&R algorithm, which efficiently solves the $1|\sum ST_{sd}|\sum t_i$ scheduling problem. A counterexample to a known dominance rule presented in [72, 71] is also provided. New computational results are reported that demonstrate the effectiveness of the algorithm.

Chapter 6 formulates a discrete optimization problem called the Preferred Pareto Optimal Subset Problem (PPOSP) for the post optimality selection problem [53]. The PPOSP helps decision-makers obtain a reduced subset of preferred Pareto optimal solutions. Theoretical properties of the PPOSP are reported, and several algorithms

and heuristics are also presented.

The dissertation is summarized in Chapter 7. Some concluding remarks on the BB&R algorithm and the PPOSP formulation are provided [53, 54, 56, 55].

Chapter 2

Background

Three single machine scheduling problems are used to establish the effectiveness of the BB&R algorithm proposed in this dissertation. Scheduling problems are common combinatorial optimization problems that have attracted widespread interest within the domains of manufacturing, transportation, computer processing, production planning, as well as computational complexity theory [8, 13]. These problems involve solving for an optimal schedule under various constraints and objectives (e.g., machine environments, job characteristics). For example, single or multiple machines, job shop or flow shop models, and job preemptions are all variants of scheduling problems. Various objectives include minimizing makespan, number of late jobs, and total tardy time; see [8, 13, 44, 62, 87, 93] for reviews of various scheduling problems.

An overview of B&B algorithms used for solving the scheduling problems, and meta-heuristic methods used for post-optimality selection are provided in this chapter. This chapter is organized as follows. Section 2.1 provides a brief introduction to B&B algorithms, while Section 2.2 provides a brief introduction to meta-heuristic methods for multi-objective combinatorial optimization.

2.1 Branch and Bound

B&B algorithms are one of the most common techniques for solving large NP-hard combinatorial optimization problems [39, 104]. Solving these NP-hard combinatorial optimization problems to optimality can be very challenging. B&B algorithms

are general search methods that implicitly search the entire feasible solution space to find an optimal solution. To apply B&B algorithms, there must be a means of computing lower and upper bounds on an instance of the combinatorial optimization problem, and a means of dividing the feasible region of a problem to create smaller sub-problems. Various parameters and components of a B&B algorithm must be tailored based on the specific definition of the combinatorial optimization problem.

The underlying concept for any B&B algorithms is divide and conquer. The original problem is divided into many smaller sub-problems. These smaller sub-problems can be either solved or eliminated for consideration based on bounding information generated from other sub-problems. This allows the B&B algorithm to implicitly enumerate the feasible solution space without examining all feasible solutions. In general, a B&B algorithm can be viewed as building and exploring a search tree that represents the entire feasible solution space. The two main components for any B&B algorithm are the branching scheme, which constructs the search tree, and the bounding scheme, which prunes and eliminates branches from the search tree.

The branching scheme consists of partitioning the entire feasible solution space into smaller and smaller subsets. Each subset can be further divided into smaller subsets. Each node in the search tree represents a subset, and the order of visiting each subset is part of the exploration strategy. The exploration strategy consists of two interrelated components, a heuristic function that measures the *goodness* of each node, and an overall tree traversal scheme. Together they determine a range of exploration strategies, from a depth-first search strategy, where the heuristic function depends on the depth of a node, to a best-first search strategy, where the heuristic function value takes priority.

The bounding scheme determines what branches of the search trees still need to be explored. One key component is the bounding function. Such a bounding function estimates how good a feasible solution may be generated from exploring a particular

node. If the bounding function gives a tight bound, then the node can be prune or *fathomed*. In addition to pruning by bound, dominance relationships may also be used to reduce the number of branches in the search tree. A node dominates another node if the dominated node can only lead to solutions that are no better than solutions found by exploring the dominant node. These dominance relationships are typically problem specific, and are dependent on the characteristics of the solution structure.

The BB&R algorithm [55, 54, 56] considers a new technique within the general B&B algorithm framework. The key component of the BB&R algorithm consists of using enhanced *memory*-based dominance relationships, where states are memorized and compared. A new exploration strategy, namely the Distributed Best First Search (DBFS), which exploits the benefits of both the depth-first search strategy and the best-first search strategy, is also incorporated into the BB&R algorithm. Chapters 3, 4, and 5 introduce variations of the BB&R algorithm for three different scheduling problems.

The use of dominance relationships in B&B algorithm is not new. The concept of storing states in memory to help build an optimal solution is also not an entirely new concept. Dynamic programming [9] shares a similar concept, where optimal solutions are built backwards, following a sequence of optimal decisions. Tabu search [40] stores previously visited solutions for guiding the local search process. As mentioned above, dominance rules are problem dependent. Note that there are some similar concepts in the artificial intelligence planning community. Heuristic searches [105, 106] are used to solve large planning problems. These heuristic searches are typically graph-based and are used to explore large state-spaces. The objective of these heuristic searches is to find a path from a node representing a start state to a node that represents a goal state. Every visited node in the search process is stored in memory. By storing all nodes, this avoids exploring nodes that have previously been visited. This idea of avoiding duplicates of previously visited nodes is similar to the dominance rules used

in the BB&R algorithm.

The general framework of the BB&R algorithm are outlined by the following steps:

Step 1: Compute upper and lower bounds ub and lb for the optimization problem.

Step 2: Generate a root node.

Step 3: Insert the root node into a heap.

Step 4: If the heap is not empty then go to the next step. Otherwise, the optimal solution is found and the algorithm stops.

Step 5: Obtain a current node by removing the top node from the heap.

Step 6: Using non-memory based dominance rule filter out the possible branching from the current node.

Step 7: For each new subproblem use memory-based dominance rule to further eliminate dominated branches.

Step 8: For each remaining subproblem compute a lower bound lb.

Step 9: If $lb \geq ub$ then prune the current node by going to Step 6. Otherwise, go to the next step.

Step 10: For each remaining subproblem generate a new node and add the new node to the hash table (for memory-based dominance rules) and the heap.

Step 11: Go to Step 6.

Note that the heap data structure can be interchange with other data structure changing the exploration strategy.

2.2 Meta-heuristics

Although exact algorithms like B&B algorithms guarantee finding the optimal solution, they are often impractical for large combinatorial optimization problems. In the last few decades, there has been an increasing interest in meta-heuristics methods for solving combinatorial optimization problems. The three most popular approaches are simulated annealing, tabu search, and evolutionary algorithms. Meta-heuristics methods do not provide any guarantees and could lead to sub-optimal solutions (see [11] for a survey of meta-heuristics methods for combinatorial optimization problems). The second topic of interest in this dissertation addresses post-optimality selection for multi-objective combinatorial optimization problems. This section provides a brief overview of the different meta-heuristic used.

Simulated annealing, as the name suggests evolved from the idea of the annealing process, the gradual solidification process of cooling of a liquid. Several multi-objective simulated annealing algorithms (MOSA) have been proposed. Some references include [21, 49, 100]. The differences across these proposed MOSA are their implementation on scalarization of the objective functions, neighborhood functions, and the temperature adjustment rules for varying the acceptance probability.

Another common method is tabu search, a memory based method. The key concept is the incorporation of a tabu list, which memorizes previously visited states. The tabu lists are used to guide the search process into unexplored regions of the search space. Some references for multi-objective tabu search (MOTS) include [7, 37, 47]. The differences across these MOTS are their implementation on using multiple tabu lists for each objective function and the variation on using short, intermediate, and long term memory tabu lists.

Perhaps the most popular of the three methods discussed here are evolutionary algorithms. Evolutionary algorithms have been the dominant focus in multi-objective

combinatorial optimization. The fundamental concept underlying these methods is *survival of the fittest*. These are population dependent approaches, where solutions compete among each other and are modified based on evolution procedures. Several surveys and books in multi-objective evolutionary algorithms have been published in recent years, including [18, 19, 20, 28, 29, 46]. A few popular evolutionary algorithms include Multi Objective Genetic Algorithm [35], Non-dominated Sorting Genetic Algorithm II [30], and Pareto Archived Evolution Strategy [59].

Combinations of the different approaches mentioned above have also been proposed for multi-objective combinatorial optimization problems, including combination of simulated annealing and genetic algorithms [15], combination of local search and genetic algorithms [48], and interactive methods with simulated annealing and tabu search [78, 92].

Chapter 3

The $1|r_i|\sum U_i$ Scheduling Problem

The scheduling problem addressed in this chapter is the single machine scheduling problem, denoted as $1|r_i|\sum U_i$ [68]. The problem consists of a set of jobs $J = \{1, 2, \ldots, n\}$ to be scheduled in sequence, where associated with each job is a release time r_i, a processing time p_i, and a due-date d_i. The indicator variable $U_i = 0$ if job i is scheduled on time, and $U_i = 1$ if job i is late. A job is considered late if the completion time c_i of a scheduled job i is greater than its due-date d_i. By design, late jobs can be arbitrarily appended to the end of the sequence of on-time jobs. Without loss of generality, assume that $r_i + p_i \leq d_i$ for all $i = 1, 2, \ldots, n$. The objective of the $1|r_i|\sum U_i$ scheduling problem is to minimize the number of late jobs, $\min \sum_{i=1}^{n} U_i$, where jobs are scheduled on a single machine without preemptions.

The $1|r_i|\sum U_i$ scheduling problem is NP-hard [68]. The more general problem, where jobs are weighted, $1|r_i|\sum w_i U_i$, is NP-hard in the strong sense [68]. Polynomial-time special cases of $1|r_i|\sum U_i$ include when the release times are equal or when the jobs are similarly ordered (i.e., $r_i < r_j \Rightarrow d_i \leq d_j$); these can be solved in $O(n \log n)$ time [79, 66, 58].

Exact methods for solving the $1|r_i|\sum U_i$ scheduling problem include branch and bound (B&B) algorithms [76, 6, 27], a mixed integer linear program formulation [63], and a combination of constraint propagation and B&B methods [5]. Dauzère-Pérès and Sevaux [26] also propose a Lagrangean relaxation algorithm based on a new mixed integer linear programming formulation. Dauzère-Pérès [25] provides lower bounds based on a relaxation of a mixed integer linear programming formulation as well as

the minimizing Late Job (MLJ) heuristic. Meta-heuristics such as genetic algorithms have also been developed and applied to the problem [94]. M'Hallah and Bulfin [76] and Pèridy et al. [86] also present results for the weighted version of the scheduling problem.

This chapter introduces the Branch, Bound, and Remember (BB&R) algorithm, an exact algorithm that can be used to solve the $1|r_i|\sum U_i$ scheduling problem. Several dominance rules for the $1|r_i|\sum U_i$ are presented in the next two sections, including enhancements to two previously known dominance rules as well as a new memory-based dominance rule. A new dynamic programming algorithm is also introduced and used to compute tighter upper bounds for the $1|r_i|\sum U_i$ scheduling problem. A BB&R algorithm using the Distributed Best First Search (DBFS) exploration strategy [55] is described and compared to the traditional depth-first search (DFS)and best-first search (BFS) exploration strategies. The computational results reported indicate that the BB&R algorithm outperforms the current best known algorithms.

The chapter is organized as follow. Section 3.1 describes three nonmemory-based dominance rules for the $1|r_i|\sum U_i$ scheduling problem: the Early Job Rule (EJR), the Nearly Due Date Order Rule (NDDOR), and the Idle Time Rule (ITR). Section 3.2 describes a new memory-based dominance rule as well as a proof showing that the dominance rules presented in this chapter can be combined to form an exact algorithm. Section 3.3 provides details of the BB&R algorithm with the DBFS exploration strategy. Computational results are reported in Section 3.4, followed by concluding comments in Section 3.5.

3.1 Dominance Rules

This section formally presents three dominance rules, two of which are extensions of dominance rules introduced in Baptiste et al. [6] and Dauzère-Pérès and Sevaux

[27]. A brief introduction to dominance rules, as well as the necessary notation are provided.

Dominance rules are properties that exploit the structure of optimal solutions, and hence, can be used as pruning strategies. More specifically, these rules identify properties that at least one optimal solution must satisfy. Therefore, these dominance rules can prune many solutions, including optimal solutions. However, they will not prune *all* optimal solutions. Baptiste et al. [5, 6] present several dominance rules that they incorporated into their B&B algorithm. Dauzère-Pérès and Sevaux [26, 27] also suggest a dominance rule incorporated into both their B&B algorithm and their Lagrangean relaxation method for a mixed integer programming formulation. These dominance rules are designed to provide a significant reduction in the search space.

To describe these dominance rules, the following notations and assumptions are needed. Jobs are assumed to be sorted by due-date (i.e., $i < j \Rightarrow d_i < d_j \vee (d_i = d_j \wedge r_i \leq r_j)$). Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ be a sequence of on-time jobs, where $\sigma_i \in J$ for $i = 1, 2, \ldots m$. Let

- c_σ denotes the completion time of the sequence of on-time jobs,
- c_{σ_i} denotes the completion time of job σ_i (define $c_{\sigma_0} = 0$),
- s_{σ_i} denotes the start time of job σ_i (define $s_{\sigma_{m+1}} = c_\sigma$),
- $S_\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ denotes the set of jobs that have been scheduled on time,
- T_σ denotes the set of jobs that must be tardy,
- $F_\sigma = J \setminus (S_\sigma \cup T_\sigma)$ denotes the set of unscheduled free jobs,
- E_σ denotes the set of free jobs that must be on-time,
- $\hat{r}_\sigma = \max\{c_\sigma, \min_{i \in F_\sigma} r_i\}$ denotes the earliest start time for the next possible job that can be scheduled,

- $TP_\sigma = \sum_{i=1}^{m} p_{\sigma_i}$ denotes the sum of the processing times for the scheduled jobs.

Given a sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$ of on-time jobs, assume that jobs are started as soon as possible, i.e., $s_{\sigma_i} = \max(c_{\sigma_{i-1}}, r_{\sigma_i})$.

Baptiste et al. [6] present a dominance rule that identifies jobs that must be on time; the enhanced version of this dominance rule will be referred to as the *Early Job Rule* (EJR). Dauzère-Pérès and Sevaux [27] present a dominance rule that is based on the due-dates of the jobs; the enhanced version of this dominance rule will be referred to as the *Nearly Due Date Order Rule* (NDDOR). These two dominance rules are guaranteed (individually) to not prune all optimal solutions. Unlike the previously proposed dominance rules, these dominance rules are dynamic (i.e., they can be applied when constructing the sequence of on-time jobs). Note that, dynamic dominance rules are not new; Baptiste et al. [5] applied dynamic dominance rules along with their global constraint propagation method.

Baptiste et al. [6] present the EJR dominance rule as a pruning rule based on their decomposition of the search space. The original dominance rule proposed by Baptiste et al. [6] considers only static parameters, such as job processing times, release times and due-dates, while the EJR is dynamic, in that it considers job start times and completion times, which are sequence dependent variables. The EJR is now formally defined.

Definition 3.1.1 *Early Job Rule (EJR)*
A sequence of on-time jobs $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ satisfies the Early Job Rule if the following condition holds: For all $i \in T_\sigma$, there does not exist σ_j, $j = 1, 2, ..., m$ such that $(p_i < p_{\sigma_j} \vee (p_i = p_{\sigma_j} \wedge i < \sigma_j)) \wedge \max(c_{\sigma_{j-1}}, r_i) + p_i \leq \min(d_i, s_{\sigma_{j+1}})$.

The NDDOR dominance rule is motivated by the observation that among the on-time jobs, those with earlier due-dates should be scheduled first. The NDDOR is a stronger version of a dominance rule proposed in Dauzère-Pérès and Sevaux [27].

The NDDOR is more restrictive since it considers the start times of scheduled jobs as opposed to only the release times. Like the EJR, it is dynamic and provides greater pruning. The NDDOR is now formally defined.

Definition 3.1.2 *Nearly Due Date Order Rule (NDDOR)*
A sequence of on-time jobs $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ *satisfies the Nearly Due Date Order Rule if the following condition holds: for* $j = 2, \ldots, m$ $(\sigma_{j-1} < \sigma_j) \vee (s_{\sigma_{j-1}} < r_{\sigma_j})$.

In addition to the EJR and NDDOR, a simple Idle Time Rule (ITR) can further reduce the number of solutions that need to be examined. The ITR is now formally defined.

Definition 3.1.3 *Idle Time Rule (ITR)*
A sequence of on-time jobs $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ *satisfies the Idle Time Rule if the following condition holds: For all* $j = 1, 2, \ldots, m-1$, *there does not exist* $k \in J \setminus S_\sigma$ *such that* $\max(r_k, c_{\sigma_j}) + p_k \leq \min(d_k, s_{\sigma_{j+1}})$.

The ITR eliminates unnecessary idle time from sequences of on-time jobs. The motivation behind this rule suggests that jobs should be scheduled as soon as possible. Idle time places more time constraint on unscheduled jobs, and hence, should be eliminated.

All three dominance rules introduced in this section can be used simultaneously to reduce the solution space while not pruning out all optimal solution. This is formally stated in Theorem 1. The following definitions are needed prior to presenting this result.

Definition 3.1.4 *Let* z *be the number of on-time jobs in an optimal schedule. Let*

$$I(\sigma, t) = \begin{cases} 1 & \text{if the machine is idle during } (t-1, t), \\ 0 & \text{otherwise.} \end{cases}$$

Let

- Ω *denote the set of all optimal sequences,*
- $\Omega^1 = \{\sigma \in \Omega : TP_\sigma \leq TP_\delta \ \forall \delta \in \Omega\}$,
- $\Omega^2 = \left\{\sigma \in \Omega^1 : \sum_{i \in S_\sigma} i \leq \sum_{i \in S_\delta} i \ \forall \delta \in \Omega^1\right\}$,
- $\Omega^3 = \left\{\sigma \in \Omega^2 : \sum_{t=1}^{d_n} tI(\sigma, t) \geq \sum_{t=1}^{d_n} tI(\delta, t) \ \ \forall \delta \in \Omega^2\right\}$,
- $\Omega^4 = \{\sigma \in \Omega^3 : \sum_{i=1}^{z} i\sigma_i \geq \sum_{i=1}^{z} i\delta_i \ \forall \delta \in \Omega^3\}$.

Theorem 1 *If* $\widehat{\sigma} = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_z) \in \Omega^4$, *then* $\widehat{\sigma}^m = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_m)$ *satisfies the EJR, the NDDOR, and the ITR, for* $m = 1, 2, \ldots, z$.

Proof: Suppose $\widehat{\sigma}^m$ violates the EJR. Then there exists a tardy job $t \in T_{\widehat{\sigma}^m}$ and an on-time job $\widehat{\sigma}_j \in S_{\widehat{\sigma}^m}$ such that

$$(p_t < p_{\widehat{\sigma}_j} \vee (p_t = p_{\widehat{\sigma}_j} \wedge t < \widehat{\sigma}_j)) \wedge (\max(c_{\widehat{\sigma}_{j-1}}, r_t) + p_t \leq \min d_t, s_{\widehat{\sigma}_{j+1}})).$$

This implies that a new sequence $\widehat{\sigma}' = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_{j-1}, t, \widehat{\sigma}_{j+1}, \ldots, \widehat{\sigma}_z)$ of on-time jobs can be created by replacing job $\widehat{\sigma}_j$ with job t. This new sequence has the same number of on-time job as $\widehat{\sigma}$, and is therefore optimal. If $p_t < p_{\widehat{\sigma}_j}$, then $TP_{\widehat{\sigma}'} < TP_{\widehat{\sigma}}$, which contradicts that $\widehat{\sigma} \in \Omega^4$. If $p_t \geq p_{\widehat{\sigma}_j}$, then $p_t = p_{\widehat{\sigma}_j} \wedge t < \widehat{\sigma}_j$, which would imply $TP_{\widehat{\sigma}'} = TP_{\widehat{\sigma}}$ and $\sum_{i \in S_{\widehat{\sigma}'}} i < \sum_{i \in S_{\widehat{\sigma}}} i$, which contradicts that $\widehat{\sigma} \in \Omega^4$. Therefore, $\widehat{\sigma}^m$ must satisfy the EJR.

Now suppose $\widehat{\sigma}^m$ violates the ITR. Then there exists a job $k \in J \backslash S_{\widehat{\sigma}^m}$ and j, $1 \leq j \leq m-1$, such that $\max(r_k, c_{\widehat{\sigma}_j}) + p_k \leq \min(d_k, s_{\widehat{\sigma}_{j+1}})$. This implies that k can be scheduled between $\widehat{\sigma}_j$ and $\widehat{\sigma}_{j+1}$ without changing the starting times of any of the other jobs. If $k \notin \{\widehat{\sigma}_{m+1}, \ldots, \widehat{\sigma}_z\}$, then the sequence $(\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_j, k, \widehat{\sigma}_{j+1}, \ldots, \widehat{\sigma}_z)$ has $z+1$ on-time jobs, which contradicts the optimality of $\widehat{\sigma}$. Therefore, $k \in \{\widehat{\sigma}_{m+1}, \ldots, \widehat{\sigma}_z\}$.

Suppose $k = \widehat{\sigma}_t$ for some $m+1 \leq t \leq z$ and let $\widehat{\sigma}' = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_j, k, \widehat{\sigma}_{j+1}, \ldots, \widehat{\sigma}_{t-1}, \widehat{\sigma}_{t+1}, \ldots, \widehat{\sigma}_z)$. Then $\widehat{\sigma}'$has the same number of on-time job as $\widehat{\sigma}$ and is therefore optimal. Furthermore, $TP_{\widehat{\sigma}'} = TP_{\widehat{\sigma}}$ and $\sum_{i \in S_{\widehat{\sigma}'}} i = \sum_{i \in S_{\widehat{\sigma}}} i$, but $\sum_{t=1}^{d_n} tI(\widehat{\sigma}', t) > \sum_{t=1}^{d_n} tI(\widehat{\sigma}, t)$, which contradicts that $\widehat{\sigma} \in \Omega^4$. Therefore, $\widehat{\sigma}^m$ must satisfy the ITR.

Now suppose $\widehat{\sigma}^m$ violates the NDDOR. Suppose two consecutive jobs, $\widehat{\sigma}_{j-1}$ and $\widehat{\sigma}_j$, violate the NDDOR (i.e., $\widehat{\sigma}_{j-1} > \widehat{\sigma}_j \wedge s_{\widehat{\sigma}_{j-i}} \geq r_{\widehat{\sigma}_j}$). Let $\widehat{\sigma}' = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_{j-2}, \widehat{\sigma}_j, \widehat{\sigma}_{j-1}, \widehat{\sigma}_{j+1}, \ldots, \widehat{\sigma}_z)$ be the schedule obtained by interchanging these two jobs. Then $\widehat{\sigma}_{j-1} > \widehat{\sigma}_j$ implies $d_{\widehat{\sigma}_{j-1}} \geq d_{\widehat{\sigma}_j}$ which then implies that both jobs will be on time. Furthermore, $s_{\widehat{\sigma}_{j-1}} \geq r_{\widehat{\sigma}_j}$, implies that $\widehat{\sigma}_j$ will satisfy its release date in $\widehat{\sigma}'$. Therefore, the interchange creates a new optimal schedule. $\widehat{\sigma}'$ and $\widehat{\sigma}$ contain the same set of jobs and the machine is idle during precisely the same moments in time, hence $TP_{\widehat{\sigma}'} = TP_{\widehat{\sigma}}$, $\sum_{i \in S_{\widehat{\sigma}'}} i = \sum_{i \in S_{\widehat{\sigma}}} i$, and $\sum_{t=1}^{d_n} tI(\widehat{\sigma}', t) = \sum_{t=1}^{d_n} tI(\widehat{\sigma}, t)$. But $\sum_{i=1}^{z} i\widehat{\sigma}'_i > \sum_{i=1}^{z} i\widehat{\sigma}_i$, which contradicts that $\widehat{\sigma} \in \Omega^4$. Therefore, $\widehat{\sigma}^m$ must satisfy the NNDOR. □

3.2 Memory-based Dominance Rule

This section describes the General Memory Dominance Rule (GMDR), used in the BB&R algorithm presented in Section 3.3. A proof is provided showing that the GMDR can be used with the EJR, the NDDOR, and ITR such that there exists an optimal solution that satisfies all the dominance rules.

Similar to the dominance rules described in Section 3.1, memory based dominance rules (MBDR) are also used to reduce the search space. Unlike the other dominance rules, they do not exploit the structure of optimal solutions, but rather compare partial sequences of on-time jobs and determine whether a particular partial sequence is guaranteed to lead to a solution that is at least as good as other solutions found by the other partial sequences. The GMDR is now formally defined.

Definition 3.2.1 *General Memory Dominance Rule (GMDR)*

Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ *and* $\delta = (\delta_1, \delta_2, \ldots, \delta_q)$ *be partial sequences of on-time jobs. Then* σ *dominates* δ *if* $(F_\sigma \supseteq F_\delta) \wedge (\hat{r}_\sigma \leq \hat{r}_\delta)$ *and one of the following holds:*

1. $m > q$
2. $(m = q) \wedge (TP_\sigma < TP_\delta)$
3. $(m = q) \wedge (TP_\sigma = TP_\delta) \wedge \left(\sum_{i \in S_\sigma} i < \sum_{i \in S_\delta} i\right)$
4. $(m = q) \wedge (TP_\sigma = TP_\delta) \wedge \left(\sum_{i \in S_\sigma} i = \sum_{i \in S_\delta} i\right) \wedge \left(\sum_{t=1}^{d_n} tI(\sigma, t) > \sum_{t=1}^{d_n} tI(\delta, t)\right)$
5. $(m = q) \wedge (TP_\sigma = TP_\delta) \wedge \left(\sum_{i \in S_\sigma} i = \sum_{i \in S_\delta} i\right) \wedge \left(\sum_{t=1}^{d_n} tI(\sigma, t) = \sum_{t=1}^{d_n} tI(\delta, t)\right)$
   $\wedge \left(\sum_{i=1}^{m} i\sigma_i > \sum_{i=1}^{q} i\delta_i\right)$

The GMDR suggests that given two partial sequences of on-time jobs σ and δ, if σ dominates δ, then it is unnecessary to evaluate full sequences of on-time jobs that are constructed by scheduling more jobs onto the end of δ, and that it is sufficient to only evaluate full sequences of on-time jobs that are constructed by scheduling more jobs onto the end of σ.

Theorem 2 formally states that the GMDR can be used simultaneously with the EJR, NDDOR, and the ITR without pruning out all optimal solutions.

Theorem 2 *If* $\widehat{\sigma} = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_z) \in \Omega^4$, *then* $\widehat{\sigma}^q = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_q)$ *is not dominated by any other sequence, for* $q = 1, 2, \ldots, z$.

Proof: Suppose $\sigma^m = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ dominates $\widehat{\sigma}^q = (\widehat{\sigma}_1, \widehat{\sigma}_2, \ldots, \widehat{\sigma}_q)$ for some q such that $1 \leq q \leq z$. $F_{\sigma^m} \supseteq F_{\widehat{\sigma}^q}$ and $\widehat{r}_{\sigma^m} \leq \widehat{r}_{\widehat{\sigma}^q}$ imply that the sequence $(\widehat{\sigma}_{q+1}, \widehat{\sigma}_{q+2}, \ldots, \widehat{\sigma}_z)$ of remaining jobs can be appended to σ^m to obtain a new feasible sequence $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m, \widehat{\sigma}_{q+1}, \widehat{\sigma}_{q+2}, \ldots, \widehat{\sigma}_z)$. The remainder of the proof consists in showing that if any of the five conditions in the GMDR hold true, then it would contradict that $\widehat{\sigma} \in \Omega^4$.

Case 1 *$m > q$. Then σ is a feasible sequence that contains more on-time jobs than $\widehat{\sigma}$, which contradicts the optimality of $\widehat{\sigma}$.*

Case 2 *$(m = q) \wedge (TP_{\sigma^m} < TP_{\widehat{\sigma}^q})$. Then σ is a feasible sequence that contains the same number of on-time jobs as $\widehat{\sigma}$, hence σ is an optimal sequence. This also holds true in all the following cases. $TP_{\sigma^m} < TP_{\widehat{\sigma}^q}$ implies $TP_\sigma < TP_{\widehat{\sigma}}$,which contradicts that $\widehat{\sigma} \in \Omega^4$.*

Case 3 *$(m = q) \wedge (TP_{\sigma^m} = TP_{\widehat{\sigma}^q}) \wedge \left(\sum_{i \in S_{\sigma^m}} i < \sum_{i \in S_{\widehat{\sigma}^q}} i\right)$. Then $TP_{\sigma^m} = TP_{\widehat{\sigma}^q}$ implies $TP_\sigma = TP_{\widehat{\sigma}}$. Furthermore, $\sum_{i \in S_{\sigma^m}} i < \sum_{i \in S_{\widehat{\sigma}^q}} i$ implies $\sum_{i \in S_\sigma} i < \sum_{i \in S_{\widehat{\sigma}}} i$, which contradicts that $\widehat{\sigma} \in \Omega^4$.*

Case 4 *$(m = q) \wedge (TP_{\sigma^m} = TP_{\widehat{\sigma}^q}) \wedge (\sum_{i \in S_{\sigma^m}} i = \sum_{i \in S_{\widehat{\sigma}^q}} i) \wedge (\sum_{t=1}^{d_n} tI\ (\sigma^m, t) > \sum_{t=1}^{d_n} tI\,(\widehat{\sigma}^q, t))$. Then $TP_\sigma = TP_{\widehat{\sigma}}$ and $\sum_{i \in S_\sigma} i = \sum_{i \in S_{\widehat{\sigma}}} i$. Furthermore,$\sum_{t=1}^{d_n} tI\ (\sigma^m, t) > \sum_{t=1}^{d_n} tI\,(\widehat{\sigma}^q, t)$ implies $\sum_{t=1}^{d_n} tI\,(\sigma, t) > \sum_{t=1}^{d_n} tI\,(\widehat{\sigma}, t)$, which contradicts that $\widehat{\sigma} \in \Omega^4$.*

Case 5 *$(m = q) \wedge (TP_{\sigma^m} = TP_{\widehat{\sigma}^q}) \wedge \left(\sum_{i \in S_{\sigma^m}} i = \sum_{i \in S_{\widehat{\sigma}^q}} i\right) \wedge$
$\left(\sum_{t=1}^{d_n} tI\,(\sigma^m, t) = \sum_{t=1}^{d_n} tI\,(\widehat{\sigma}^q, t)\right) \wedge (\sum_{i=1}^{m} i\sigma_i > \sum_{i=1}^{q} i\delta_i)$. Then $TP_\sigma = TP_{\widehat{\sigma}}$,
$U \sum_{i \in S_\sigma} i = \sum_{i \in S_{\widehat{\sigma}}} i$, and $\sum_{t=1}^{d_n} tI\,(\sigma, t) = \sum_{t=1}^{d_n} tI\,(\widehat{\sigma}, t)$. Furthermore,$\sum_{i=1}^{m} i\sigma_i^m > \sum_{i=1}^{q} i\delta_i^q$ implies $\sum_{i=1}^{m} i\sigma_i > \sum_{i=1}^{q} i\delta_i$, which contradicts that $\widehat{\sigma} \in \Omega^4$.* □

3.3 Branch, Bound, and Remember Algorithm

This section introduces the BB&R algorithm for solving the $1|r_i|\sum U_i$ scheduling problem. Section 3.3.1 describes the bounding scheme, including a dynamic programming algorithm that produces tighter upper bounds compared to other known heuristics. Section 3.3.2 describes the branching scheme with two different exploration strategies and illustrates how the dominance rules described in Sections 3.1

and 3.2 are used. Pseudo-codes for the BB&R algorithm are also provided in Section 3.3.2. In addition, Section 3.3.3 also describes an extension to the BB&R algorithm that leads to further computational speed up.

The B&B algorithm presented in this chapter incorporates memory-based dominance properties to prune a subproblem if it is dominated by another subproblem that has already been generated. To implement this, it is necessary to store (remember) the subproblems that have already been generated (and hence the name Branch, Bound, and Remember).

Note that the technique of memorizing previously visited nodes has been previously studied. In the scheduling domain, Jouglet et al. [50] have also used memory to record "no-good recording" and "better sequence" to prune dominated solutions in the solution space. Pèridy et al. [86] also introduces the use of short term scheduling memory for solving the $1|r_i|\sum w_i U_i$ scheduling problem. Dynamic programming techniques [10, 84] share a similar concept. Morin and Marsten [80] tried to combine dynamic programming and B&B strategies to improve computational efficiency of dynamic programming. Heuristic searches [105, 106] have been used to solve large planning problems. These graph-based searches store previously visited nodes in the search tree to avoid revisiting previously explored paths, similar to the memory-based dominance rules used in the BB&R algorithm.

3.3.1 Bounding Scheme

This section provides an overview of the Minimizing Late Job (MLJ) heuristic [25], a dynamic programming algorithm [58], and an extended dynamic programming algorithm based on Kise et al. [58]. These heuristics are used to calculate lower and upper bounds for the number of late jobs. These bounds provide an estimate on the quality of a branch. The efficiency and the quality of the lower and upper bounds can lead to significant performance improvements to the overall algorithm.

The bounding scheme works as follows. Initially, prior to any branching, an upper bound is computed based on the MLJ heuristic and the extended dynamic programming algorithm; the minimum of these two bounds is retained. As the branching process proceeds with additional jobs being scheduled, lower bounds are computed based on the remaining free jobs, using a dynamic programming algorithm with relaxed release times and due-dates. If the lower and upper bounds are tight, then the branch is pruned.

Kise et al. [58] propose a dynamic programming algorithm for solving a special case of the $1|r_i|\sum U_i$ scheduling problem, where the jobs are similarly ordered (i.e., $r_i < r_j \Rightarrow d_i \leq d_j$). This dynamic programming algorithm is incorporated into the BB&R algorithm to generate lower bounds. Since the instances solved in this chapter are generally not similarly ordered, the jobs' release times and due-dates are relaxed to generate the lower bound. Two separate lower bounds are computed based on either relaxing the release times or relaxing the due-dates, where the maximum of the two lower bounds is retained.

An extended dynamic programming algorithm (EDP) based on Kise et al. [58] is used to compute an upper bound. The pseudo-code illustrated in Figure 3.1 outlines the new EDP algorithm. Let *Jobs* be the set of free jobs sorted in due-date order. The function $REPLACE(\sigma, k)$, (see Figure 3.1), returns the shortest feasible schedule by considering all schedules that replace a job in σ with job k. If no feasible schedule exists, the function $REPLACE(\sigma, k)$ returns σ. The function $INSERT(\sigma, k)$ returns the shortest feasible schedule by considering all schedules that insert job k into σ. If no feasible schedule exists, the function $INSERT(\sigma, k)$ returns σ. The array max_n_jobs(k) stores the maximum number of jobs that can be scheduled from the subset of jobs, $\{Jobs[1], \ldots, Jobs[k]\}$. The matrix $C(k, m)$ stores the earliest completion time for scheduling exactly m jobs among $\{Jobs[1], \ldots, Jobs[k]\}$ and $Seq(m, k)$ is the partial sequence of on-time jobs associated with $C(k, m)$.

The EDP algorithm follows the same basic recursion as in Kise et al. [58]. Let $Comp_{k,m}$ be the best completion time for a partial sequence with m jobs, where job k may (or may not) be scheduled. The recursion in the dynamic programming algorithm in Kise et al. [58] is $Comp_{k,m} = \min(Comp_{k-1,m}, \max(Comp_{k-1,m-1}, r_k) + p_k)$. In order to compute $Comp_{k,m}$, the dynamic programming algorithm in Kise et al. [58] considers two types of sequences; sequences of length m where job k is not scheduled, and the sequence where job k is appended to a sequence with length $m-1$. Moreover, the EDP algorithm used in the BB&R algorithm considers two additional type of sequences; partial sequences of length $m-1$ where job k is inserted, and also partial sequences of length m where job k replaces another job. These two additional types of sequences are generated by the functions *REPLACE* and *INSERT* (see Figure 3.1). By design, the EDP algorithm considers additional possible schedules, and frequently generates tighter upper bounds than the MLJ heuristics. Section 3.4 reports computational results comparing the EDP with the MLJ heuristic described in this section.

Dauzère-Pérès [25] introduced the MLJ heuristic, which is constructive and consists of attempting to schedule new jobs with release dates earlier than the current completion time of the last scheduled job. Jobs are also chosen in a way that minimizes the completion time of the last scheduled job. The MLJ heuristic is also used to compute the upper bound for the number of late jobs. This heuristic runs in $O(n^2)$ time.

3.3.2 Branching and Dominance Scheme

This section describes how all the dominance rules are used in conjunction with the branching scheme in the BB&R algorithm. At each branching stage in the search tree, the sequence of on-time jobs is checked for consistency with the dominance rules. The BB&R algorithm using the DBFS exploration strategy is described. The

EDP($Jobs$**,** R, D, P**)**

R, D, P are the release times, due-dates, and processing times respectively.
max_n_jobs $= \{0, \ldots, 0\}$, $C = \{\{\infty, \ldots, \infty\}, \ldots, \{\infty, \ldots, \infty\}\}$
max_n_jobs[1] $= 1$
$C(1,1) = R(Jobs[1]) + P(Jobs[1])$
$Seq(1,1) = Jobs[1]$
for $k = 2$ to the number of $Jobs$ **do**
  max_n_jobs[k] = max_n_jobs[k] $+ 1$
  for $m =$ max_n_jobs[k] $\rightarrow 2$ **do**
    if $C(k-1, m) < \infty$ **then**
      temp_seq = REPLACE($Seq(m, m), Jobs[k]$)
    else
      temp_seq $= \infty$
    end if
    temp_seq2 = INSERT($Seq(m-1, m-1), Jobs[k]$)
    if Complete_time(temp_seq) $\leq$ Complete_time(temp_seq2) **then**
      $C(k, m)$ = Complete_time(temp_seq)
      if Complete_time(temp_seq) $< \infty$ **then**
        $Seq(m, m)$ = temp_seq
      end if
    else
      $C(k, m)$ = Complete_time(temp_seq2)
      $Seq(m, m)$ = temp_seq2
    end if
  end for
  if max_n_jobs[k] > 0 **then**
    if $C(k-1, 1) \leq \max(0, R(Jobs[k]) + P(Jobs[k])$ **then**
      $C(k, 1) = C(k-1, 1)$
    else
      $C(k, 1) = \max(0, R(Jobs[k]) + P(Jobs[k])$
      $Seq(1, 1) = Jobs[k]$
    end if
  end if
  max_n_jobs[k] $= \max\{j : C(k, j) < \infty\}$
end for

Figure 3.1: Upper Bound Extended Dynamic Programming Algorithm for the $1|r_i|\sum U_i$ Scheduling Problem

pseudo-codes for the BB&R algorithm is also presented.

Given a particular state $(\sigma, F_\sigma, T_\sigma, E_\sigma)$, a new state is explored by adding a new job to the partial sequence σ, (i.e., a new sequence $(\sigma_1, \sigma_2, \ldots, \sigma_m, f_\sigma)$ where m is the length of σ and $f_\sigma \in F_\sigma$). The new state is denoted by $(\sigma', F'_\sigma, T'_\sigma, E'_\sigma)$. The dominance rules are used in two ways: to filter F_σ to find the set of jobs that are free and satisfy the dominance rules; and once the new partial sequence is scheduled, to reduce the number of further branching needed from that state.

The NDDOR and the ITR are initially used to filter F_σ. Only jobs that are in F_σ and satisfy the NDDOR and the ITR are considered for branching. If no such jobs can be found, then the current branch is pruned. Scheduling a new job modifies the previous F_σ, T_σ, and E_σ, where T_σ is modified by checking each job in F_σ for tardiness, and E_σ is computed based on the EJR, (see Section 3.2). The new set of tardy jobs and early jobs are then checked for consistency with the EJR. If any of the late jobs becomes an early job, or if a tardy job does not satisfy the EJR, then the branch is pruned. Note that in Baptiste et al. [5], the dominance rule is used as a preprocessor to identify unscheduled jobs to be considered for branching, which must be either on-time or late. The EJR embedded in the BB&R algorithm is used to actively prune further branching.

If the new state (after branching) satisfies the EJR, the NDDOR, and the ITR, then the GMDR is applied. The set of free jobs F'_σ is used as the hash key to find the corresponding entry in the hash table. If this entry in the hash table is empty, then the current state is stored; otherwise, the GMDR is used to compare the current state and the previously stored state(s). If the new state dominates the previously stored state(s), then the new state replaces the old state(s) in the hash table. If any old state stored in the hash table dominates the new state, then the current branch is pruned.

The combination of the EJR, the NDDOR, and the ITR, with the addition of the

GMDR significantly reduces the search space for the BB&R algorithm. Section 3.2 provided the proof showing that combining all the dominance rules will not eliminate all optimal solutions (i.e., the BB&R algorithm is exact). The pseudo codes for the BB&R algorithm with the DBFS exploration strategies are now presented.

The DBFS exploration strategy is designed to find an optimal solution earlier than Depth-first search. The DBFS is a hybrid between DFS and Best-first search (BFS). In DBFS, states are explored based on the length of the sequence of on-time jobs and a *best*-measure, a heuristic function that evaluates the potential of a state leading to an optimal solution. The DBFS explores states by sequentially considering states with longer and longer sequence of on-time jobs. Let level 1 states be all states with a sequence of on-time jobs of length 1, and let level 2 states be all states with a sequence of on time jobs of length 2, and so forth. The DBFS starts by choosing a state at level 1 to explore. It then chooses a state at level 2 to explore and continue until it reaches the deepest level, at which time it will return to level 1 and repeat. When the DBFS chooses a node at level k, it chooses the one with the highest *best*-measure value. The children of the chosen state are generated and added to level $k+1$. An outline of the DBFS exploration strategy is given in Figure 3.2. Note that there may be iterations in the search process where some levels may not have any unexplored states. In such cases, no new states are explored for that iteration, and no new states are added to the next level. The pseudo-code for the BB&R algorithm with the DBFS (BB&R-DBFS) implementation is given in Figure 3.3.

In the BB&R-DBFS pseudo-code, a heap structure is needed to store states for each level of the search tree. When a state is expanded by scheduling each of the jobs in PF_σ, each new state is verified to satisfy the EJR and the GMDR prior to being added into the next level heap. Note that there may exist states in the heap that are dominated by the GMDR after they have been added to the heap. For example state B may dominate state A by GMDR, but the predecessor of state A may have

Distributed Best First Search

```
Initialize level 0 by storing the root node
while unexplored states exist do
  for each level i, 0 to maximum depth do
    Expand the best node in level i
    Store all the children of the best node in level i + 1
  end for
end while
```

Figure 3.2: Outline of DBFS

been explored before the predecessor of state B, which results in adding state A to the heap before state B. A simple dominance bit (i.e., an indicator that indicates a state is dominated if set), can be used to keep track of such states to avoid extra exploration of state A. In addition, BB&R-DBFS uses a Best_Measure heuristic. Two Best_Measure heuristics are considered in this chapter. The first, denoted by *DD* is based on due-date order. That is, if σ is a sequence of on-time jobs in the current state with m jobs, then $Best = -1 * d_{\sigma_m}$. This Best_Measure favors states with sequences of on-time jobs with earlier due-dates. The second, denoted by *RP*, is more sophisticated and takes earlier release times as well as the processing times of the free jobs into consideration. Define $w_i = d_i - \max(\hat{r}_\sigma, r_i)$ for $i \in F_\sigma$ to be the time window for each job i. The Best_Measure is defined as $Best = \sum_{i \in F_\sigma} w_i/p_i$. This second Best_Measure prefers schedules with smaller value for $\hat{r}_\sigma$ as well as schedules with shorter free jobs in longer time windows. Note that BB&R-DFS also has an implicit Best_Measure, which is based on exploring PF_σ in due-date order.

3.3.3 Enhancements to the BB&R Algorithm

This section describes an enhancement to the BB&R algorithm presented above.

The *look-ahead* NDDOR (LA-NDDOR) is based on the NDDOR. Suppose that job $i \in PF_\sigma$ is appended to σ to obtain $\sigma' = (\sigma_1, \ldots, \sigma_m, \sigma_{m+1})$ where $\sigma_{m+1} = i$.

BB&R-DBFS($\sigma, F_\sigma, T_\sigma, E_\sigma, \hat{r}_\sigma, UB$, **hash_table, heap**($1, \ldots,$ **size**(F_σ))

```
LB = Lower_Bound(F_σ, r̂_σ)
if LB ≥ UB then
  return
end if
Initialize heap(1)
while heap is not empty do
  for i = 1 → UB do
    cur_state = heap(i).pop
    cur_lb = Lower_Bound(cur_state)
    if cur_lb + size(cur_state.T_σ) < UB then
      PF_σ = NDDOR(cur_state.F_σ) and ITR((cur_state.F_σ)
      for each j ∈ PF_σ do
        new_state.σ = cur_state.σ + j
        update new_state from cur_state
        new_best = Best_measure(new_state)
        Violated_EJR = EJR(new_state)
        Violated_GMDR = GMDR(new_state)
        if not Violated_EJR ∧ not Violated_GMDR then
          Store(new_state) in hash_table
          heap(i+1).add(new_state, new_best)
        end if
      end for
    end if
  end for
end while
```

Figure 3.3: BB&R-DBFS Pseudo-Code for the $1|r_i|\sum U_i$ Scheduling Problem

Let $k = \min\{h : h \in F_\sigma \setminus \{\sigma_{m+1}\}\}$. If $\sigma_{m+1} > k$ and $s_{\sigma_{m+1}} \geq r_k$, then k cannot be scheduled in position $m+2$ because it would violate the NDDOR. Furthermore, it cannot be scheduled in any later position because it will violate the EJR. To see this, let $\sigma'' = (\sigma_1, \ldots, \sigma_m,\ \sigma_{m+1}, \ldots, \sigma_q)$. $\sigma_q > k$ because $\sigma_q \in F_\sigma \setminus \{\sigma_{m+1}\}$ and $k = \min\{h : h \in F_\sigma \setminus \{\sigma_{m+1}\}\}$. In addition, $r_k \leq s_{\sigma_{m+1}} \leq s_{\sigma_q}$, therefore, the NDDOR will be violated if k is scheduled in position $q+1$. Consequently, the NDDOR will prevent job k from being scheduled on time in any super-sequence of σ'. Thus job k should be added to $T_{\sigma'}$ instead of $F_{\sigma'}$. This test can be repeated for each job in

LA-NDDOR($F_\sigma, \sigma = (\sigma_1, \ldots, \sigma_m), \sigma_{m+1}$)
  Sort F_σ in Due-Date order, $F' = \{\}, T' = T_\sigma$
  for $k \in F_\sigma \setminus \{\sigma_{m+1}\}$ **do**
    if $c_{\sigma_{m+1}} + p_k > d_k$ **then**
      $T' = T' \bigcup \{k\}$
    else
      if $F' = \{\}$ and $(\sigma_{m+1} > k \wedge s_{\sigma_{m+1}} \geq r_k)$ **then**
        $T' = T' \bigcup \{k\}$
      else
        $F' = F' \bigcup \{k\}$
      end if
    end if
  end for

Figure 3.4: LA-NDDOR Pseudo-Code for the $1|r_i|\sum U_i$ Scheduling Problem

$F_\sigma \setminus \{\sigma_{m+1}\}$, in due-date order, until one is found that can be scheduled in position $m + 2$.

The LA-NDDOR can enhance the algorithms in two ways. First it produces a tighter lower bound earlier in the branching process, and second, if any of such jobs become early, then the entire branch will be pruned by the EJR. Figure 3.4 provides the pseudo-code for this enhancement.

The discussion above yields the following theorem which states that the LA-NDDOR may be used in the BB&R algorithm in conjunction with the other dominance rules such that the BB&R algorithm remains exact.

Theorem 3 *The Branch, Bound, and Remember Algorithm using the look-ahead NDDOR is an exact algorithm.*

3.4 Computational Results

This section reports computational results for the BB&R-DBFS algorithm described in Section 3.3. The computational results of the DBFS exploration strategy are

compared against the computational results of the DFS and the BFS exploration strategies. Computational results for using the two different *best*-measures as well as the LA-NDDOR are also reported. This section also reports computational results for comparing the EDP and MLJ heuristics described in Section 3.3.1.

The effectiveness of the BB&R algorithm is evaluated over 7,200 randomly generated test instances. These test instances were generated using the same generation scheme described in Baptiste et al. [5], based on four parameters: number of jobs, processing time range, maximum slack margins, and machine load, denoted as $(n, [p_{\min}, p_{\max}], slack_{\max}, load)$. The slack margin is defined for each job as $d_i - r_i - p_i$. The machine load is defined to be the ratio between the sum of the job's processing times and the difference of the maximum due-date and the minimum release time. The parameters used for generating the test instances are $n = \{80, 100, \ldots, 300\}$, $[p_{\min}, p_{\max}] = \{[25, 75], [0, 100]\}$, $slack_{\max} = \{50, 100, \ldots, 500\}$, and $load = \{0.5, 0.8, 1.1, 1.4, 1.7, 2.0\}$. For each combination of parameter settings, five random instances are generated for a total of 7,200 instances. The parameters used in generating the test instances in this chapter are identical to the parameters used in generating the instances in Baptiste et al. [6] and Dauzère-Pérès and Sevaux [27]. The experiments in this chapter were executed on a 3 GHz Pentium D PC.

Several variations of the BB&R algorithm are investigated. The DBFS exploration strategy is compared with the DFS and the BFS exploration strategies. Two different *best*-measures, described in Section 3.3.2, are also tested in conjunction with the DBFS exploration strategy. These two variations are denoted as BB&R-DBFS-DD and BB&R-DBFS-RP. The DD *best*-measure favors jobs with earlier due-dates, while the RP *best*-measure favors jobs with earlier release times and shorter processing times.

Table 3.1 reports the average and maximum running time for the BB&R-DBFS-DD and BB&R-DBFS-RP algorithm. The test set is organized by $n - p_{\min} - p_{\max}$, with

each instance in the test set restricted to a one hour total processing time limit. Both variations of the BB&R-DBFS were able to solve all instances to optimality. For some of the larger instances, BB&R-DBFS-DD had significantly longer maximum running time compared to BB&R-DBFS-RP. The BB&R-DBFS-RP was able to solve all instances in under fifteen minutes, with the exception of the $260 - 0 - 100$ and $300 - 0 - 100$ instances. For the remainder of the chapter, the RP *best*-measure is used for evaluting the variations and extension of the BB&R algorithm.

Table 3.1: $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Average amd Maximum CPU Time (sec.)

	BB&R-DBFS-DD		BB&R-DBFS-RP	
$n - p_{\min} - p_{\max}$	Avg.	Max	Avg.	Max
80-0-100	0.5	5.4	0.6	7.6
80-25-75	0.4	1.8	0.4	1.2
100-0-100	1.9	393.0	1.6	281.8
100-25-75	0.5	4.6	0.5	2.7
120-0-100	0.9	15.0	1.3	22.6
120-25-75	0.7	7.1	0.7	4.8
140-0-100	1.2	10.7	1.7	252.2
140-25-75	1.0	20.3	0.8	10.6
160-0-100	1.9	33.6	2.7	73.0
160-25-75	1.3	10.9	1.0	9.2
180-0-100	2.6	109.4	3.5	135.7
180-25-75	1.9	19.0	1.4	11.8
200-0-100	5.1	237.6	5.5	248.5
200-25-75	3.3	149.1	2.3	112.3
220-0-100	5.5	145.7	8.3	496.6
220-25-75	4.0	38.6	2.7	32.5
240-0-100	12.4	515.3	10.7	414.0
240-25-75	5.9	62.4	3.7	38.4
260-0-100	15.8	447.2	18.2	959.6
260-25-75	7.8	127.2	5.1	49.8
280-0-100	26.1	734.3	26.5	633.5
280-25-75	13.1	284.7	7.6	162.5
300-0-100	26.4	510.5	45.7	2924.3
300-25-75	19.6	439.7	11.5	234.5

Tables 3.2 summarizes the results of the test set with the LA-NDDOR extension. Table 3.2 reports the average running time for the BB&R algorithm using the RP

Table 3.2: $1|r_i|\sum U_i$ BB&R Algorithms: Average CPU Time (sec.) with LA-NDDOR

$n - p_{\min} - p_{\max}$	BB&R-DFS	BB&R-BFS-RP	BB&R-DBFS-RP
80-0-100	0.5	0.9	0.4
80-25-75	0.4	0.5	0.4
100-0-100	1.5	0.9	0.6
100-25-75	0.5	0.6	0.4
120-0-100	0.9	2.5	0.6
120-25-75	0.8	0.9	0.5
140-0-100	1.4	1.6	0.8
140-25-75	1.4	1.1	0.7
160-0-100	2.4	2.1	1.1
160-25-75	1.4	1.2	0.8
180-0-100	4.1	2.5	1.3
180-25-75	2.8	1.6	1.0
200-0-100	9.6	4.7	2.2
200-25-75	12.9	1.9	1.5
220-0-100	13.3	6.1	2.6
220-25-75	9.5	2.3	1.8
240-0-100	52.4 (**2**)	5.7	3.9
240-25-75	15.2	2.7	2.4
260-0-100	69.6 (**1**)	7.3	5.5
260-25-75	32.8	3.3	3.2
280-0-100	125.4 (**5**)	10.3	9.1
280-25-75	72.7 (**2**)	4.2	4.7
300-0-100	147.5 (**5**)	14.0	11.5
300-25-75	123.2 (**2**)	5.1	6.5

best measure with different exploration strategies. The DBFS exploration strategy is compared with the DFS and the BFS exploration strategies. Table 3.3 shows the standard deviation of the running time of BB&R-BFS-RP and BB&R-DBFS-RP. The BB&R-DBFS-RP and the BB&R-BFS-RP were able to solve all instances to optimality; however, BB&R-DFS was unable to solve 36 of the 7,200 instances. The number in parentheses associated with some of the entries in Table 3.2 reports the number of instances that were incomplete. The average running times reported in Table 3.2 include those instances that were incomplete.

By using the LA-NDDOR, for the larger instances, BB&R-DBFS-RP was an order of magnitude faster on average then BB&R-DFS. All instances were solved to opti-

mality in less than eight minutes. On average, BB&R-DBFS-RP also out performs BB&R-BFS-RP, moreover, Table 3.3 shows that for most of the instances, BB&R-DBFS-RP has a standard deviation that is less than half of the standard deviation of BB&R-BFS-RP. By design, the LA-NDDOR extension was able to reduce the number of branches in the BB&R algorithm.

Table 3.3: $1|r_i|\sum U_i$ BB&R Algorithms: Standard Deviation in CPU Time (sec.) with LA-NDDOR

$n - p_{\min} - p_{\max}$	BB&R-BFS-RP	BB&R-DBFS-RP
80-0-100	2.6	0.2
80-25-75	0.5	0.1
100-0-100	2.7	2.3
100-25-75	0.6	0.1
120-0-100	16.7	0.5
120-25-75	1.7	0.3
140-0-100	4.9	0.6
140-25-75	2.5	0.5
160-0-100	5.3	1.2
160-25-75	1.8	0.5
180-0-100	7.6	1.6
180-25-75	2.7	0.9
200-0-100	25.3	6.1
200-25-75	3.9	3.4
220-0-100	26.1	3.9
220-25-75	4.5	2.1
240-0-100	21.8	8.0
240-25-75	4.4	2.8
260-0-100	41.0	9.7
260-25-75	5.6	3.9
280-0-100	68.8	22.1
280-25-75	8.7	7.9
300-0-100	64.9	34.0
300-25-75	8.5	10.9

M'Hallah and Bulfin [76] report the best computational results to date for solving the $1|r_i|\sum U_i$ scheduling problem. On average, their algorithm took 193.4 seconds for solving instances with size $n = 200$ on a 1 Ghz Pentium IV PC. The BB&R-BFS-RP algorithm on average took 5.7 seconds, while the BB&R-DBFS-RP algorithm on

average took 1.9 seconds. Although the experiments in these papers were executed on different platforms, the average running time for the two variations of the BB&R algorithm were two orders of magnitude faster, which clearly dominates the results in M'Hallah and Bulfin (2007).

Table 3.4: $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Average CPU Time (sec.) with Different Dominance Rules

$n - p_{\min} - p_{\max}$	w/o EJR	w/o IRT	w/o NDDOR	w/o GMDR
80-0-100	0.5	0.5	0.5	13.6 (**7**)
80-25-75	0.5	0.5	0.5	12.3 (**2**)
100-0-100	0.8	1.1	2.3	67.6 (**27**)
100-25-75	0.6	0.6	0.5	42.0 (**15**)
120-0-100	0.8	1.4	1.1	84.3 (**58**)
120-25-75	0.8	1.0	0.7	64.5 (**41**)
140-0-100	1.1	2.3	1.4	-
140-25-75	1.0	1.6	1.0	-
160-0-100	1.6	3.7	2.4	-
160-25-75	1.2	2.3	1.1	-
180-0-100	2.1	5.5	3.5	-
180-25-75	1.7	3.4	1.7	-
200-0-100	3.9	10.1	7.6	-
200-25-75	3.0	5.9	2.8	-
220-0-100	4.5	13.4	7.6	-
220-25-75	3.5	7.9	3.2	-
240-0-100	7.5	20.9	14.8	-
240-25-75	4.9	11.4	4.5	-
260-0-100	10.5	30.6	24.3	-
260-25-75	6.5	15.6	6.2	-
280-0-100	18.3	49.4	45.7	-
280-25-75	10.1	22.7	9.6	-
300-0-100	24.5	61.8	41.8	-
300-25-75	14.7	30.9	14.8	-

Table 3.4 and 3.5 reports the average and maximum running time for the BB&R-DBFS algorithm with RP best measure when each of the dominance rules described in Section 3.1 and 3.2 is removed individually. The number in parenthesis in Table 3.4 reports the number of unsolved instances. These two tables provide an insight to the relative impact of each of the dominance rules. The GMDR has the largest

Table 3.5: $1|r_i|\sum U_i$ BB&R-DBFS Algorithms: Maximum CPU Time (sec.) with Different Dominance Rules

$n - p_{\min} - p_{\max}$	w/o EJR	w/o IRT	w/o NDDOR	w/o GMDR
80-0-100	6.4	4.6	6.1	849.6
80-25-75	1.2	1.3	1.4	1653.1
100-0-100	75.5	78.7	590.8	1834.6
100-25-75	3.9	2.7	4.6	2755.4
120-0-100	9.4	9.6	27.6	1784.5
120-25-75	7.8	6.0	6.5	2096.0
140-0-100	10.4	21.7	21.6	-
140-25-75	13.5	16.9	11.1	-
160-0-100	26.8	48.9	45.5	-
160-25-75	8.5	12.1	15.0	-
180-0-100	25.8	56.2	153.3	-
180-25-75	14.1	23.7	16.1	-
200-0-100	161.5	283.7	457.5	-
200-25-75	177.0	167.7	146.9	-
220-0-100	65.6	156.8	190.6	-
220-25-75	43.8	66.8	43.7	-
240-0-100	309.7	474.7	905.6	-
240-25-75	49.0	83.3	53.5	-
260-0-100	156.0	349.6	957.2	-
260-25-75	67.0	118.4	70.1	-
280-0-100	397.6	1027.1	1563.8	-
280-25-75	195.6	291.0	185.9	-
300-0-100	1787.1	1813.5	1126.4	-
300-25-75	299.2	265.8	405.6	-

impact on the performance of the BB&R-DBFS algorithm. Without GMDR, the algorithm has difficulty solving all the test instances. At size $n = 80, 100, 120$, it was only able to solve 98.5%, 93%, and 83.5% of the test instances respectively. By using the GMDR, the BB&R-DBFS algorithm has over two orders of magnitude speed up in computational time over the computational time when the GMDR is not in use.

Despite the overwhelming impact of the GMDR, the other dominance rules also have a significant impact on the performance of the BB&R-DBFS algorithm as well. On average, the BB&R-DBFS algorithm takes two times longer when the EJR or the NDDOR is not used. For the larger instances, the BB&R-DBFS algorithm takes

up to five times longer when ITR is not used. Table 3.5 shows that for the difficult instances, each of the individual dominance rules can have a significant impact on the degradation in the performances of the BB&R-DBFS algorithm. Table 3.4 and 3.5 shows that on average, the ITR rule has a stronger impact than the EJR and NDDOR on the performances of the BB&R-DBFS algorithm. However, in the worst cases, Table 3.5 shows that both NDDOR and ITR have a significant impact on the performances of the BB&R-DBFS algorithm.

In addition to the computational results for the BB&R algorithm, the test set was also used to assess the effectiveness of the EDP described in Section 3.3.1. The column labeled EDP in Table 3.6 reports the number of instances for which EDP provided a tighter upper bound than MLH. The column labeled MLJ reports the number of instances for which MLJ provided a tighter upper bound than EDP. Moreover, the column labeled EDP Optimal in Table 3.6 also reports the number of times the upper bound estimated by the EDP is equal to the optimal number of tardy jobs, and also the average gap between the optimal number of tardy jobs and the estimated upper bound.

The EDP heuristics clearly outperformed the MLJ heuristic. The EDP was able to consistently find tighter upper bounds. For more than half of the smaller instances, EDP was able to find the optimal solution. The tighter upper bound also contributed to the overall effectiveness of the BB&R algorithm.

3.5 Conclusion

This chapter presents the BB&R algorithm using the DBFS exploration strategy to solve the $1|r_i|\sum U_i$ scheduling problem. The chapter provides enhancements to two previously known dominance rules (reported by Baptiste et al. [6] and Dauzère-Pérès and Sevaux [27], respectively) and describes a new memory-based dominance rule.

Table 3.6: EDP vs. MLJ Upper Bounds Comparison for the $1|r_i|\sum U_i$ Scheduling Problem

$n - p_{\min} - p_{\max}$	EDP	MLJ	EDP Optimal	Avg. Gap
80-0-100	228	8	190	0.41
80-25-75	174	7	186	0.41
100-0-100	236	9	162	0.56
100-25-75	194	10	174	0.45
120-0-100	251	7	133	0.77
120-25-75	218	10	149	0.57
140-0-100	270	7	126	0.78
140-25-75	221	7	148	0.64
160-0-100	280	4	114	0.93
160-25-75	242	7	118	0.8
180-0-100	271	6	103	1.06
180-25-75	247	4	111	0.92
200-0-100	282	1	82	1.22
200-25-75	257	6	100	0.99
220-0-100	282	3	85	1.29
220-25-75	259	6	88	1.17
240-0-100	286	4	88	1.33
240-25-75	264	3	87	1.21
260-0-100	290	1	73	1.49
260-25-75	264	11	77	1.35
280-0-100	292	2	68	1.65
280-25-75	275	4	64	1.5
300-0-100	293	4	66	1.88
300-25-75	269	4	71	1.57

This chapter describes how these dominance rules can be embedded in a new B&B algorithm using an effective DBFS exploration strategy. The resulting new algorithm, BB&R, is proven to be exact. The BB&R-DBFS solved all 7,200 randomly generated test instances to optimality, outperforming the current best known exact algorithms. Furthermore, the LA-NDDOR extension provided additional computational speed up. The running time for the LA-NDDOR extension in conjunction with the DBFS exploration strategy is an order of magnitude faster than the BB&R-DFS variation. The ITR rule and the NDDOR have the most impact on the performances of the BB&R-DBFS algorithm.

Like all B&B algorithms, improving the lower and upper bounds reduces its execution time. An extended dynamic programming algorithm is also presented and is shown to significantly improve the upper bound estimation. The combination of all the dominance rules, new exploration strategy, and improved upper bound computation demonstrate that the BB&R algorithm is very efficient.

The BB&R algorithm has been successfully applied to the $1|r_i|\sum U_i$ scheduling problem. A natural extension to this work is to investigate other applications. One immediate extension of the $1|r_i|\sum U_i$ scheduling problem is to investigate the total tardiness scheduling problem, $1|r_i|T$ [51]. Chapter 4 introduces a BB&R algorithm for the total tardiness scheduling problem and shows its effectiveness. There are also practical military applications of the $1|r_i|\sum U_i$ scheduling problem that involve limited, highly valued assets that process certain tasks within a given time window (e.g., a surveillance satellite that must photograph as many locations as possible within its overpass time window.) Another military application of a limited, high value asset is the Air Force Airborne laser (ABL) system, which employs a 100 ton system of chemical lasers encased within a Boeing 747 aircraft designed for theater ballistic missile (TBM) defense. In particular, the ABL system is designed to detect the launch of a TBM, track its trajectory, and then destroy the missile using a high powered (megawatt class) laser (which is achieved by heating the TBM's own fuel supply until the fuel explodes and destroys the missile.) Therefore, once the TBM's boost phase is complete, the ABL system is no longer effective. The monetary cost and critical military mission of the ABL system makes its optimal utilization a strategic military priority.

Chapter 4

The $1|r_i|\sum t_i$ Scheduling Problem

Chapter 3 introduced a modified B&B algorithm, called the BB&R algorithm, that uses the Distributed Best First Search (DBFS) exploration strategy, which is a hybrid between Best-First Search (BFS) and Depth-First Search (DFS) [91]. The DBFS exploration strategy was incorporated with the BB&R algorithm to solve the $1|r_i|\sum U_i$ scheduling problem. In particular, the algorithm was able to solve problem instances with up to 300 jobs, outperforming the best known algorithms reported in the literature [27, 6, 76]. Memory-based dominance rules, that store (i.e., remember) sub-problems that have already been generated (and hence, the name Branch, Bound, and Remember) are also incorporated into the BB&R algorithm. Lastly, the BB&R algorithm with DBFS offers several advantages over the more traditional DFS or BFS. In particular, DBFS is able to find optimal solutions earlier in the search process, and by design, it explores fewer sub-problems that will eventually be dominated by another sub-problem, and hence, reducing the number of branches.

The BB&R algorithm with the DBFS exploration strategy is used in this chapter to solve the $1|r_i|\sum t_i$ scheduling problem. A modified dynamic programming algorithm is also presented to efficiently compute tighter bounds. Several previously known dominance rules proposed by Jouglet et al. [50], Baptiste et al. [4] and Chu [16] are also incorporated into the BB&R algorithm.

This chapter is organized as follows. Section 4.1 describes the $1|r_i|\sum t_i$ scheduling problem, and the notation used in this chapter. Section 4.2 describes the dominance rules used for the $1|r_i|\sum t_i$ scheduling problem, as well as a proof showing that the

combination of using all these dominance rules preserves exactness. Section 4.3 describes the bounding schemes incorporated in the BB&R algorithm, including a more efficient implementation of a modified dynamic programming algorithm for computing the lower bounds. Section 4.4 provides an overview of the BB&R algorithm. Computational results are reported in Section 4.5, followed by conclusions in Section 4.6.

4.1 Background and Notations

The scheduling problem addressed in this chapter is a single machine scheduling problem, denoted as $1|r_i|\sum t_i$ [51]. The problem consists of a set of jobs $J = \{1, 2, \ldots, n\}$ to be scheduled in sequence, where associated with each job is a release time r_i, a processing time p_i, and a due-date d_i, for $i = 1, 2, \ldots, n$, where all parameters are positive integers. A job cannot be started before its release date. Tardiness of a job $i \in J$ is defined as $t_i = \max(0, c_i - d_i)$, where c_i is the completion time of job i. The objective of the $1|r_i|\sum t_i$ scheduling problem is to minimize the total tardiness $\sum t_i$. Rinnooy Kan [51] proves this problem to be NP-hard in the strong sense.

A well-studied variation of the $1|r_i|\sum t_i$ scheduling problem is the $1||\sum t_i$ scheduling problem, where all jobs have equal release dates. The $1||\sum t_i$ is also known to be NP-hard [31]. Several dominance rules for the equal release date problem have been proposed in the literature [34, 96]. Exact algorithms such as dynamic programming algorithms and B&B algorithms have also been proposed by Lawler [65], Potts and Van Wassenhove [88], Szwarc et al. [96], and Chang et al. [14].

The $1|r_i|\sum t_i$ scheduling problem considered in this chapter has received less research attention. Chu and Portmann [17] and Chu [16] propose sufficient conditions for local optimality, and develop B&B algorithms in conjunction with dominance rules for solving this problem. The best known B&B algorithm is reported by Baptiste et

al. [4] and Jouglet et al. [50], where the algorithm was able to solve problems with up to 50 jobs (for the hardest instances) and was tested with up to 500 jobs (for the easiest instances).

The following notation and assumptions are used in the remainder of the chapter. Jobs are assumed to be sorted by earliest due-date order, and ties are broken based on the release time followed by the processing time (i.e., $i < j \Rightarrow d_i < d_j \vee (d_i = d_j \wedge r_i < r_j) \vee (d_i = d_j \wedge r_i = r_j \wedge p_i < p_j)$. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ be a sequence of scheduled jobs, where $\sigma_i \in J$ for $i = 1, 2, \ldots m$. For a set of jobs $J' \subseteq J$, let

- $r(J') = \min_{j \in J'} r_j$,
- $p(J') = \sum_{j \in J'} p_j$,
- $d(J') = \max_{j \in J'} d_j$.

Let

- $c_\sigma = c(\sigma)$ denote the completion time of the sequence of scheduled jobs σ,
- c_{σ_i} denote the completion time of job σ_i (define $c_{\sigma_0} = 0$),
- s_{σ_i} denote the start time of job σ_i (define $s_{\sigma_{m+1}} = c_\sigma$),
- $F_\sigma = F(\sigma)$ denote the set of unscheduled (free) jobs,
- $T_\sigma = T(\sigma) = \sum_{j \in \sigma} t_j$ denote the total tardiness of the jobs in σ,
- $\hat{r}_\sigma = \max\{c_\sigma, \min_{i \in F_\sigma} r_i\}$ denote the earliest start time of the free jobs,
- $T_{jk}(\bar{r}) = \max(0, \max(r_j, \bar{r}) + p_j - d_j) + \max(0, \max(\max(r_j, \bar{r}) + p_j, r_k) + p_k - d_k)$ denote the tardiness of job j and job k when scheduling job j immediately before job k given that the machine becomes available at time $\bar{r}$,

- $C_{jk}(\bar{r}) = \max(\max(r_j, \bar{r}) + p_j, r_k) + p_k$ denote the completion time of job k when immediately preceded by job j given that the machine becomes available at time $\bar{r}$.

A state in the BB&R algorithm will be represented by $(\sigma, F_\sigma, T_\sigma, \hat{r_\sigma})$.

4.2 Dominance Rules

This section presents several dominance rules used in the BB&R algorithm for the $1|r_i|\sum t_i$ scheduling problem. As described in Section 3.1, dominance rules are properties that exploit the structure of optimal solutions, and hence, can be used as pruning strategies. These rules identify specific properties that at least one optimal solution must satisfy. These rules can prune many solutions, including optimal solutions; however, they will not prune all optimal solutions. These dominance rules are designed to provide a significant reduction in the search space.

In order to describe these dominance rules, define an *active schedule* as a schedule such that no jobs in the schedule can be scheduled earlier without causing a delay for another job. In addition, a set of schedules is said to be *dominant* if it contains at least one optimal schedule.

The BB&R algorithm in this chapter uses several dominance rules proposed in Chu [16] and Jouglet et al. [50]. *Individually*, these dominance rules have been shown to be exact (i.e., at least one optimal solution must satisfy a specific dominance rule). These dominance rules have been modified such that they can be combined. This section provides a proof showing that the combination of dominance rules used in this BB&R algorithm can be used simultaneously without pruning all optimal solutions.

Chu and Portmann [17] describe a sufficient condition for local optimality for the total tardiness criterion. Jouglet et al. [50] expand on this work and provide a

necessary and sufficient condition for local optimality and define a dominant subset of schedules based on this necessary and sufficient condition, which is now formally stated.

Definition 4.2.1 *Jouglet et al. Necessary and Sufficient Condition* [50]
An active schedule S is said to be Locally Optimal Well Sorted (LOWS-active) if every pair of adjacent jobs j and k satisfy at least one of the following conditions:

1. $T_{jk}(\bar{r}) < T_{kj}(\bar{r})$ *(where* $\bar{r}$ *is the completion time of the job preceding job j),*
2. $T_{jk}(\bar{r}) = T_{kj}(\bar{r})$ *and* $\max(r_j, \bar{r}) \leq \max(r_k, \bar{r})$,
3. $T_{jk}(\bar{r}) > T_{kj}(\bar{r})$ *and* $\max(r_j, \bar{r}) < \max(r_k, \bar{r})$.

Theorem 4 states that given any schedule S, there exists a LOWS-active schedule that is at least as good as S.

Theorem 4 [50] *The subset of LOWS-active schedules is dominant for the one machine total tardiness problem.*

A modified LOWS-active schedule criterion is used in the BB&R algorithm presented in this chapter. This modification is necessary in order to prove the exactness of combining several other dominance rules used in the BB&R algorithm. Prior to presenting the modified LOWS-active schedule criterion, the following total order is defined.

Definition 4.2.2 *Given two partial sequences of jobs* $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ *and* $\theta = (\theta_1, \theta_2, \ldots, \theta_m)$, σ *precedes* θ, *denoted as* $\sigma \rightarrow \theta$, *if either* $c_{\sigma_k} = c_{\theta_k}$, *for* $k = 1, 2, \ldots, m$ *or there exists j such that* $c_{\sigma_j} < c_{\theta_j}$ *and* $c_{\sigma_k} = c_{\theta_k}$, *for* $k = j+1, j+2, \ldots, m$.

If $\sigma \rightarrow \theta$ and $\theta \rightarrow \sigma$, then $\sigma \leftrightarrow \theta$. However, if $\theta \nrightarrow \sigma$, then σ strictly precedes θ. The modified LOWS-active schedule criterion, termed *LOWS*-active* is now formally defined.

Definition 4.2.3 *An active schedule S is said to be LOWS*-active if every pair of adjacent jobs j and k satisfy at least one of the following conditions:*

1. $T_{jk}(\bar{r}) < T_{kj}(\bar{r})$ *(where $\bar{r}$ is the completion time of the job preceding job j),*

2. $T_{jk}(\bar{r}) = T_{kj}(\bar{r})$ *and* $[(C_{jk}(\bar{r}) < C_{kj}(\bar{r})) \vee \{(C_{jk}(\bar{r}) = C_{kj}(\bar{r})) \wedge ((p_j < p_k) \vee (p_j = p_k \wedge j < k))\}]$,

3. $T_{jk}(\bar{r}) > T_{kj}(\bar{r})$ *and* $C_{jk}(\bar{r}) < C_{kj}(\bar{r})$.

Note that the modification in the LOWS*-active schedule criterion compared to the LOWS-active schedule is a minor change, however, this modification is necessary to prove the exactness of the algorithm presented in this chapter, in combination with the other dominance rules. The following proposition is needed to prove that the subset of LOWS*-active schedules is dominant.

Proposition 1 *Suppose $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ is a partial sequence of jobs such that σ_i and σ_{i+1} do not satisfy the LOWS*-active criterion. Let $\sigma' = (\sigma_1, \sigma_2, \ldots, \sigma_{i-1}, \sigma_{i+1}, \sigma_i, \sigma_{i+2}, \ldots, \sigma_m)$ be the sequence of jobs obtained from σ by interchanging jobs σ_i and σ_{i+1}. Then either $T(\sigma') < T(\sigma)$ or $T(\sigma') = T(\sigma)$ and $\sigma' \rightarrow \sigma$.*

Proof: To simplify the notation, let $j = \sigma_i$, $k = \sigma_{i+1}$, and $\bar{r} = c_{\sigma_{i-1}}$. All three of the LOWS*-active conditions must be violated. Condition (1) of Definition 4.2.3 is violated implies that $T_{jk}(\overline{r}) \geq T_{kj}(\overline{r})$.

Case 1: $T_{jk}(\overline{r}) > T_{kj}(\overline{r})$. Condition (3) of Definition 4.2.3 is violated implies that $C_{jk}(\overline{r}) \geq C_{kj}(\overline{r})$, which implies that interchanging jobs j and k will decrease the total tardiness, i.e., $T(\sigma') < T(\sigma)$.

Case 2: $T_{jk}(\bar{r}) = T_{kj}(\bar{r})$. Condition (2) of Definition 4.2.3 is violated implies that $C_{jk}(\bar{r}) \geq C_{kj}(\bar{r})$.

Case 2a: $C_{jk}(\bar{r}) > C_{kj}(\bar{r})$. Interchanging jobs j and k will not increase the total tardiness, nor will it increase the completion times of jobs $\sigma_{i+2}, \sigma_{i+3}, \ldots, \sigma_m$, but it will decrease the completion time of the job in position $i+1$ in σ'. Therefore, $\sigma' \rightarrow \sigma$.

Case 2b: $C_{jk}(\bar{r}) = C_{kj}(\bar{r})$. Condition (2) of Definition 4.2.3 is violated implies that $p_j \geq p_k$. Interchanging jobs j and k will not increase the total tardiness, nor will it change the completion times of the jobs in positions $i+1, i+2, \ldots, m$. The completion time of the job in position i in σ' is less than or equal to c_{σ_i}, and hence, $\sigma' \rightarrow \sigma$. □

Proposition 1 states that interchanging any adjacent jobs in a non-LOWS*-active schedule will either decrease the total tardiness or decrease the order of the sequence defined by Definition 4.2.2. Theorem 5 formally states that the subset of LOWS*-active schedules is dominant.

Theorem 5 *Any sequence of jobs σ can be transformed via a series of pairwise interchanges into a sequence of jobs σ' such that σ' is LOWS*-active and $T(\sigma') \leq T(\sigma)$.*

Proof: Proposition 1 shows that interchanging a pair of adjacent jobs that violate the LOWS*-active criteria will either strictly decrease the total tardiness or leave it unchanged. Only a finite number of interchanges can be made that decrease the total tardiness. There can only be a finite number of interchanges between two interchanges that decrease the total tardiness since each such interchange results in a new sequence that precedes the old one. □

The next dominance rule presented is a memory-based dominance rule. Unlike the LOWS*-active schedule criterion, a memory-based dominance rule compares two partial sequences to determine if one dominates the other. Memory-based dominance rules are not new; Baptiste et al. [4] and Jouglet et al. [50] used a similar memory-based dominance rule in their algorithm, which they termed "better sequence". The

Memory Dominance Rule (MDR) presented can be combined with the other dominance rules presented such that the full BB&R algorithm is proven to be exact. The following definition defines the MDR used within the BB&R algorithm.

Definition 4.2.4 Memory Dominance Rule (MDR)
Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ and $\delta = (\delta_1, \delta_2, \ldots, \delta_m)$ be two LOWS-active partial schedules such that $\{\sigma_1, \sigma_2, \ldots, \sigma_m\} = \{\delta_1, \delta_2, \ldots, \delta_m\}$. Then σ dominates δ if at least one of the following conditions is satisfied.*

1. *$T_\sigma < T_\delta$ and $\hat{r}_\sigma \leq \hat{r}_\delta$,*
2. *$T_\sigma = T_\delta$ and $\hat{r}_\sigma < \hat{r}_\delta$,*
3. *$T_\sigma = T_\delta$ and $\hat{r}_\sigma = \hat{r}_\delta$ and $\sigma \rightarrow \delta$.*

To prove that using the MDR with the LOWS*-active schedule criterion will not prune out all optimal solutions, the following definitions are needed.

Definition 4.2.5 *A sequence is a minimal element in a set of sequences if it is in the set and if it is not strictly preceded by any other sequence in the set. Let*

- *Ω denote the set of all optimal sequences,*
- *$\Omega^1 \subset \Omega$ denote the set of all optimal sequences that are LOWS*-active,*
- *$\Omega^2 \subset \Omega^1$ denote the minimal elements of optimal LOWS*-active sequences.*

Theorem 6 *If $\theta = (\theta_1, \theta_2, \ldots, \theta_n) \in \Omega^2$ and $\theta^m = (\theta_1, \theta_2, \ldots, \theta_m)$ is dominated by another sequence $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ by the MDR, then σ is a subsequence of a sequence in Ω^2.*

Proof: Let $\sigma' = (\sigma_1, \sigma_2, \ldots, \sigma_m, \theta_{m+1}, \theta_{m+2}, \ldots, \theta_n)$. σ dominates θ^m implies that $\hat{r}_\sigma \leq \hat{r}_{\theta^m}$. Therefore, the completion times of $\theta_{m+1}, \theta_{m+2}, \ldots, \theta_n$ in σ' are less than or

equal to their respective completion times in θ. Furthermore, $T_\sigma \leq T_{\theta^m}$ implies that $T_{\sigma'} \leq T_\theta$, and hence, σ' is an optimal sequence. Since θ is optimal, then $T(\sigma) = T(\theta^m)$.

Now suppose that $c_{\sigma_m} < c_{\theta_m}$. If any job in $(\theta_{m+1}, \ldots, \theta_n)$ can be shifted to start earlier in σ' than in θ, then σ' strictly precedes θ, and by Proposition 1, σ' can be transformed into a LOWS*-active sequence that strictly precedes θ. However, this is a contradiction, since $\theta \in \Omega^2$. Therefore, no job in $(\theta_{m+1}, \theta_{m+2}, \ldots, \theta_n)$ can be shifted to start earlier. In particular, θ_{m+1} can not be shifted to start earlier, and hence , $s_{\theta_{m+1}} = r_{\theta_{m+1}}$, which implies that θ_{m+1} cannot be interchanged with σ_m in σ'. Therefore, σ' is a LOWS*-active, optimal schedule that strictly precedes θ, which is also a contradiction. Therefore, $c_{\sigma_m} = c_{\theta_m}$.

It has been established that $T(\sigma) = T(\theta^m)$ and $c_{\sigma_m} = c_{\theta_m}$, which implies that $\widehat{r}_\sigma = \widehat{r}_{\theta^m}$. Thus σ dominates θ^m implies that $\sigma \rightarrow \theta^m$. Therefore, $\sigma' \rightarrow \theta$. σ' can be transformed into an LOWS*-active sequence σ'' that precedes σ' and is optimal. σ'' cannot strictly precede θ because $\theta \in \Omega^2$. Hence $\sigma'' \leftrightarrow \sigma' \leftrightarrow \theta$. □

Two other supplementary dominance rules are used in the BB&R algorithm, namely the First Job Rule (FJR) [16] and the Equal Length Job Rule (ELJR) [4]. These rules are now formally presented.

Theorem 7 First Job Rule (FJR)[16]
If there is a job i such that $i \in J$, and for all jobs $j \in J$, $p_i \leq p_j, d_i \leq d_j$, then there is an optimal schedule in which job i precedes any job k such that $r_i \leq r_k$.

In case more than one job satisfies the FJR conditions, then the one with the smallest index will be chosen.

The FJR suggests that certain jobs must be scheduled prior to scheduling any other jobs. To show that the FJR can be used with the LOWS*-active schedule criterion, the following theorem formally states that all LOWS*-active schedules also satisfy FJR.

Theorem 8 *All LOWS*-active schedules satisfy the FJR.*

Proof: Suppose σ is a LOWS*-active schedule that does not satisfy the FJR. Let k be the job with the smallest index such that $p_k \leq \min_{i \in J} p_i$ and $d_k \leq \min_{i \in J} d_i$. Let i be the job such that $r_k \leq r_i$ and job i precedes job k in σ. Let j be the job immediately preceding job k in σ and $\bar{r}$ be the completion time of the job immediately preceding job j. Job i precedes job k and $r_k \leq r_i$ implies that $r_k \leq s_j$, and hence, jobs j and k can be interchanged without increasing the completion time of any jobs (i.e., $C_{jk}(\bar{r}) \geq C_{kj}(\bar{r})$). $p_k \leq p_j$ and $d_k \leq d_j$ imply that $T_{jk}(\bar{r}) \geq T_{kj}(\bar{r})$. Therefore, jobs j and k do not satisfy conditions (1) or (3) of the LOWS*-active criterion, and hence, jobs j and k must satisfy condition (2). However, the only way that condition (2) can be satisfied when $C_{jk}(\bar{r}) \geq C_{kj}(\bar{r})$ is to have $p_j < p_k$ or $p_j = p_k$ and $j < k$. Neither of these are possible since $p_k \leq p_j$, $d_k \leq d_j$, and k has the smallest index of any job that satisfies the FJR conditions. □

Although the FJR is stated in terms of the original problem, it can also be applied to sub-problems in the BB&R algorithm. Given a partial sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$, if there exists a job k such that $p_k \leq p_j$ and $d_k \leq d_j$ for all jobs $j \in F_\sigma$ and $r_k \leq \hat{r}_\sigma$, then job k can be scheduled before all the other jobs in F_σ.

Another dominance rule used in the BB&R algorithm is the ELJR, which was originally proposed by Baptiste et al. [4].

Theorem 9 Equal Length Job Rule (ELJR)[4]
Let i and k be two jobs such that $p_i = p_k$. If $r_i \leq r_k$ and $d_i \leq \max(r_k + p_k, d_k)$, then there exists an optimal schedule in which job i precedes job k.

Theorem 10 formally states that the ELJR can also be combined with the LOWS*-active schedule criterion, the MDR, and the FJR.

Theorem 10 *There exists an optimal schedule in Ω^2 that satisfies the ELJR.*

Proof: Let σ be an optimal sequence that is not strictly preceded by any other optimal sequence (i.e., σ is a minimal element). Let σ' be obtained by interchanging jobs until the ELJR is satisfied. Then σ' is optimal and is not strictly preceded by any other optimal sequence. To complete the proof, it must be shown that σ' is a LOWS*-active schedule or can be transformed to be LOWS*-active.

Suppose σ' is not a LOWS*-active schedule. Suppose jobs j and k are adjacent jobs in σ' that do not satisfy the LOWS*-active conditions. Let σ'' be obtained from σ' by interchanging jobs j and k. Let $\bar{r}$ be the completion time of the job immediately preceding job j.

Case 1: $T_{jk}(\bar{r}) > T_{kj}(\bar{r})$. The proof of Proposition 1 shows that the total tardiness of σ'' is less than σ', which contradicts that σ' is optimal.

Case 2: $T_{jk}(\bar{r}) = T_{kj}(\bar{r})$.

Case 2a: : $C_{jk}(\bar{r}) > C_{kj}(\bar{r})$. This is a contradiction since this implies that σ'' strictly precedes σ'.

Case 2b: $C_{jk}(\bar{r}) = C_{kj}(\bar{r})$.

Case 2bi: $p_j > p_k$. This is a contradiction since this implies that σ'' strictly precedes σ'.

Case 2bii: $p_j = p_k$. Since jobs j and k do not satisfy the LOWS*-active schedule criterion, then $j > k$. This implies that $d_j \geq d_k$ due to the order in which the jobs are sorted. If $r_j \geq r_k$, then jobs j and k do not satisfy the ELJR, which is a contradiction. If $r_j < r_k$, then $d_j > d_k$ (due to the order in which the jobs are sorted and $j > k$), so j and k can be interchanged without violating the ELJR. After all such interchanges have

been performed, σ' will be a LOWS*-active, optimal schedule that satisfies the ELJR. □

All the dominance rules presented in this section are combined and used in the BB&R algorithm presented in this chapter. Theorems 5-10 establish that these dominance rules can be combined such that there must be at least one optimal solution that remains unpruned by these rules. Note that there are other dominance rules presented in the literature that are not used in this BB&R algorithm. For example the generalized Emmons rules [4] and Theorem 3 of Chu [16] were not incorporated into the BB&R algorithm. Some of the dominance rules in Jouglet et al. [50] based on insertion and interchanging of jobs were also not included because they do not fit well into the exploration strategy described in Section 4.4. Moreover, additional dominance rules increase the difficulty in finding a proof of exactness for combining additional rules, and it is not clear whether such a proof or a counterexample exists. It may be possible to combine additional dominance rules to the BB&R algorithm. However, the proofs presented in this chapter will not guarantee that the BB&R algorithm will remain exact with these other dominance rules. Prior to outlining the BB&R algorithm, the next section presents the different upper and lower bound algorithms used in the BB&R algorithm.

4.3 Bounding Scheme

This section provides an overview of the algorithms for computing the upper and lower bounds used in the BB&R algorithm for the $1|r_i|\sum t_i$ scheduling problem. Two upper bound algorithms proposed by Chu [16], namely the IPRTT and the NDPRTT are used to compute the initial upper bound. Two lower bound algorithms, including a modified dynamic programming algorithm originally proposed by Lawler [65] and a lower bound algorithm proposed by Baptiste et al. [4] are used to compute the lower

bound. A brief overview of each of the bounding algorithms is provided.

The two upper bound algorithms, IPRTT and NDPRTT, are Greedy algorithms. Both of these algorithms are based on a function called *Priority Rule for Total Tardiness* (PRTT) [16], defined as

$$\text{PRTT}(i, \Delta) = \max(r_i, \Delta) + \max(\max(r_i, \Delta) + p_i, d_i),$$

where $i \in J$ and Δ is the time at which the machine becomes available. Theorem 11, presented in Chu [16] uses the PRTT function to define a locally sufficient condition for optimality.

Theorem 11 [16]

Given only two jobs i and j to be scheduled on a machine that becomes available at time Δ, the sufficient condition for processing job i before job j in order to obtain an optimal solution is $PRTT(i, \Delta) \leq PRTT(j, \Delta)$.

At each iteration, the IPRTT attempts to schedule a job k with the current minimum PRTT function value. It then attempts to insert any unscheduled jobs that can be completed before job k. If no such jobs can be inserted before job k, the process is repeated until all jobs are scheduled.

The NDPRTT schedules jobs based on the earliest release time, ties are broken based on the smallest PRTT function value, and further ties are broken based on smaller processing times. Baptiste et al. [4] propose a lower bound based on the Generalized Emmons Rule. The following two propositions are used in their lower bound algorithm.

Proposition 2 [4]

Let j and k be two jobs such that $r_j \leq r_k, p_j \leq p_k$, and $d_j \leq d_k$. Then there exists an optimal schedule in which job k begins after the end of job j.

Proposition 3 [4]

Let j and k be two jobs such that $r_j \leq r_k, p_j \leq p_k$, and $d_j > d_k$. Then exchanging d_j and d_k does not increase the optimal total tardiness.

The Baptiste et al. [4] algorithm allows preemption, and modifies the due-date of each job based on the current schedule. The modified due-dates are lower bounds such that the computed total tardiness will not overestimate the true optimal total tardiness. For the remainder of the chapter, this algorithm will be referred to as the BLB.

Another lower bound for the $1|r_i|\sum t_i$ scheduling problem can be obtained by relaxing the release times to all be zero. The resulting $1||\sum t_i$ scheduling problem can be solved in $O(n^4 \sum_{i=1}^{n} p_i)$ time using Lawler's dynamic program, but the running time may be too slow for the lower bound to be useful. Instead, a branch and remember (B&R) algorithm was used to solve the relaxed problem. The method of branching is based on the decomposition method that Lawler used in his dynamic program and the improvements developed by Chang et al. [14]. Furthermore, the states of this B&R algorithm are saved from the first time the lower bound algorithm is called until the last time the lower bound algorithm is called. Therefore, as the overall BB&R algorithm proceeds, the lower bound algorithm builds a database of states. Many of the sub-problems for which lower bounds must be calculated are very similar to each other, and hence, they share many states. The optimal solution for the shared states do not have to be recomputed because they are stored in the database. This approach greatly reduces the total computational effort required to compute the lower bounds. Other more sophisticated exact algorithms for solving the $1||\sum t_i$ scheduling problem have also been proposed in the literature. Szwarc et al. [96, 97] present different B&B algorithms for solving the $1||\sum t_i$ problem. In addition to the decomposition methods used by Lawler [65] and Chang et al. [14], Szwarc et al. [96] use additional decomposition rules that further eliminate possible positions

for scheduling jobs. It may be possible to incorporate the rules used in these exact algorithms for solving the $1||\sum t_i$ problem with the BB&R algorithm, to obtain a even better overall performance.

The modified dynamic programming algorithm can also be used to estimate tighter lower bounds based on decomposing unscheduled jobs into smaller groups. The set of unscheduled jobs are broken into groups based on the following steps:

Step 1: Sort all unscheduled jobs in earliest due-date order, $(j_1, \ldots, j_m)$.

Step 2: Let $\delta = r_{j_1}$.

Step 3: Let the current earliest unscheduled job be j_i; add j_i to the current group.

Step 4: Remove j_i from the set of unscheduled jobs.

Step 5: $\delta = \delta + p_i$.

Step 6: If $r_{i+1} > \delta$, then start a new group, and let $\delta = r_{i+1}$.

Step 7: Repeat from Step 3 until there are no more unscheduled jobs.

The modified dynamic programming algorithm is then used to compute the lower bound for each of the groups. The sum of all the total tardiness for each group is the new lower bound. For the remainder of the chapter, this new method for using Lawler's dynamic programming algorithm will be referred to as Decomp-DP.

4.4 Branch, Bound, and Remember Algorithm

This section introduces the BB&R algorithm for the $1|r_i|\sum t_i$ scheduling problem. The BB&R algorithm is an enumeration, divide and conquer technique. Like other B&B algorithm, the goal is to explore sub-problems until some of these sub-problems

may be fathomed, and hence, reduces the search space. The BB&R algorithm differs from other B&B algorithm in two fundamental ways. First, it incorporates the DBFS exploration strategy that determines which sub-problem to explore. Second, by design, the BB&R algorithm stores previously generated sub-problems such that memory-based dominance rules can be applied efficiently.

The BB&R algorithm is a constructive B&B algorithm. It enumerates the solution space by building a search tree, constructing feasible schedules by iteratively appending unscheduled jobs to partial schedules. Each internal node in the search tree is a sub-problem, while a leaf in the search tree corresponds to a feasible solution. The nodes in the search tree can be identified as states, $(\sigma, F_\sigma, T_\sigma, \hat{r}_\sigma)$. A new state is created by adding a new job to the partial sequence σ. The dominance rules are applied at each node, pruning possible branches along the search tree. Each visited node in the search tree is stored in a hash table, a data structure that provides an efficient look-up capability. By storing each node, the states are *remembered*, and hence, the MDR can be applied. Two lower bounds are computed at each node, and the maximum of the two is recorded.

The order in which the search tree is constructed can greatly affect the performance of any B&B algorithm. The BB&R algorithm uses the DBFS exploration strategy described in Section 3.3.2 for constructing the search tree. See Section 3.3.2 for a description of the DBFS exploration strategy and psuedo-code. For evaluation purposes, a DFS exploration strategy was also used in place of the DBFS exploration strategy. See Section 4.5 for a comparison of the computational performances between DBFS and DFS.

In the BB&R algorithm, a node in the search tree may be pruned in one of two ways. It can be pruned either by the computed bounds or by the dominance rules. The bounding scheme works as follow. Initially, prior to any branching, an upper bound is computed based on the IPRTT and NDPRTT algorithms; the minimum of

```
BB&R(σ, F_σ, T_σ, r̂_σ, hash_table, heap(1, ..., size(F_σ))
  LB = max(BLB(F_σ, r̂_σ), Decomp-LB(F_σ, r̂_σ))
  UB = min(IPPRT, NDPPRT)
  Initialize heap(0)
  while heap is not empty do
    for i = 0 → n do
      cur_state = heap(i).pop
      cur_lb = max(BLB(cur_state), Decomp-LB(cur_state))
      if (cur_lb + cur_state.T_σ) < UB then
        PF_σ = ITR(cur_state.F_σ) and FJR(cur_state.F_σ) and LOWS*(cur_state.F_σ)
        and ELJR(cur_state.F_σ)
        for each j ∈ PF_σ do
          new_state.σ = cur_state.σ + j
          update new_state from cur_state
          Violated_MDR= MDR(new_state)
          if not Violated_MBDR then
            Store(new_state) in hash_table
            heap(i+1).add(new_state)
          end if
        end for
      end if
    end for
  end while
```

Figure 4.1: BB&R Pseudo-Code for the $1|r_i|\sum t_i$ Scheduling Problem

these two bounds is retained. As the branching process proceeds with additional jobs being scheduled, lower bounds are computed using the BLB and the Decomp-DP. If the lower and upper bounds are tight, then the branch is pruned.

As mentioned above, the dominance rules are also used for pruning branches along the search tree. In addition, the dominance rule can also be used to filter jobs in F_σ that do not need to be considered as a next schedulable job. All of the dominance rules are applied upon visiting each node of the search tree. The dominance rules are applied in the following order: The FJR is first used to examine the set F_σ, it identifies a job that must be immediately scheduled next. If such a job exists, a single sub-problem is created by appending the job to σ. Otherwise, the Idle Time Rule (ITR)

is then used to reduce the number of jobs that can be considered as a candidate for being scheduled next. Let this set of jobs be denoted by PF_σ (possible first). The ITR removes jobs from F_σ that have release times greater than $\min_{j \in F_\sigma} \max(\hat{r}_\sigma, r_j) + p_j$. The LOWS*-active criterion is then applied to each of the jobs in PF_σ to further filter PF_σ. Lastly, the ELJR is then applied to the remaining jobs in PF_σ. If at any point PF_σ becomes empty, the entire branch is pruned. Sub-problems are created by appending the jobs that satisfy the FJR, the ITR, the ELJR, and the LOWS*-active criterion to σ, one job at a time. Let a new state be denoted as $(\sigma', F_{\sigma'}, T_{\sigma'}, \hat{r}_{\sigma'})$. The MDR is then applied to each new state. If a new state satisfies the MDR, then the new state is stored and later explored. Figure 4.1 depicts the pseudo-code for the BB&R algorithm with the DBFS implementation. Note that in addition to a hash table, a heap structure is also needed for the DBFS implementation to store states for each level of the search tree.

4.5 Computational Results

This section reports the computational results for the BB&R algorithm described in Section 4.4. All the dominance rules and the different bounding algorithms presented in this chapter have been incorporated into the BB&R algorithm. The BB&R algorithm is evaluated for 2280 randomly generated test instances, using the same scheme reported in Chu [16] and Baptiste et al. [4]. The generation scheme is based on four parameters: number of jobs, processing time range, α, and β, denoted as (n, [$p_{\min}$, $p_{\max}$], α, β). Each instance consists of three vectors, p, r, and d, which are randomly generated from three discrete uniform distributions. The processing times are randomly sampled from the set $\{1, 2 \ldots, 10\}$, the release times are randomly sampled from the set $\{0, 1, \ldots, \lfloor \alpha \sum p_i \rfloor\}$ and $d_i - (r_i + p_i)$ is randomly sampled from the set $\{0, 1, \ldots, \lfloor \beta \sum p_i \rfloor\}$. The parameters used for generating the test in-

Table 4.1: BLB and the Decomp-DP Lower Bounds Comparison for the $1|r_i|\sum t_i$ Scheuduling Problem in CPU Time (sec.)

	$\alpha = 0.5, \beta = 0.05$		$\alpha = 0.5, \beta = 0.25$		$\alpha = 0.5, \beta = 0.5$	
n	BLB	Decomp-DP	BLB	Decomp-DP	BLB	Decomp-DP
10	0.08	0.1	0.08	0.09	0.09	0.08
20	0.08	0.08	0.09	0.08	0.12	0.08
30	0.09	0.09	0.25	0.09	1.9	0.09
40	0.12	0.1	4.5	0.1	81.5	0.1
50	0.2	0.1	33.4	0.1	-	0.2

stances are $n = 10, 20, \ldots, 100, 120, \ldots, 200, 250, 300, 400, 500$, $[p_{\min}, p_{\max}] = \{[0, 10]\}$, $\alpha = 0, 0.5, 1.0, 1.5$, and $\beta = 0.05, 0.25, 0.5$. For each combination of parameter settings, 10 random instances are generated for a total of 2280 instances. Each instance in the test set is restricted to a one hour total processing time limit and a 8 million state space memory limit. All the experiments in the chapter were executed on a 2 GHz Pentium D using 1 GB of RAM.

Prior to examining the full power of the BB&R algorithm, the two different lower bound algorithms, namely the BLB and the Decomp-DP are tested individually using the DBFS exploration strategy in conjunction with all the dominance rules described in Section 4.2. Table 4.1 reports the average running time in CPU seconds for instance size $n = 10, 20, \ldots, 50$, with the hardest parameter setting, $\alpha = 0.5$ and $\beta = 0.05, 0.25, 0.5$. Note that in Table 4.1, all the test instances are solved to optimality except for the set of instances with $n = 50$, $\alpha = 0.5$, and $\beta = 0.5$, where BLB was only able to solve 20 percent of the test instances. The computational results for the larger instances are reported in Table 4.2. Table 4.2 reports the average running times and the percent of instances *solved*. The average running time is reported only for the set of instances where the complete set is solved to optimality.

The computational results reported in Table 4.1 and 4.2 show that the BB&R algorithm with the Decomp-DP lower bound algorithm provides significantly better results than the BB&R algorithm with the BLB lower bound algorithm. Table 4.1

Table 4.2: BLB and Decomp-DP Lower Bounds Comparison for the $1|r_i|\sum t_i$ Scheduling Problem in CPU Time (sec.) and Percentage Solved (Larger Instances)

	$\alpha = 0.5, \beta = 0.05$		$\alpha = 0.5, \beta = 0.25$		$\alpha = 0.5, \beta = 0.5$	
n	BLB	Decomp-DP	BLB	Decomp-DP	BLB	Decomp-DP
60	0.5	0.2	80%	0.2	0%	0.2
70	1.7	0.2	30%	0.2	0%	0.3
80	5.1	0.3	0%	0.9	0%	0.3
90	12.6	0.4	0%	1.0	0%	0.5
100	48	0.5	0%	1.6	0%	90%
150	30%	2.0	0%	3.7	0%	90%
200	0%	8.1	0%	13.7	0%	13.8
250	0%	19	0%	90%	0%	90%
300	0%	244	0%	245	0%	80%
400	0%	90%	0%	40%	0%	20%

shows that the running time when using the Decomp-DP lower bound algorithm scales more efficiently as the size of the problem instances increases. Moreover, Table 4.2 shows that when using the Decomp-DP lower bound algorithm with the BB&R algorithm, larger size instances can be solved to optimality. For $\alpha = 0.5$ and $\beta = 0.5$, the BB&R algorithm with the BLB lower bound algorithm is not able to solve any test instances with $n \geq 60$, while using the Decomp-DP lower bound algorithm, the BB&R algorithm is able to solve 80% of all instances with $n = 300$.

For further evaluation purposes, the BB&R algorithm is implemented using both the DBFS exploration strategy and the DFS exploration strategy. Let the BB&R algorithm using the DBFS exploration strategy be denoted as BB&R-DBFS, and let the BB&R algorithm using the DFS exploration strategy be denoted as BB&R-DFS. Tables 4.3 and 4.4 report the average and the maximum running time for the BB&R-DBFS and BB&R-DFS respectively. Note that the number represented in parenthesis denotes the number of test instances *solved*.

Tables 4.3 and 4.4 report the results for only the larger instances. For the smaller instances, with $n < 100$, the BB&R-DBFS algorithm was able to solve all instances to optimality with an average running time of 0.2 seconds and maximum running

time of 13 seconds. Instances are considered unsolved if a solution cannot be found within the one hour time limit or due to allocating more memory than available. In Table 4.3, all unsolved instances with $n \leq 400$ were due to the time restriction, while all unsolved instances with $n = 500$ were due to memory limitation. Note that in Table 4.3, though it may appear that as n increases, the problems are getting easier (since the average CPU times are shrinking; see in particular $n = 400, 500$), the test instances are in fact getting harder to solve (since fewer instances are being solved to optimality). To illustrate this point, for $\alpha = 0.5$ and $\beta = 0.05$, the average CPU time reported for $n = 500$ is 772 sec., which is less than 2034 sec., the average CPU time reported for $n = 400$. However, for $n = 500$, only one test instances is solved to optimality, while for $n = 400$, seven test instances are solved to optimality.

The computational results reported in Table 4.4 show that the DFS exploration strategy is substantially inferior to the DBFS exploration strategy. For $\alpha \neq 0$, the average running time for using the DBFS exploration strategy is faster or at least as good as the average running time for using the DFS exploration strategy. Note that for $\alpha = 0$, Lawler's dynamic programming algorithm [65] solves all instances to optimality, and hence, the exploration strategies do not affect the performance of the BB&R algorithm. In addition, Table 4.4 shows that there are many more instances that were left unsolved by the BB&R-DFS algorithm. Unlike the BB&R-DBFS algorithm, the BB&R-DFS algorithm encounters memory limitations starting at instances with $n = 140$. The BB&R-DFS algorithm consumes more memory and has a slower computational running time compared to the BB&R-DBFS algorithm.

In addition to evaluating the computational performances of the BB&R-DBFS algorithm, Table 4.5 provides the necessary data for the physical memory usage of the algorithm. Table 4.5 reports the maximum number of states stored for the set of test instances with $\alpha = 0.5$ and $\beta = 0.5$. This parameter setting is chosen for this evaluation because these test instances consume more memory than any other

Table 4.3: $1|r_i|\sum t_i$ BB&R-DBFS Algorithm: Average and Maximum CPU Time (sec.)

		$n=60$		$n=100$		$n=150$		$n=200$		$n=250$		$n=300$		$n=400$		$n=500$	
α	β	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max
0	0.05	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0	0.25	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.1
0	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.3
0.5	0.05	0.3	0.4	1.1	2	2	4	23	45	54	110	529	1242	2034(7)	-	772(1)	-
0.5	0.25	0.4	2	3	21	4	9	28	67	90(9)	-	532	1342	645(4)	-	981(2)	-
0.5	0.5	0.5	1	1(9)	-	9(9)	-	29	79	192(9)	-	269(7)	-	1540(1)	-	273(1)	-
1	0.05	0.2	0.4	0.5	0.6	12	53	6	11	18	47	37	82	403(9)	-	342(7)	-
1	0.25	0.2	0.8	2	14	0.6	2	9	36	12	83	0.1(9)	-	0.1(9)	-	0.1	0.1
1	0.5	0.1	0.2	0.1	0.4	0.1	0.1	1	13	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
1.5	0.05	0.1	0.2	0.2	0.3	0.2	0.4	2	3	5	10	5	11	21	52	27	65
1.5	0.25	0.1	0.2	0.1	0.5	0.1	.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
1.5	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Table 4.4: $1|r_i|\sum t_i$ BB&R-DFS Algorithm: Average and Maximum CPU Time (sec.)

		$n = 60$		$n = 100$		$n = 150$		$n = 200$		$n = 250$		$n = 300$		$n = 400$	
α	β	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max
0	0.05	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0	0.25	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2
0.5	0.05	0.4	1	4	9	28	80	196	607	602	1849	1340(3)	-	41(1)	-
0.5	0.25	0.7	3	3	20	192(9)	-	136(6)	-	191(3)	-	-	-	-	-
0.5	0.5	0.9	6	4(9)	-	459(8)	-	49(8)	104	149(5)	-	467(4)	-	-	-
1	0.05	0.2	0.6	1	4	5	17	116	601	266	1260	482(9)	1872	1817(1)	-
1	0.25	0.2	0.6	11	100	5	48	0.3	0.9	0.5	5	0.1(9)	0.1	0.1(9)	-
1	0.5	0.1	0.1	0.1	0.3	0.1	0.1	1	13	0.1	0.1	0.1	0.1	0.1	0.1
1.5	0.05	0.1	0.1	0.3	1.5	1	4	2	3	11	27	9	46	115(9)	993
1.5	0.25	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
1.5	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

instances prior to any memory limitations. Also, the test instances with $\alpha = 0.5$ and $\beta = 0.5$ seem to be the hardest test instances. Note that from Table 4.5, the largest instances that were solved without running into memory limitation are the $n = 250$ size test instances. However, the maximum number of stored states is from the $n = 100$ test instances. The maximum number of stored states for $n = 100$ and $n = 250$ is $277,977$ and $187,231$ states, respectively. For each state stored, the BB&R algorithm requires 11 integer types and a variable size bit vector. The largest bit vector used in our experiments is 63 bytes long. The total memory consumption for each stored state is 107 bytes, where 44 bytes are from the integer type and 63 bytes are from the bit vector. Therefore, for $n = 100$ and $n = 250$, the BB&R-DBFS algorithm uses approximately 30 MB and 20 MB of memory, respectively. Note that this is a slight overestimate since the bit vector is not always 63 bytes long. Furthermore, all computational experiments are limited to a maximum of 8 million states. Assuming that each states consumes 100 bytes, then the BB&R-DBFS algorithm approximately consumes at most 800 MB of memory. Therefore, from Tables 4.1, 4.2, and 4.5, the performance superiority of the BB&R algorithm compared to previous algorithms is not due to the additional memory, but rather, a result of the dominance rules, the DBFS exploration strategy, and the improved Decomp-DP lower bound algorithm.

To further provide a more complete evaluation of the BB&R-DBFS algorithm, it is also compared with the algorithm presented in Jouglet et al. [50], denoted as the JBC algorithm. Both the BB&R-DBFS and the JBC algorithm are executed on the same computing platform over the same test instances used in the Jouglet et al. [50]. Tables 4.6 and 4.7 report the average running time for the BB&R-DBFS and the JBC algorithm respectively. Note that the values that are in parenthesis denote the number of instances *solved*. The last column labeled "Largest n" reports the largest size instances where at least 80% of the test instances for that parameter setting are

Table 4.5: $1|r_i|\sum t_i$ BB&R-DBFS Algorithm: Maximum and Average Number of Stored States ($\alpha = 0.5, \beta = 0.5$)

n	Max.	Avg.
10	9	2.5
20	22	4.8
30	113	38.9
40	335	76.9
50	3726	479.8
60	2008	656.3
70	4031	569.3
80	1731	549
90	4441	1154.9
100	277977	28411
150	190869	26148
200	37658	14287.7
250	187231	89696

solved to optimality.

The performance of the BB&R-DBFS algorithm compares favorably to the JBC algorithm, both in terms of speed and the size of the largest problems that can be solved. Table 4.6 and 4.7 demonstrate that the average running times for the BB&R-DBFS algorithm are between two to four orders of magnitude faster for the harder parameter settings. For example, for $n = 60, \alpha = 0.5$, and $\beta = 0.25$, the average running time for their algorithm was 88 seconds while the average running time for BB&R-DBFS was 0.3 seconds. For $n = 60, \alpha = 0.5$, and $\beta = 0.5$, the average running time for the JBC algorithm was 1619 seconds while the average running time for BB&R-DBFS was 0.4 seconds. BB&R-DBFS solved all the instances with $\alpha = 0$ without branching since the lower bound based on the $1||\sum t_i$ problem was tight. The computational results in Table 4.6 for $\alpha = 0$ clearly show that the B&R algorithm used to solve the $1||\sum t_i$ problem provides a significant speedup.

In terms of the size of the largest problems that can be solved, for the hardest parameter settings with $\alpha = 0.5$ and $\beta = 0.5$, the JBC algorithm was unable to solve 80% of the instances with $n = 70$, whereas BB&R-DBFS was able to solve 80% of

Table 4.6: $1|r_i|\sum t_i$ BB&R-DBFS Algorithm Using Jouglet et al. [50] Test Instances in Average CPU Time (sec.)

α	β	$n = 60$	$n = 70$	$n = 80$	$n = 90$	$n = 100$	$n = 150$	$n = 200$	$n = 250$	$n = 300$	$n = 350$	Largest n
0	0.05	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	500
0	0.25	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	500
0	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.1	0.2	500
0.5	0.05	0.2	0.3	0.4	0.4	1.4	5.4	12	55	477	-	340
0.5	0.25	0.3	0.4	0.3	1.1	1.5	40(9)	18(9)	73(9)	238(5)	-	190
0.5	0.5	0.4	0.3	0.7	0.5	1.2(9)	13	35	94(9)	240(8)	-	290
1	0.05	0.1	0.2	0.3	0.5	0.5	1.6	5	10	43	252	400
1	0.25	0.2	0.2	0.2	0.6	0.7	20	2	19(7)	16	0.1(8)	240
1	0.5	0.1	0.2	0.2	0.2	11	0.1	0.1	0.1	0.1	0.1	500
1.5	0.05	0.1	0.1	0.2	0.2	0.2	0.7	1.3	5	6	20	500
1.5	0.25	0.1	0.1	0.1	0.1	0.1	0.3	0.1	22	0.1	0.1	500
1.5	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1(9)	500

Table 4.7: $1|r_i|\sum t_i$ JBC Algorithm Using Jouglet et al. [50] Test Instances in Average CPU Time (sec.)

α	β	$n = 60$	$n = 70$	$n = 80$	$n = 90$	$n = 100$	$n = 150$	$n = 200$	$n = 250$	$n = 300$	$n = 350$	Largest n
0	0.05	0.1	0.2	0.3	0.4	0.4	2.3	9.1	24	48	149	500
0	0.25	0.5	1	2.4	6	8	31	460	1365(7)	-	-	240
0	0.5	7	11	27	52	111	647(9)	-	-	-	-	180
0.5	0.05	2.6	7	20	59	202	2771(7)	-	-	-	-	150
0.5	0.25	88	590	1725(7)	-	-	-	-	-	-	-	70
0.5	0.5	1619(8)	3387(5)	-	-	-	-	-	-	-	-	60
1	0.05	1.2	6	28	25	123	-	-	-	-	-	150
1	0.25	1.6	4	8	7	0.6	15(9)	7	292	32(8)	0.1(7)	340
1	0.5	0.01	0.04	0.03	0.2	1	54	0.1	.1(9)	0.1	0.1	500
1.5	0.05	0.3	0.4	1.8	2.4	6	13	174	434	150(9)	634	500
1.5	0.25	0.03	0.03	0.05	0.04	0.1	0.3	1.2	4	14	0.1	500
1.5	0.5	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	66	500

the instances with up to $n = 290$ jobs. For six different combinations of α and β, BB&R-DBFS was able to solve problems that were at least twice as large as those solved with the JBC algorithm. However, it is important to note that for $\alpha = 1$ and $\beta = 0.25$ with $n = 250$, the BB&R-DBFS algorithm did not perform as well the JBC algorithm. The JBC algorithm was able to solve all 10 instances while the BB&R-DBFS algorithm was only able to solve 7 instances. It is also worth noting that the BB&R-DBFS algorithm performed much better then the JBC algorithm for the same parameter settings for $n > 250$. Note that $n = 500$ is the largest size instance in the test set, though the computational results indicate that for BB&R-DBFS can solve much larger instances for some of the combinations of α and β parameter settings.

4.6 Conclusion

This chapter presents the BB&R algorithm using the DBFS exploration strategy to solve the $1|r_i|\sum t_i$ scheduling problem. Several previously known dominance rules are incorporated into the BB&R algorithm. This chapter provides a proof showing that the combination of dominance rules used in the BB&R algorithm remains exact. In addition, this chapter provides a memory-based enhancement to the Lawler [65] dynamic programming algorithm, which improves the computational performance for computing the lower bound. Furthermore, a new decomposition approach used with the Lawler [65] dynamic programming algorithm provides tighter lower bounds. The computational results of this chapter show that the combination of all the dominance rules, DBFS exploration strategy and improved bound computation results in a highly efficient BB&R-DBFS algorithm. The BB&R-DBFS algorithm outperforms the current best known algorithms for the hardest test instances, and performs equally well for the easier test instances. The new DBFS exploration strategy provides a significant computational speedup compared to the more traditional DFS

exploration strategy. Incorporating the new DBFS exploration strategy also allows the BB&R-DBFS algorithm to solve larger instances.

Chapter 5

The $1|ST_{sd}|\sum t_i$ Scheduling Problem

Over the past fifty years, there has been a growing research interest in scheduling problems, however, the majority of the literature, assumes that setup times are negligible. In practice, assumptions with sequence *independent* setup time are inadequate in modeling real-world problems [85, 99].

This chapter considers minimizing total tardiness on a single machine scheduling problem with sequence dependent setup times. This problem, denoted as $1|ST_{sd}|\sum t_i$ [44, 1], has several variations including minimizing total setup time, minimizing makespan, and minimizing the maximum tardiness, among others. See Allahverdi et al. [1] for a comprehensive survey of various scheduling problems with setup times. Note that when the objective is to minimize total setup time, the problem is equivalent to the classic traveling salesman problem that is NP-hard. One well studied variation of the $1|ST_{sd}|\sum t_i$ is the $1||\sum t_i$ scheduling problem, where the setup time is ignored. The $1||\sum t_i$ scheduling problem is also NP-hard [31]. Several exact algorithms, including dynamic programming and branch and bound (B&B) algorithms, have been proposed by Lawler [65], Potts and Van Wasenhove [88], Szwarc et al. [96], and Chang et al. [14].

While there are numerous exact algorithms for solving the $1||\sum t_i$ scheduling problem, there are few exact algorithms in the literature for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. Most of the literature propose the use of meta-heuristics such as simulated annealing [98], genetic algorithms [2, 90, 99], tabu search [69], and ant colony algorithms [36]. Gupta and Smith [45] also propose the greedy randomized

adaptive search procedure (GRASP) and a local search heuristic for the problem. Constructive heuristics and improvement heuristics have also been developed, though the solution quality with such heuristics is poor and requires intensive computational time [67]. Tan et al. [99] compare the performances of various meta-heuristics for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. Tan et al. [99] also report that B&B algorithm seems to be the most effective for solving smaller size problems (less than 15 jobs), while simulated annealing and random-start local search have better performances for larger size problems. Lin and Ying [69] also compare various meta-heuristics for the weighted total tardiness problem, $1|ST_{sd}|\sum w_i t_i$.

The most common exact algorithm for solving the $1|ST_{sd}|\sum t_i$ scheduling problem is B&B algorithms. Ragatz [89] proposes a B&B algorithm for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. An algorithm for computing a lower bound and some dominance properties are also presented in Ragatz [89]. Other variations of B&B algorithms have also been proposed by Souissi and Chu [95], Luo and Chu [72, 70], and Luo et al. [71]. The differences among these proposed B&B algorithms include variations of the dominance rules, bounding schemes, and the exploration strategies used. Luo and Chu [72] report the best computation results, claiming to solve instances with up to size 30 jobs.

The Branch, Bound, and Remember (BB&R) framwork presented in Chapters 3 and 4, with the Best First Search (BFS) exploration strategy is used in this chapter to solve the $1|ST_{sd}|\sum t_i$ scheduling problem. A new memory based dominance rule is incorporated into the BB&R algorithm for pruning dominated sub-problems. In addition, the Branch and Remember (B&R) algorithm presented in Chapter 4 for solving the $1||\sum t_i$ is used to compute tighter lower bounds [65].

This chapter is organized as follows. Section 5.1 describes the $1|ST_{sd}|\sum t_i$ scheduling problem and the necessary notation used in this chapter. Section 5.2 outlines the BB&R algorithm, including the new memory based dominance rule and the B&R

algorithm for computing the lower bound. Section 5.3 provides a counterexample to the B&B algorithm described in Luo and Chu [72] and Lu et al. [71]. Computational results are reported in Section 5.4, followed by concluding comments in Section 5.5.

5.1 Notations

The $1|ST_{sd}|\sum t_i$ single machine scheduling problem consist of a set of jobs $J = \{1, 2, \ldots, n\}$ to be scheduled in sequence, where each job has a processing time p_i, a due date d_i, and a vector of setup times $S_i = (s_{0,i}, s_{1,i}, \ldots, s_{n,i})$, where $s_{i,j}$ is the setup time for job j if it is scheduled immediately after job i. The setup time $s_{0,i}$ represents the setup time incurred for scheduling job i as the first job in the scheduled sequence. All jobs are available for processing at time zero. Processing a job incurs a sequence dependent setup time and a processing time.

The following notations and assumptions are used in the remainder of the chapter. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ be a partial sequence of scheduled jobs, where $\sigma_i \in J$ for $i = 1, 2, \ldots, m$ and $m \leq n$. Let

- $c_{\sigma_i} = \sum_{j=1}^{i} s_{\sigma_{j-1},j} + p_{\sigma_j}$, the completion time of job σ_i in job sequence σ,
- $C_\sigma = c_{\sigma_m}$, the completion time of job sequence σ,
- $t_{\sigma_i} = max(0, c_{\sigma_i} - d_{\sigma_i})$, the tardiness of job σ_i in job sequence σ,
- $T_\sigma = \sum_{i=1}^{m} t_{\sigma_i}$, the total tardiness for job sequence σ,
- $T_\sigma(t)$, the total tardiness of job sequence σ if its starting time is at time t with no initial setup time,
- F_σ, the set of unscheduled (i.e., free) jobs.

The objective of the $1|ST_{sd}|\sum t_i$ scheduling problem is to find a sequence of scheduled jobs with minimum total tardiness, $\sum_{i=1}^{n} t_i$.

5.2 Branch, Bound, and Remember Algorithm

This section introduces the BB&R algorithm for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. Section 5.2.1 describes the memory-based dominance rule used for reducing the solution space. Section 5.2.2 provides an overview of the bounding scheme used to compute the lower bound, including a B&R algorithm first introduced in Kao et al. [55]. A description of the BB&R algorithm is provided in Section 5.2.3.

5.2.1 Dominance Rule

The memory-based dominance rule, called the Memory Dominance Rule (MDR) for the $1|ST_{sd}|\sum t_i$ scheduling problem, compares two partial sequences to determine if one dominates the other. The MDR determines which partial sequence provides the guarantee that would lead to a solution that is better or at least as good as other solutions generated from the other partial sequence. The MDR is memory-based since it requires the BB&R algorithm to store all partial sequences that have been previously explored for comparison. The following definition defines the MDR used in the BB&R algorithm.

Definition 5.2.1 *Let* $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ *and* $\delta = (\delta_1, \delta_2, \ldots, \delta_m)$ *be partial sequences of jobs. Then* σ *dominates* δ *if* $(F_\delta = F_\sigma) \wedge (c_\sigma \leq c_\delta) \wedge (t_\sigma \leq t_\delta) \wedge (\sigma_m = \delta_m)$.

The MDR suggests that dominant partial sequence can result in a solution with equal or less total tardiness than any other solution generated from the dominated partial sequence. Theorem 12 shows that the MDR will not prune a superior solution.

Theorem 12 *Let* $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_i)$ *and* $\beta = (\beta_1, \beta_2, \ldots, \beta_i)$ *be two partial sequences of scheduled jobs. Let* $\beta^* = (\beta_1, \beta_2, \ldots, \beta_i, \beta_{i+1}, \ldots, \beta_n)$ *be a full sequence of scheduled jobs with the least total tardiness that is generated by* β. *If* α *dominates* β,

then there exist a full sequence of scheduled job α^* generated by the partial sequence of scheduled jobs α such that $T_{\alpha^*} \leq T_{\beta^*}$.

Proof: If α dominates β, then we can construct a sequence $\alpha^* = (\alpha_1, \alpha_2, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_n)$. Let the subsequence $\beta' = (\beta_{i+1}, \beta_{i+2}, \ldots \beta_n)$. The full sequence of scheduled jobs α^* is a feasible job schedule because $F_\alpha = F_\beta$. Since $\alpha_i = \beta_i$, then the setup time $s_{\alpha_i,\beta_{i+1}} = s_{\beta_i,\beta_{i+1}}$. Also, since $C_\alpha \leq C_\beta$ and $T_\alpha \leq T_\beta$, then $T_{\alpha^*} = T_\alpha + T_{\beta'}(C_\alpha + s_{\alpha_i,\beta_{i+1}}) \leq T_\beta + T_{\beta'}(C_\beta + s_{\beta_i,\beta_{i+1}}) = T_{\beta^*}$. Therefore, there exist a job sequence that is at least as good as β^*. □

5.2.2 Bounding Scheme

This section provides an overview of the bounding scheme used in the BB&R algorithm for the $1|ST_{sd}|\sum t_i$ scheduling problem. The quality of the upper and lower bounds can lead to significant improvements in the performance of the overall algorithm. A local search method is used to generate the initial upper bound. Two lower bound algorithms are then used to compute the lower bound at each branch.

The bounding scheme works as follows. Prior to any branching, a local search method is used to generate an initial solution as the upper bound. The branching process proceeds by scheduling additional jobs to the partial sequences. At each branch, two lower bound algorithms are used to compute a lower bound based on the remaining free jobs. The maximum of the two computed lower bounds is kept. If the lower and upper bounds are equal (i.e., tight), then the branch is pruned.

The initial local search method for generating the upper bound uses a 2-exchange neighborhood. A 2-exchange neighborhood generates a new solution by randomly swapping two jobs in the scheduled sequence. If the new solutions generated have less total tardiness, the new solution is accepted. One thousand 2-exchange iterations are executed to generate the initial upper bound.

Two lower bound algorithms are used at each branch to compute the lower bound. The first lower bound algorithm, denoted as the RLB algorithm, was originally proposed by Ragatz [89]. Given a partial sequence σ, the RLB algorithm combines the tardiness of the scheduled jobs, T_σ, with a lower bound based on the remaining unscheduled jobs, F_σ. The RLB algorithm computes a lower bound by adjusting the processing time and the due dates of the jobs in F_σ. The processing time are adjusted to include the minimum setup time. Jobs are then scheduled on a shortest operation time order. The corresponding due dates for each jobs are also re-ordered to an earliest due date order. Tardiness is then computed based on these adjusted processing times and due dates [89, 99].

The second lower bound algorithm used for computing the lower bound is a B&R algorithm presented in Chapter 4. This lower bound algorithm, denoted as the KSJLB algorithm in this chapter, is a B&R algorithm that uses the decomposition rules based on Lawler's dynamic program for solving the $1||\sum t_i$ scheduling problem [65]. The KSJLB also incorporates the improved decomposition methods proposed by Change et al. [14]. To provide a further speedup to the original dynamic program proposed by Lawler [65], the KSJLB algorithm builds a database of states as the lower bounds are computed. Since many sub-problems for which the lower bounds must be computed share the same states, the KSJLB algorithm avoids re-solving these sub-problems by storing each state, which efficiently reduces the computational effort required to compute the lower bound.

In order to apply the KSJLB algorithm, the $1|ST_{sd}|\sum t_i$ problem is relaxed by adjusting each job's processing time to include the minimum setup time and ignoring the setup time in the schedule. That is for each job j, the adjusted processing time is $p'_j = p_j + min\{s_{i,j}|i = 1, \ldots, j-1, j+1, \ldots, n\}$, which is then used to compute the lower bound by the KSJLB algorithm. Note that if all setup times are equal, the KSJLB algorithm finds the optimal solution.

5.2.3 The Algorithm

The BB&R algorithm presented in this chapter uses a Best First Search (BFS) exploration strategy. Various B&B exploration strategies have been proposed in the literature. Souissi and Chu [95] propose four different exploration strategies for their B&B algorithm, which are variations of Depth First Search (DFS) and BFS. Ragatz [89] also proposes a different exploration strategy where the search consists of switching between DFS and BFS in the branching process. Note that the BB&R algorithms presented in Chapters 3 and 4 incorporate the Distributed Best First Search (DBFS) exploration strategy (see Section 3.3.2). Chapters 3 and 4 show that the BB&R algorithm with the DBFS exploration strategy outperforms the best known algorithms in the literature for both the $1|r_i|\sum Ui$ and the $1|r_i|\sum t_i$ scheduling problems.

The BB&R algorithm is a constructive B&B algorithm. Solutions are generated by sequentially appending unscheduled jobs until a complete schedule is found. Each node in the B&B search tree is denoted by a three-tuple $(\sigma, F_\sigma, T_\sigma)$, where σ is a partial sequence of scheduled jobs, F_σ is the set of unscheduled jobs, and T_σ is the total tardiness for the partial sequence σ. Branching in the BB&R algorithm consists of exploring a node by appending an unscheduled job to σ. Lower bounds are computed at each node based on T_σ and F_σ. The maximum lower bound obtained by the RLB and the KSJLB algorithm is kept as the lower bound at that node. Each visited node is then stored in a hash table, and hence, *remembered*. By storing each node, the MDR can then be applied for pruning dominated branches. Additional branches are also pruned if the lower and upper bounds are tight.

The BB&R algorithm is now formally outlined by the following steps:

Step 1: Generate the upper bound, *ub*, by the 2-exchanged neighborhood local search.

Step 2: Compute the lower bound, *lb*, by taking the maximum of the two lower bounds computed by the RLB and the KSJLB algorithms.

Step 3: If $lb = ub$, then the optimal solution is found and the algorithm stops. Otherwise go to the next step.

Step 4: Generate a root node, $\sigma = ()$, $F_\sigma = J$, and $T_\sigma = 0$.

Step 5: Insert the root node into a heap.

Step 6: If the heap is not empty then go to the next step. Otherwise, the optimal solution is found and the algorithm stops.

Step 7: Obtain a current node, $(\sigma', F'_\sigma, T'_\sigma)$ by removing the top node from the heap.

Step 8: If F'_σ is empty, then update the upper bound ub.

Step 9: For each free job $j \in F'_\sigma$, create a new job sequence σ'' by appending j to σ'.

Step 10: For each new job sequence σ'', compute a lower bound lb' using the RLB and the KSJLB algorithms.

Step 11: If $lb' \geq ub$ then prune the current node by going to Step 6. Otherwise, go to the next step.

Step 12: For each new job sequence, generate a new node $(\sigma'', F''_\sigma, T''_\sigma)$.

Step 13: Search the hash table and apply MDR to the new node for pruning.

Step 14: If the new node does not violate MDR, then add the new node to the hash table and the heap.

Step 15: Go to Step 6.

The heap structure described in Step 5 is sorted by the current lower bound of the sub-problems. Nodes with partial sequences that have the best lower bound are removed from the heap and explored earlier in the search process then nodes with partial sequences that have a worst lower bound. Each node is only stored in the

hash table if it is not pruned by either the bounds or the MDR. Note that the hash table is indexed by bit vectors, which denote the set of unscheduled free jobs. Using a bit vector to represent the set of free jobs allow a fast look-up time for the MDR to compare previously explored nodes with the current node.

5.3 Counterexample

This section presents a counter example to a dominance rule used in the B&B algorithm presented in Luo and Chu [72] and Luo et al. [71]. For clarification, the notations and the theorem presented in Luo and Chu [72] are provided. The following additional notations will be used in this section:

- $J(K)$, the set of jobs in the partial job schedule K.
- $U(K)$, the set of unscheduled jobs.
- $C(K)$, the completion time of the last job in K.
- $K|u$, the new partial job schedule obtained by appending job u to the partial job schedule K.
- $\sum(K|u)$, the job schedule composed of $K|u$, completed by the partial optimal job schedule, which belong to $J - J(K|u)$, starting from $C(K|u)$.
- $|\cdot|_{[i][j]}$, the number of jobs from position i to position j, including the two jobs at position i and j.
- S be a sequence of jobs, and S', a sequence of jobs after some jobs are interchanged in sequence S. Then $[i]$ refers to the job index of the ith position in S.

The dominance rule presented in Luo and Chu [72] and Luo et al. [71] is formally stated in the following theorem.

Theorem 13 [71, 72]

If there exists i, where $[i] \in J(K)$, $\Delta_1 = s_{[i-1][k]} + p_{[k]} + s_{[k][i+1]} - s_{[i-1][i]} - p_{[i]} - s_{[i][i+1]} \leq 0$, $\|\cdot\|_{[i][k]} = n_1$, $\|\cdot\|_{u,end} = n_2$, $\Delta_2 = s_{[i-1][k]} + s_{[k][i+1]} + s_{[k-1][i]} + s_{[i]u} - s_{[i-1][i]} - s_{[i][i+1]} - s_{[k-1][k]} - s_{[k]u} \leq 0$, $\Delta_T = (n_1 - 1)\Delta_1 + s_{[i-1][k]} + s_{[k-1][i]} - s_{[i-1][i]} - s_{[k-1][k]} + n_2\Delta_2 \leq 0$, *then* $\sum(K|u)$ *is dominated.*

The counterexample for Theorem 13 is a 7 jobs instance with processing time $\{290, 95, 100, 102, 197, 106, 103\}$ and due dates $\{783, 683, 824, 708, 808, 700, 784\}$. Define sequence $\alpha = (1, 2, 3, 4, 5, 6, 7)$. Then a new sequence $\beta = (1, 2, 6, 4, 5, 3, 7)$ can be constructed by interchanging job 3 and 6. Let the relevant setup times for these sequences be $s_{0,1} = 0$, $s_{1,2} = 7$, $s_{2,3} = 7$, $s_{3,4} = 7$, $s_{4,5} = 8$, $s_{5,6} = 9$, $s_{6,7} = 9$, $s_{2,6} = 8$, $s_{6,4} = 7$, $s_{5,3} = 7$, $s_{3,7} = 9$. By Theorem 13, $\Delta_1 = -5$, $\Delta_2 = -1$, and $\Delta_T = -17$, and hence, β is a dominant sequence. However, the total tardiness is $T_\alpha = 371$ and $T_\beta = 488$, hence, Theorem 13 pruned a superior sequence.

Luo and Chu [72] proves Theorem 13 by dividing the scheduled sequence of jobs into three parts, namely *part 0*, *part 1*, and *part 2*, where part 0 corresponds to the portion of the sequence prior to the index where where jobs $[i]$ and $[k]$ are interchanged, part 1 corresponds to the portion of the sequence of jobs between the two jobs $[i]$ and $[k]$, including the jobs $[i]$ and $[k]$, and part 3 corresponds to the remaining part of the sequence after the interchange of the two jobs $[i]$ and $[k]$. The breakdown in the proof presented by Luo and Chu [72] is on how the total tardiness is computed in part 1 of the sequence. Their formula using Δ_1 for computing the differences in tardiness in part 1 of the sequence is incorrect, since it does not take into account that negative tardiness does not exist. The tardiness of a job is either 0 or a positive value equal to the completion time minus the due date. The negative tardiness that

is factored into their dominance rule over compensates for the difference in tardiness, Δ_1, after the new sequence is constructed.

This counterexample shows that the B&B algorithm presented in Luo and Chu [72] and Luo et al. [71] may over prune, and hence, may not have solved all their test instances to optimality. In addition to losing the exactness of their algorithm, over pruning can reduce the computation effort of the overall computational performances of their B&B algorithm.

5.4 Computational Results

This section reports computational results for the BB&R algorithm described in Section 5.2. The computational results for the BFS exploration strategy is compared with the computational results of the DBFS and DFS exploration strategies. In addition, this section also compares the effectiveness of the two lower bound algorithms, the RLB and the KSJLB, described in Section 5.2.2. The overall performance of the BB&R algorithm is also compared to the computational results reported in Luo and Chu [72].

The BB&R algorithm is evaluated over 2,880 randomly generated test instances. These test instances were generated using the same generation scheme described in Ragatz [89], Luo and Chu [72], Luo et al. [71], and Luo et al. [70]. Five different parameters are used to generate the test instances:

- N, the number of jobs,
- VP, the variance of the job processing time,
- RS, the range of the setup time,
- TF, the average tardiness factor,

- RD, the relative range of the due dates.

The variance of the job processing time, VP, is used to generate the processing times for each jobs. Let the mean of the processing times be denoted as MP. MP is then used along with the TF and RD, to generate the due dates. The mean of the due date distribution is set equal to $\mu = (1 - TF)(N)(MP)$, and the due dates are then generated uniformly over $(\mu-((RD)(N)(MP))/2, \mu+((RD)(N)(MP))/2)$. The setup times are also generated uniformly over $(9.5-(RS/2), 9.5+(RS/2))$. The parameters used to generating the test instances are $N = \{10, 12, 14, 16, 18{,}20, 22, 26, 30\}$, $VP = \{25, 625\}$, $RS = \{5, 19\}$, $TF = \{0.2, 0.4, 0.6, 0.8\}$, and $RD = \{0.2, 0.9\}$. For each combination of parameters settings, ten random instances are generated, for a total of 2,880 instances. All the experiments were executed on a 2.4 Ghz Pentium PC with 2GB of RAM, with each instance in the test set restricted to total processing time of 30 CPU minutes, and total memory usage of 2GB.

Three different exploration strategies for the BB&R algorithm are compared. Table 5.1 reports the average and maximum running time (in CPU seconds) for the DFS, BFS and DBFS exploration strategies. The DFS exploration strategy had the worst computational performance compared to the other two exploration strategies. Also, as shown by the maximum running time, the computational performance of the DFS exploration strategy degraded significantly as the size of the instances increased. On average, the DBFS and BFS exploration strategies was two to three times faster than the DFS exploration strategy. The DBFS exploration strategy results were comparable to the BFS exploration strategy. On average the overall computational performance of the BFS exploration strategy was doing slightly better than the DBFS exploration strategy. It appears that as the size of the problem instances increases, the BFS exploration strategy becomes more efficient relative to the other exploration strategies compared. Note that the average running times reported in Table 5.1 do not include those problem instances that are unsolved due to the time limitation or

memory limitation.

Table 5.2 reports the fraction of problem instances solved with respect to the computational time limits for the DFS, DBFS and BFS exploration strategies. The data reported in Table 5.2 also includes the instances that were unsolved due to memory limitations. While all instances of size $N = 20$ are solved to optimality within a 15 CPU minute time limit, the DFS exploration strategy could not solve 14 of the 320 problem instances because of memory limitations and 3 of the 320 problem instances because of time limitations. For the $N = 22$ instances, the DFS exploration strategy failed to solve 39 problem instances due to memory limitations and 12 problem instances due to time limitations out of the 320 problem instances. The results show that the BB&R algorithm is more susceptible to the memory limitation constraint than the time limitations constraint. The DBFS exploration strategy failed to solve 22 of the 320 problem instances due to memory limitations for problem instances with $N = 22$, and the BFS exploration strategy failed to solve 21 of the 320 problem instances due to memory limitations for problem instances with $N = 22$. All problem instances of $N = 22$ that were not solved to optimality by the DBFS and BFS exploration strategy were cause by memory limitations. The affects of the memory limitation is primarily caused by the database of states needed for both the lower bound computation and for the MDR. These experiments display the classic tradeoff between additional memory usages and reduction in computational times.

The computational experiments also show that the BB&R algorithms had the most difficulty with problem instances with relative due date range $RD = 0.2$. Nearly *all* instances there were unsolved either because of memory or time limitations had a relative due date range $RD = 0.2$. A small RD value correspond to having jobs with closer due dates. This in turn can generate many solutions that have similar objective function values. The impact of a narrow due dates range RD is also reported in Ragatz (1993).

Table 5.1: $1|ST_{sd}|\sum t_i$ BB&R Algorithms: Average and Maximum CPU Time (sec.)

	DFS		DBFS		BFS	
N	Avg.	Max	Avg.	Max	Avg.	Max
10	0.04	0.27	0.02	0.11	0.02	0.13
12	0.23	3.5	0.09	1.08	0.09	0.98
14	1.3	17.2	0.4	3.7	0.38	3.4
16	6.4	136.8	1.8	26.3	1.7	24.9
18	46.5	707.5	9.4	162.1	8.9	148.5
20	105.4	1800	35.6	569	32.4	512.3
22	164.6	1800	85.3	1760	77.7	1599

Table 5.2: $1|ST_{sd}|\sum t_i$ BB&R Algorithms: Fraction Solved By Time Limit with Different Exploration Strategies

	DFS			DBFS			BFS		
N	225s	15m	30m	225s	15m	30m	225s	15m	30m.
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
14	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
16	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
18	0.92	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
20	0.82	0.92	0.94	0.95	1.0	1.0	0.95	1.0	1.0
22	0.68	0.79	0.84	0.81	0.91	0.93	0.83	0.91	0.93

Table 5.3: Performance of the Lower Bound Algorithms for the $1|ST_{sd}|\sum t_i$ Scheduling Problem

	RLB		KSJLB	
N	Avg %	Avg Gap	Avg %	Avg Gap
10	1.3	84.6	87.1	9.5
12	0.9	87.3	90.8	10.5
14	0.3	86.1	89.6	9.9
16	0.2	85.8	88.8	9.6
18	0.2	84.6	88.5	8.7
20	0.1	85.1	88.2	8.3
22	0.04	84	86.5	8.8
26	-	-	-	-
30	-	-	-	-

Table 5.3 also reports the data comparing the effectiveness between the RLB and KSJLB algorithms for computing the lower bounds. The column labeled Avg % reports the percentage of the bounds computed where the corresponding algorithm provided a tighter lower bound. The column labeled Avg Gap reports the average gap between the initial lower bound computed by the respective algorithm with the optimal objective function value found. These results show that the RLB algorithm provides a weaker initial lower bound compared to the KSJLB. On average, the initial lower bounds computed by KSJLB were almost always within a 10% gap from the optimal solution. The tighter initial gap from the computed lower bound by the KSJLB algorithm provided significantly more pruning. Furthermore, most of the pruning that was attributed to the bounds was done by the KSJLB algorithm. However, it was observed that the RLB algorithm became much more effective towards the end of the B&B search processes. The lower bounds computed by the RLB algorithm were able to prune more branches when most jobs were already scheduled relative to when fewer jobs were scheduled, as in the early stages of the B&B search process. The lower bounds computed by the KSJLB algorithm otherwise provided more consistent pruning throughout the B&B search process.

The performance of the BB&R algorithm with the BFS exploration strategy com-

Table 5.4: $1|ST_{sd}|\sum t_i$ BB&R Algorithms Comparison with Luo and Chu [72]

	Luo and Chu (2006)		BB&R with BFS	
N	CPU Sec.	Solved (%)	CPU Sec.	Solved (%)
10	0.405	100	0.02	100
14	0.988	93.75	0.38	100
18	13.176	80.10	8.9	100
22	43.998	68.23	77.7	83.4
26	60.882	56.17	-	-
30	100.83	47.5	-	-

pares favorably to the computational results reported in Luo and Chu [72], both in terms of speed and the percentage of the largest problems solved. Table 5.4 reports the computational time of the BB&R algorithm with the BFS exploration strategy and the computational time of Luo and Chu's algorithm. Table 5.4 also reports the percentage of problem instances solved to optimality with a 900 CPU seconds and a 225 CPU seconds time limit for Luo and Chu's algorithm and the BB&R algorithm, respectively. Note that the experiments reported in Luo and Chu [72] were computed on an Intel Pentium II 600 Mhz processor machine, whereas the experiments in this chapter were executed on a Pentium D 2.4 Ghz processor machine. To adjust for this difference in computing platform, the time limitation is reduced to a quarter of the time limitation used in Luo and Chu [72]. With the adjusted time limitation, the BB&R algorithms with BFS and DBFS exploration strategies were able to solve more problem instances. Table 5.4 shows that with the smaller problem instances, the the BB&R algorithm with BFS exploration stratiegy can be an order of magnitude faster, however, with larger problem instances, Luo and Chu's reported a faster running time. Although the reported running time of Luo and Chu's algorithm were faster for the larger instances, these averages were computed with only instances that were solved to optimality. The experiments in this chapter show that for larger instances, there were more unsolved instances by Luo and Chu's algorithm as compared to the BB&R algorithm with BFS exploration strategy. These unsolved instances can

significantly increase the reported average running time of Luo and Chu's algorithm. Also note that the counter example described in Section 5.3 shows that the results reported in Luo and Chu may be sub-optimal.

5.5 Conclusion

This chapter presents the BB&R algorithm with the BFS exploration strategy for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. A memory-based dominance rule is incorporated into the BB&R algorithm. A proof is also provided showing that the dominance rule will not over prune. A B&R algorithm for solving the $1||\sum t_i$ scheduling problem was also used for computing the lower bound for the $1|ST_{sd}|\sum t_i$ scheduling problem. The computational results reported show that the BB&B algorithm with the BFS exploration strategy is competitive, if not superior, to the best results reported in the literature. Furthermore, the computational results also show that the B&R algorithm for computing the lower bound is very efficient, consistently computing initial lower bounds with an average gap of less then 10%. Different exploration strategies for the BB&R algorithm were also compared. The DBFS and BFS exploration strategy provides a significant speed up over a traditional DFS exploration strategy. The BFS exploration strategy is shown to be slightly better then the DBFS exploration strategy.

Chapter 6

Post Optimality Selection

The previous three chapters focuses on single objective combinatorial optimization problems, however, many real-world optimization problems involve multiple (and often conflicting) objectives. These problems are relevant in a variety of engineering disciplines, scientific fields, and various industrial applications [20, 33]. Unlike single objective optimization problems, where one attempts to find the best solution (global optimum), in multi-objective optimization problems, there may not exist one solution that correspond to the best with respect to all objectives. Solving a multi-objective optimization problems consist of generating the Pareto frontier, the set of non-dominated solutions that represents the trade-off among the objective function values. Different approaches are used to approximate and generate such sets of Pareto optimal solutions. Some interactive approach incorporates preferences into the optimization procedure to explore a specific region of the solution space. While other approaches focus on generating a diverse sets of Pareto optimal solutions. Such sets of Pareto optimal solutions can be extremely large, which motivates the need for post-optimality analysis for multi-objective optimization problems.

The area of post-optimality analysis addressed in this chapter focuses on obtaining a preferred subset of solutions from a very large set of solutions with acceptable objective function values. The goal in obtaining large sets of Pareto optimal solutions is to provide the decision-maker with a diverse set of such solutions. Although obtaining diverse Pareto optimal solutions is important, it is often impractical for a human decision-maker to manually examine each such solution, and hence, efficiently

identify a good subset of such solutions. Previous research in this area has focused on generalizing the representation of the full set of Pareto optimal solutions with a smaller subset [57, 73]. Such procedures are not post-optimality analysis procedures, but rather, extensions to multi-objective optimization procedures, which are designed to generate diverse sets of Pareto optimal solutions [75, 74, 57]. Another area of research that incorporates preferences into the optimization procedures are interactive methods [77, 78]. These interactive methods provide a decision-makes with better control over the optimization process, allowing them to explore specific regions of the search space. However, solutions obtained are quite sensitive towards the preferences of the decision-maker. These approaches also require the decision-maker to have a thorough knowledge of the problem. Korhonen and Halme [61] suggest the use of a value function in helping decision-makers to identify the most preferred solutions. Alternatively, to objectively evaluate and distinguish good subsets of Pareto optimal solutions, Das [24] proposes an ordering and degree of efficiency among Pareto optimal solutions, which provides a way to measure and prune out less desirable Pareto optimal solutions.

This chapter analyzes a discrete optimization problem formulation for obtaining a preferred subset of Pareto optimal solutions from a larger set. This formulation alleviates the sensitivity of value function approaches, while obtaining a esired size subset of Pareto optimal solutions. Two exact algorithms are presented for solving the discrete optimization problem. In addition, five heuristics that obtain near-optimal solutions are introduced. The complexity of the discrete optimization problem formulation is presented. The exact algorithms and heuristics are applied to five test problems of various sizes, to provide comparisons of their computational performances.

The chapter is organized as follows. Section 6.1 formally introduces the discrete optimization problem formulation and necessary terminology and notation used in this chapter. Section 6.2 discusses its complexity. Section 6.3 outlines the exact

algorithms and other heuristics used to solve the discrete optimization problem. Section 6.4 reports computational results of the heuristics and algorithm, including the Greedy Reduction (GR) algorithm [103], applied to five test problems. Section 6.5 contains concluding comments and directions for future research.

6.1 Discrete Optimization Problem Formulation

This section formally presents the discrete optimization problem formulation that was previously introduced in Venkat et al. (2004). A brief review of the definitions and terminology used in Venkat et al. (2004) is provided.

Consider the multi-objective optimization problem:

$$\begin{aligned} &\min \mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_k(\mathbf{x})) = \mathbf{z} = (z_1, z_2, \ldots, z_k) \\ &\text{subject to: } \mathbf{x} \in S \end{aligned} \tag{6.1.1}$$

with k (≥ 2) objective functions $f_i : \Re^n \rightarrow \Re$, $i = 1, 2, \ldots, k$, where the decision variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ belong to the feasible region $S \subseteq \Re^n$.

Definition 6.1.1 *A solution $\mathbf{x}^* \in S$ and its objective function vector $\mathbf{z}^* = \mathbf{F}((x^*) \in \mathcal{F}^k$ is* Pareto optimal *if there does not exist another solution $\mathbf{x} \in S$ such that $f_i(\mathbf{x}) \leq f_i(\mathbf{x}^*)$ for all $i = 1, 2, \ldots, k$ with $f_j(\mathbf{x}) < f_j(\mathbf{x}^*)$ for at least one $j \in \{1, 2, \ldots, k\}$.*

Let $S^{PO} = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\} \subseteq S$ denote a set of Pareto optimal solutions, which may not contain the complete set of all Pareto optimal solutions. Let $\mathcal{F}^k = \{(z_1, z_2, \ldots, z_k) : \mathbf{z} = \mathbf{F}(\mathbf{x})$ for $\mathbf{x} \in S\}$ denote the feasible k-objective space.

A *value function* $V : \mathcal{F}^k \rightarrow \Re$, represents the preferences of a decision-maker across the objective functions. It provides a total ordering for the set of Pareto optimal solutions. In general, the value function is assumed to be strongly decreasing in its components (i.e., the preference of the decision-maker increases if the objec-

tive function value decreases, given that the other objective function values remain unchanged).

Definition 6.1.2 *A* percentile vector *of a solution* $\mathbf{x}^j \in S^{PO}, j = 1, 2, \ldots, N$, *is the corresponding vector of percentile values* $\mathbf{p}^j = (p_1^j, p_2^j, ..., p_k^j), j = 1, 2, \ldots, N$, *where* $p_i^j \in (0, 1]$ *is a percentile ranking of the* j^{th} *solution component based on the* i^{th} *objective function value.*

By definition, given a set of Pareto optimal solutions, S^{PO}, for every $\mathbf{x}^j \in S^{PO}, j = 1, 2, \ldots, N$, there exists a unique percentile vector $\mathbf{p}^j$. Therefore, there is a one-to-one mapping for all $\mathbf{x} \in S^{PO}$ to some $\mathbf{p}^j \in \wp^k$, termed the *percentile set* defined below.

Definition 6.1.3 *The* percentile space $(0, 1]^k$ *contain the* percentile set $\wp^k = \{\mathbf{p}^1, \mathbf{p}^2, \ldots, \mathbf{p}^N\} \subset (0, 1]^k$, *where each percentile vector* $\mathbf{p}^j$ *is defined as the percentile values corresponding to solution* $\mathbf{x}^j \in S^{PO}$.

A *percentile function* $q : (0, 1]^k \rightarrow \Re$, is a value function on the percentile space. Note that the percentile function q has domain $(0, 1]^k$, which contains the percentile set $\wp^k$. Therefore, a vector $\mathbf{p}' \in (0, 1]^k$ may not correspond to a percentile vector in the percentile set, where $q(\mathbf{p}')$ measures the desirability of solutions that have percentile values that are at least as high as values for each component in $\mathbf{p}'$.

The defined preferred Pareto optimal solution subset(s) can be obtained by solving the following discrete optimization problem, first introduced in Venkat et al. (2004), which optimizes the percentile function q.

$$\begin{aligned} \max \quad & q(p_1, p_2, \ldots, p_k) \\ \text{subject to:} \quad & |N_{sub}| \geq N' \\ & N_{sub} = \{\mathbf{x} \in S^{PO} : p_i(\mathbf{x}) \geq p_i, i = 1, 2, \ldots, k\}, \end{aligned} \tag{6.1.2}$$

where N' is the minimum number of solutions in the preferred subset of Pareto optima S^{PO}, and p_i, $i = 1, 2, \ldots, k$, correspond to the percentile *threshold* for each of the k objectives. This discrete optimization problem formulation is termed the Preferred Pareto Optimal Subset Problem (PPOSP).

Optimal solutions for the PPOSP are defined by percentile values $\mathbf{p}^* \in (0, 1]^k$, termed the *threshold percentile vector*, where $q(\mathbf{p}^*)$ is the maximum value such that there are at least N' solutions with percentile vectors that dominate $\mathbf{p}^*$, (i.e., if $\mathbf{p}^* = (p_1^*, p_2^*, \ldots, p_k^*)$, then a percentile vector $\mathbf{p'} = (p'_1, p'_2, \ldots, p'_k)$ dominates $\mathbf{p}^*$ if for every $i = 1, 2, \ldots, k$, $p'_i \geq p_i^*$). The threshold percentile vector defines the preferred reduced solution set N_{sub} based on the percentile function q. Each solution in N_{sub} is more desirable, having higher q value than any solutions that do not dominate the threshold percentile vector. By design, the only subjective parameters are N' and the percentile function.

The PPOSP is formulated over the percentile set. There are several advantages in optimizing over the percentile set rather than the objective function space. In many real world multi-objective problems, the objective functions typically have different evaluation metrics and units. For example, objective functions can measure costs, distances or volume. There may also be a large range of values associated with the different objective functions. Normalizing and adjusting these values require application-dependent knowledge and expertise. The percentile set on the percentile space uses a ranking (ordinal) approach, which normalizes the different objective functions, comparing the relative order instead of the value of each objective function. Another advantage of working in the percentile space comes from a usability perspective. It is often much easier for a decision-marker to visualize solutions in terms of ranks as opposed to actual values. The ability to use actual values also require detailed expert knowledge of the problem, where as ordinal ranking allows for generalization. This ability to encapsulate the data for simpler representations can

be beneficial to the subsequent decision process. By transforming the data into percentiles, detailed information regarding the objective function values of the solutions will be lost. For example, the difference $p_k^1 - p_k^2$ may be small, while the difference $f_k(x_1) - f_k(x_2)$ may be large. However, the goal of PPOSP is to obtain a reduced preferred subset of Pareto optimal solutions with minimal threshold values, and not to single out the best preferred Pareto optimal solution. The final decision is still dependent on the decision-maker's final preferences. In addition, the PPOSP is not limited to the percentile set, other normalization approaches can be used to capture the differences in each objective function values.

The PPOSP can be generalized by using different range normalization approaches.

$$\begin{aligned} \max \quad & q(n_1, n_2, \ldots, n_k) \\ \text{subject to: } & |N_N| \geq N' \\ & N_N = \{\mathbf{x} \in S^{PO} : n_i(\mathbf{x}) \geq n_i, i = 1, 2, \ldots, k\}, \end{aligned} \tag{6.1.3}$$

where n_i, $i = 1, 2, \ldots, k$, correspond to the *threshold* of the normalized value for each of the k objectives. Within this framework, the normalization can be computed based on the relative distances between the ideal and Nadir points. Normalizing in this way, information associated with the objective function values is preserved. Note that although it is relatively straightforward to obtain the ideal point, it can be difficult to obtain the Nadir point. The ability for PPOSP to incorporate different normalization scheme provides significant flexibility for the decision-maker.

It is important to also note that the optimal threshold vector to the PPOSP may not be unique; it is possible to have multiple threshold vectors that maximize the percentile function value. Since each threshold vector uniquely defines a reduced subset N_{sub}, then each threshold vector may lead to different subsets N_{sub} that are optimal for PPOSP, and hence, the different optimal threshold vectors provide high

level indicators to the quality of Pareto optimal solutions in N_{sub}.

There are two preferential parameters in PPOSP, the size of the desire subset, N', and the structure of the percentile function, typically in the form of a value function (e.g., a convex combination of the objective functions). The optimal threshold percentile vector(s) for the PPOSP define(s) the preferred reduced subset of solutions, N_{sub}. Each of the threshold percentiles is analogous to the weight preferences used in the value function approach (Korhonen and Halme 1990). However, instead of manually assigning weight preferences for each objective function, this manual procedure is captured within the PPOSP, which provides a method for filtering undesirable solutions (i.e., solutions that do not satisfy the threshold values found by the PPOSP). Finding such a reduced subset of Pareto optimal solutions reduces the burden on the decision-maker to closely examine a large number of Pareto optimal solutions.

6.2 Complexity

This section shows that the corresponding decision PPOSP problem and the more general decision problem, without the Pareto property, are both *NP*-complete. For clarity, define e_i to be a vector of size k, where all components are 0 except for the i^{th} component. Now, a formulation of the corresponding decision problem for the PPOSP and its more general form are given.

Dominating Pareto Subset Problem (DPSP)

INSTANCE: Finite Pareto set $U \subset \mathbf{Z}^k$, $|U| = N$, positive integer B and N', where $N' < N$.

QUESTION: Does there exist a subset $U' \subset U$, such that $\sum_{i=1}^{k} \min_{u \in U'}(e_i \cdot u) \geq B$ and $|U'| \geq N'$?

A more general formulation of the DPSP is to remove the Pareto restriction on

the set U.

Dominating Subset Problem (DSP)

INSTANCE: Finite set $U \subset \mathbf{Z}^k$, $|U| = N$, positive integer B and N', where $N' < N$.

QUESTION: Does there exist a subset $U' \subset U$, such that $\sum_{i=1}^{k} \min_{u \in U'}(e_i \cdot u) \geq B$ and $|U'| \geq N'$?

Prior to showing the complexity results of the DPSP and DSP, it is necessary to introduce the Maximum Edge Biclique Problem, which is *NP*-complete (Peeters 2003), and its variation, Max *N*-*M* Biclique Problem.

Maximum Edge Biclique Problem (MEBP)

INSTANCE: Bipartite graph $G = (V_1 \cup V_2, E)$, positive integer $K \leq |E|$.

QUESTION: Does G contain a biclique with at least k edges?

Max *N*-*M* Biclique Problem (Max NMBP)

INSTANCE: Bipartite graph $G = (V_1 \cup V_2, E)$, positive integer $N \leq |V_1|$, $M \leq |V_2|$.

QUESTION: Does G contain a biclique $K_{i,j}$ where $i \geq M$ and $j \geq N$?

The MEBP can be used to show that the Max NMBP is *NP*-complete.

Lemma 1 *Max NMBP is NP-complete.*

Proof: To prove that Max NMBP is *NP*-complete, first show that Max NMBP is in *NP*, and then proves that it is *NP*-complete by showing that there is a polynomial time reduction of MEBP to Max NMBP.

Given a biclique subgraph $G' = (V_1' \cup V_2', E')$, it takes $O(|V_1'| + |V_2'| + |E|)$ time to verify that it is a biclique and that it is a subgraph of G. Therefore, Max NMBP is in *NP*.

Given an arbitrary instance of MEBP ϕ, define k particular instances of Max

NMBP $\rho_i, i = 1, 2, \ldots, k$. Define ρ_i to have the same G as ϕ, $N = i$ and $M = \lceil k/i \rceil$. This transformation takes constant time for each ρ_i, and $O(k)$ time for all k instances. To complete the proof, it is necessary to show that there is *yes* response for ϕ if and only if there is a *yes* response to any ρ_i.

Suppose that the answer to an arbitrary instance of ϕ is *yes*. This implies that there exist a subgraph $G'(V_1' \cup V_2', E')$, where G' is a biclique and that $|E'| \geq k$. By the design of the reduction, each possible minimal subset combination of $|V_1'| = N$ and $|V_2'| = M$ are considered. Therefore, at least one instances of ρ_i must be a *yes*.

Suppose that the answer to one of the particular instances of ρ is *yes*. This implies that there exist a subgraph $G'(V_1' \cup V_2', E')$ where G' is a biclique and that $|V_1| \geq N$ and $|V_2'| \geq M$. By the design of the transformation, $N = i$ and $M = \lceil k/i \rceil$. The number of edges in G' is $|E'| = N \cdot M = i \cdot \lceil k/i \rceil \geq k$. Therefore, G' is a subgraph of G that is a biclique with k edges, and hence, the answer to ϕ must be *yes* □

By using the Max NMBP, it can be shown that the general DSP and DPSP are both *NP*-complete.

Theorem 14 *DSP is NP-complete.*

Proof: Given a subset $U' \subset U$, it takes $O(|U'|)$ time to verify that $\min_{u \in U'}(u_i) \geq B$ and that there are at least N' elements. Therefore DSP is in *NP*.

Given an arbitrary instance of the Max NMBP ϕ, define a particular instance of DSP ρ as follow: Without loss of generality let the set V_1 correspond to U, namely, each node $v_1 \in V_1$ is a $|k|$-tuple where $k = |V_2|$. Also let V_2 be an ordered set such that each node $v_2 \in V_2$ is labeled $l(v_2)$, where $l : V_2 \rightarrow \{1, 2, \ldots, |V_2|\}$. Each node v_2 corresponds to the $l(v_2)^{th}$ component for each of the $|V_2|$-tuple in U. For each element $v_1 \in V_1$ and it's corresponding $|V_2|$-tuple, the $l(v_2)^{th}$ component is 1 (0) if and only if there is (not) an edge $(v_1, v_2) \in E$. This defines the set U. Lastly, let $N' = N$ and $B = M$. This reduction takes $O(|V_1| \cdot |V_2|)$ time. To complete the proof,

it is necessary to show that there is a *yes* response for ϕ if and only if there is *yes* response for ρ.

Suppose that the answer to an arbitrary instance ϕ is *yes*. This implies that there exist a subgraph $G'(V_1' \cup V_2', E')$ where G' is a biclique and that $|V_1'| \geq N$ and $|V_2'| \geq M$. By the transformation, each node $v_1' \in V_1'$ corresponds to an element in U'. Since every node v_1' is adjacent to every node $v_2' \in V_2'$, then each of such $|V_2|$-tuple will have a 1 in the corresponding $l(v_2')^{th}$ component. Therefore, the summation of the minimum value of each component over U' must be at least M. Since $|V_2'| \geq M$,$|V_1| \geq N$, $B = M$, and $N' = N$, then the corresponding U' defined by V_1' will have at least N' elements where the summation of the minimum value of each component is at least B, which means that answer to ρ must also be *yes*.

Suppose that the answer to the particular instance ρ is *yes*. This implies that there exist a subset U' such that $|U'| \geq N'$ and that $\sum_{i=1}^{k} \min_{u \in U'}(e_i \cdot u_i) \geq B$. Since each k-tuple consist of either 0 or 1, then in order for $\sum_{i=1}^{k} \min_{u \in U'}(e_i \cdot u_i) \geq B$, there must exist B components with value of over all $u' \in U'$. From the transformation, for each u', the corresponding v_1' must be adjacent to v_2' hence the $l(v_2')^{th}$ component is 1. Since every v_1' shares B such common components (namely V_2'), then the sets of nodes V_1' and V_2' and edges (v_1', v_2') for all $v_1' \in V_1'$ and $v_2' \in V_2'\}$, form a biclique. Lastly, since $|V_1'| = |U'| = N' = N$ and $|V_2'| = B = M$, then it is a N-M Biclique, and hence, the answer to ϕ must also be *yes*. □

Theorem 15 *DPSP is NP-complete*

Proof: Given a subset $U' \subset U$, it takes $O(|U'|)$ time to verify that $\sum_{i=1}^{k} \min_{u \in U'}(u_i) \geq B$ and that there are at least N' elements. Therefore, DPSP is in *NP*.

Given an arbitrary instance of DSP, ϕ, it can be reduced in polynomial time to a particular instance DPSP, ρ, such that a solution exist for ϕ if and only if a solution exist for ρ. Given ϕ, the transformation converts the non-Pareto set U to a Pareto

set U_p. Without loss of generality, let U be an ordered set, where each element $u \in U$ has a corresponding label $l(u)$ with $l : U \rightarrow \{1, 2, \ldots, |U|\}$. Construct U_p in the following way: For each element u, append a $|U|$-tuple $\in \{0, 1\}^{|U|}$, where the $l(u)^{th}$ component is 1 and 0 for all other components (i.e., if $u = (u_1, \ldots, u_k)$ then $u_p = (u_1, \ldots, u_k, 0, \ldots, 0, 1, 0, \ldots, 0)$). Since only the u element has entry of 1 in the $l(u)^{th}$ component in the appended $|U|$-tuple, then the new set U_p is by design Pareto. Let N' and B remain the same. This transformation takes $O(|U|)$ time. To complete the proof, it is necessary to show that there is a *yes* response for ϕ if and only if there is a *yes* response for ρ.

First, consider the case where $N' > 1$. Suppose that the answer to an arbitrary instance ϕ is *yes*. This implies that there exist a subset U', where $|U'| \geq N'$ and $\sum_{i=1}^{k} \min_{u \in U'}(e_i \cdot u_i) \geq B$. Note since the transformed U'_p with the appended $|U|$-tuples of 0's and 1's does not affect the sum, then the answer to ρ must be *yes*.

Suppose that the answer to the particular instance ρ is *yes*. Then there is a Pareto subset U'_p, where $|U'_p| \geq N'$ and $\sum_{i=1}^{k} \min_{u'_p \in U'_p}(e_i \cdot u_i) \geq B$. From the transformation, the appended $|U|$-tuple to each element $u \in U$ has entry 1 only at the $l(u)^{th}$ component, which implies that no two elements in U_p have entry 1 at the same component in the appended $|U|$-tuple. Since $N' > 1$, then the minimum value for each component in the appended $|U|$-tuple must be 0, which does not affect $\sum_{i=1}^{k} \min_{u_p \in U'_p}(e_i \cdot u_i)$. Therefore, such a U'_p exist for ρ, then the same set excluding the appended $|U|$-tuples will also satisfy ϕ, which implies that the answer to ϕ must be *yes*.

Lastly, for the special case $N' = 1$, it is trivial case that takes $O(|U|)$ time. This is because one can examine each percentile vector individually. □

The DPSP is polynomial for $k = 2$ (i.e., for a bi-objective problem, the optimal subset N_{sub} can be found in $\mathrm{O}(|S^{PO}| \log |S^{PO}|)$ time). To see this, sorting the solution percentile vector along a single objective function provides an ordering, which also implicitly provides an ordering for the second objective function (due to the Pareto

property). Enumerating all consecutive N' subsets of the ordered set finds the optimal subset of Pareto optimal solutions (see Deterministic Sorted Local Search in Section 6.3.3).

6.3 Algorithms and Heuristics

This section introduces two exact algorithms and five heuristics for finding optimal/near-optimal solutions for the PPOSP. Section 6.3.1 describes two different enumeration approaches for the two exact algorithms. Sections 6.3.2 and 46.3.3 describe five heuristics, which can be classified as constructive and local search heuristics. The GR algorithm [103] is also re-examined in Section 6.3.4. Pseudo code for these algorithms and heuristics can be found in [52].

6.3.1 Exact Algorithms

Two different enumeration approaches are presented for solving the PPOSP. Since the threshold percentile vectors define unique subsets of Pareto optimal solutions, the PPOSP can also be solved by enumerating over all threshold percentile vectors. This enumeration takes $O(|S^{PO}|^k)$ time. Alternatively, another approach is to enumerates all possible subsets of Pareto optimal solutions of size N'. This enumeration takes $O(|S^{PO}|^{N'})$ time. Clearly, depending on the parameters S^{PO}, N' and k, the two different brute force enumerations result in different running time performances. This subsection formulates two different algorithms that solve the PPOSP using these two different underlying enumeration approaches.

Diagonal Enumeration

The Diagonal Enumeration (DE) algorithm is a modification of the first brute force enumeration approaches described above. The DE algorithm *avoids* enumerating over

all combinations of threshold percentile values. Depending on the threshold percentile vector, the corresponding subset N_{sub} may have size less than N'. In order for the percentile value function to be maximized with respect to N', the size of N_{sub} must equal N'. If $|N_{sub}| > N'$, by reducing the size of N_{sub}, q will either remain the same or increase. Lemma 2 states this formally.

Lemma 2 *If $U \subset S^{PO}$ and $U' \subset U$, where $\mathbf{p}$ and $\mathbf{p}'$ are the corresponding threshold percentile vectors associated with U and U', respectively, then $q(\mathbf{p}) \leq q(\mathbf{p}')$.*

Proof: Since every $\mathbf{u} \in U$ dominates $\mathbf{p}$, then there must exist a $\hat{\mathbf{u}} \in U$ such that $\hat{u}_i = p_i$, for at least one $i = 1, 2, \ldots, k$. Let $\hat{u}_i = p_i$ for some $i \in \{1, 2, \ldots, k\}$. U' can be one of two possible kinds of subsets; either $\hat{\mathbf{u}} \in U'$ or $\hat{\mathbf{u}} \notin U'$. If $\hat{\mathbf{u}} \in U'$, then $\mathbf{p} = \mathbf{p}'$, and hence, $q(\mathbf{p}) = q(\mathbf{p}')$. If $\hat{\mathbf{u}} \notin U'$, and since $\hat{\mathbf{u}}$ is removed from U, and $p'_i \geq p_i = \hat{u}_i$, then $q(\mathbf{p}) \leq q(\mathbf{p}')$. □

The DE algorithm exploits the results in Lemma 2 to avoid performing a full enumeration by constructing a k-dimensional table (called the *DE_table*), where each entry within the table corresponds to a subset of Pareto optimal solutions. By design, each of the k dimensions corresponds to the k objective functions, where the indices along each of the dimensions corresponds to percentile values. These indices also represent the sorted order of the percentile values (i.e., index i along dimension j corresponds to the i^{th} smallest percentile value of the j^{th} objective function.) The index of each entry can therefore be mapped to a valid threshold percentile vector. For example, let p_j^i denote the i^{th} percentile value in objective function j. Then if there are three objective functions (i.e., $k = 3$), an index in the k-dimensional *DE_table*, (x, y, z) would corresponds to the threshold percentile vector (p_1^x, p_2^y, p_3^z), and the entry that corresponds to index (x, y, z), *DE_table*$[x, y, z]$ would contain the subset of Pareto optimal solutions defined by the threshold percentile vector (p_1^x, p_2^y, p_3^z).

The enumeration is done by systematically constructing the *DE_table*, where each entry *DE_table* $[x_1, x_2, \ldots, x_k]$ can be constructed by taking the set intersection of *DE_table*$[x_1, x_2-1, x_3-1, \ldots, x_k-1]$, *DE_table*$[x_1-1, x_2, x_3-1, x_4-1, \ldots, x_k-1], \ldots,$ *DE_table*$[x_1 - 1, x_2 - 1, \ldots x_{k-1} - 1, x_k]$ (see Figure 6.1). The algorithm constructs the *DE_table* in a diagonal manner (as illustrated in Figure 6.2). The advantages in constructing the *DE_table* in such a manner is to avoid a full enumeration. If all entries along a single diagonal pass of the *DE_table* fail to contain at least N' elements, then the enumeration process can be terminated, since all diagonal passes thereafter will only contain percentile vectors with larger components. Furthermore, it is unnecessary to enumerate indices along a particular dimension if the size of the corresponding subsets are less than N' (i.e., if entry *DE_table*$[x, y, z]$ contain less then N' elements, it is unnecessary to enumerate entries with index (i, y, z), where $i > x$, (x, j, z), where $j > y$, and (x, y, k), where $k > z$). In the worst case, this algorithm will construct the entire *DE_table*, and hence, the running time is $O(|S^{PO}|^k)$.

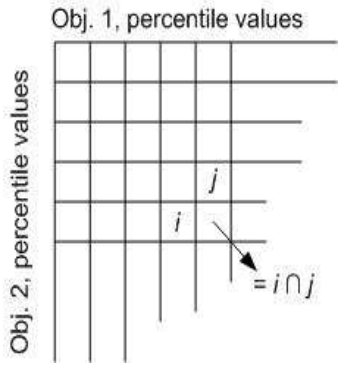

Figure 6.1: Two Dimensional Example of *DE_table* for the PPOSP.

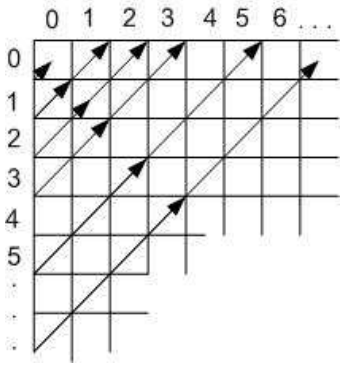

Figure 6.2: Two Dimensional Traversal of *DE_table* for the PPOSP.

Branch and Cut Algorithm

The DE algorithm solves the PPOSP by enumerating all combinations of percentile values of the threshold percentile vector. Alternatively the PPOSP can be solved by enumerating all subsets of Pareto optimal solutions of size N'. This enumeration approach is used to construct the Branch and Cut (BC) algorithm.

This enumeration approach can be done by constructing $|S^{PO}|$ search trees, where each node of a search tree corresponds to a subset of Pareto optimal solutions, and the root of each search tree is a unique element of S^{PO}. The second level of each of the search trees consists of all 2-element subsets constructed by adding a new element to the root. The third level consists of all 3-element subsets by adding a new element to its parent. Each level of the search trees is constructed by adding a new element to the parent. Therefore, each search tree will have at most N' levels, where if all of such search trees are fully constructed, then this corresponds to enumerating all N' subsets.

The BC algorithm constructs each N' search trees, starting at the root. However, it avoids performing a full enumeration by deciding whether to branch or cut at each node of the different search trees. Since each node in the search tree corresponds to a subset of Pareto optimal solutions, the corresponding percentile function value

can also be calculated. If at any node, the percentile function value is less than the current best percentile function value of a subset with N' elements, a cut is performed at that node and further enumeration along that branch is unnecessary, since from Lemma 2, any further branching along such nodes will only decrease the percentile function value.

A random subset of Pareto optimal solutions of size N' is generated for the initial best-to-date percentile function value. The higher the initial percentile function value, the less branching that is needed for the enumeration. However, in the worst case, the BC algorithm corresponds to enumerating all subsets of Pareto optimal solutions of size N', and hence the worst case running time is $O(|S^{PO}|^{N'})$.

6.3.2 Constructive Heuristics

This subsection introduces two constructive heuristics for finding good solutions to the PPOSP. The Greedy Constructive Elimination heuristic creates a preferred subset of Pareto optimal solutions by eliminating elements from S^{PO} until the size of the preferred subset is N'. In contrast, the Greedy Constructive Expansion heuristic builds a preferred subset of Pareto optimal solutions by adding elements to an empty set until the size of the subset is N'. Both of these heuristics use a greedy selection rule.

Greedy Constructive Elimination

The Greedy Constructive Elimination (GC-) heuristic starts by considering the full set of Pareto optimal solutions S^{PO}. It then finds a subset of S^{PO} of size N' by iteratively eliminating elements from S^{PO}. The percentile vector, which provides the best improvement over the percentile function value if it is removed, is eliminated at each iterative step. In the case of ties, a randomly selected percentile vector among the ties is eliminated. This heuristic has running time of $O(|S^{PO}|)$.

Greedy Constructive Expansion

The Greedy Constructive Expansion (GC+) heuristic is motivated by the Branch and Cut algorithm. Like the BC algorithm, it starts by considering $|S^{PO}|$ subsets of Pareto optimal solutions, each with a single distinct element of S^{PO}. However, unlike the BC algorithm, at each level in constructing a search tree, the GC+ heuristic greedily selects the best node to branch (i.e., an element is added to the current subset(parent) only if it decreases the percentile function value of the current subset the least). In the case of ties, a random solution is selected. A cut is performed, as in the BC algorithm, based on the best-to-date percentile function value. The GC+ heuristic builds $|S^{PO}|$ such search trees with distinct roots, where each search tree is a simple path of length at most N'.

Since each of the elements in S^{PO} are used as the initial subsets, there could be $|S^{PO}|$ different subsets of Pareto optimal solutions of size N' (i.e., each of the $|S^{PO}|$ different search trees) . The intuition behind this heuristic is to find an optimal constructive ordering (i.e., an optimal ordering of increasing the initial subset such that the resulting subset of Pareto optimal solutions is optimal), where constructing each subset of Pareto optimal solutions takes $O(|S^{PO}| \cdot N')$ time. Since there are $|S^{PO}|$ such starting subsets, the worst case running time for the GC+ heuristic is $O(|S^{PO}|^2 \cdot N')$.

6.3.3 Local Search Heuristics

This subsection introduces three local search heuristics. Local search heuristics are typically characterized by the following three steps:

1. Generate a feasible solution, s.

2. Attempt to find an improved feasible solution s' in a neighborhood of s.

3. If improved solution is found, replace s with s'. Repeat from Step 2.

The Deterministic Sorted Local Search heuristic examines subsets based on the sorted ordering of each objective function. This heuristic is different from the typical local search heuristic in that it uses a fixed deterministic neighborhood. The Element Exchange Local Search heuristic and the Percentile Neighborhood Local Search heuristic differ primarily in their neighborhood functions. While the Element Exchange Local Search heuristic defines its neighborhood function by altering the subset of Pareto optimal solutions, the Percentile Neighborhood Local Search heuristic defines its neighborhood function by perturbing the threshold percentile vector.

Deterministic Sorted Local Search

The Deterministic Sorted Local Search (DSLS) heuristic examines subsets of Pareto optimal solutions of size N' by only considering percentile vectors sorted by one of the objective functions. Therefore, S^{PO} is sorted k times by each objective function (i.e., there are k different sorted ordering of S^{PO}), where each of the k sorted orderings is examined by considering subsets of size N' with consecutive elements in the sorted S^{PO}. The best percentile function value found is then returned. Since traversing each sorted S^{PO} takes linear time, the sorting of S^{PO} dominates this heuristics' running time. In particular, the DSLS heuristic has running time $O(k|S^{PO}| \log |S^{PO}|)$.

Lemma 3 shows that in a bi-objective problem, a subset of Pareto optimal solutions cannot have the maximum percentile function value unless the subset contain only elements that are consecutive in a sorted ordering based on one of the objective functions. Using this result, the DSLS heuristic finds the optimal subset of Pareto optimal solutions for the bi-objective problem.

Lemma 3 *Let $U \subset S^{PO} \subset \Re^2$, $(\hat{u}_1, \hat{u}_2) \notin U$. If there exists some (u_1, u_2), $(v_1, v_2) \in U$ such that $u_1 > \hat{u}_1$ and $v_2 > \hat{u}_2$, then the corresponding percentile function value of*

U cannot be the optimal.

Proof: This result can be proved by constructing a new subset $\hat{U}$ with a larger percentile function value. Suppose that there exist such a $(\hat{u}_1, \hat{u}_2) \in U$. Furthermore, without loss of generality, let $(u_1, u_2) \in U$ such that $u_1 \geq u'_1$ for all $(u'_1, u'_2) \in U$, and let $(v_1, v_2) \in U$ such that $v_2 \geq u'_2$ for all $(u'_1, u'_2) \in U$. By the definition of the Pareto Property a new subset can be constructed, $\hat{U} = U/\{(u_1, u_2)\} \bigcup \{(\hat{u}_1, \hat{u}_2)\}$ or $\hat{U} = U/\{(v_1, v_2)\} \bigcup \{(\hat{u}_1, \hat{u}_2)\}$ will only increase the percentile function value. □

By the Pareto property, sorting S^{PO} based on one of the objective functions implicitly sorts the other objective function values. This ordering is a necessary condition for optimality, as shown in Lemma 3 for $k = 2$. Moreover, since Lemma 2 states that the optimal subset must be of size N', then the DSLS heuristic must find the optimal solution for the bi-objective problem.

Element Exchange Local Search

The Element Exchange Local Search (EELS) heuristic uses a single element exchange neighborhood function. The single element exchange neighborhood function transforms a feasible subset of Pareto optimal solutions by substituting percentile vectors in and out of the current feasible subset of Pareto optimal solutions. By design, this single element exchange neighborhood function can enumerate all possible subsets of size N'. This neighborhood function is quite general and provides limited direction for the local search. To provide more restrictions and to increase efficiency of the local search, two greedy modifications are added. The first modification forces the neighborhood function to greedily select the best element for the single element exchange, which provides the largest improvement to the percentile function value of the current feasible subset of Pareto optimal solutions. The second modification limits the candidate percentile vectors considered for the feasible subsets of Pareto

optimal solutions. An element that has been removed from the current subsets of Pareto optimal solutions is eliminated from any further consideration. The single element exchange neighborhood function is modified to only consider elements in the *pool*, defined as the set of candidate elements that have not been considered in any feasible subsets.

These two greedy modifications significantly increase the efficiency of the EELS heuristic. Since each percentile vector can be exchanged into a feasible subset at most once, and at each iteration there are at most $|S^{PO}|$ comparisons, then the worst case running time for a single starting initial feasible subset is $O(|S^{PO}|^2)$.

The single element exchange neighborhood function is of size $|S^{PO}|$. One variation of this neighborhood function is to perform multiple element exchanges. However, increasing the number of exchanges also increases the size of the neighborhood. Since the size of the neighborhood increases exponentially, greedily selecting the best percentile vector would be infeasible, although such an expanded neighborhood would reduce the number of local optima. To avoid being attracted to the same local optimum, the EELS heuristic is restarted with new random initial subsets. If the number of restarts is given by C, then the worst case running time for the EELS heuristic is $O(C \cdot |S^{PO}|^2)$.

Percentile Neighborhood Local Search

The Percentile Neighborhood Local Search (PNLS) heuristic is motivated by the DE algorithm. Recall that each entry in the *DE_table* corresponds to a subset of Pareto optimal solutions. The DE algorithm may enumerates many subsets of Pareto optimal solutions, with sizes much larger than N'. Lemma 2 shows that these subsets of Pareto optimal solutions are not optimal. The PNLS heuristic modifies the DE algorithm by avoiding enumeration of entries with corresponding subsets of size greater than N'.

The neighborhood function for the PNLS heuristic maps each entry in the *DE_table*

to a set of neighboring entries, where an entry is then visited based on the size constraint and the percentile function value. The neighbor of an entry is defined as follows: $(u_1, u_2, \ldots, u_k)$ is a neighbor of $(v_1, v_2, \ldots, v_k)$ if $|u_i - v_i| \leq 1$ for $i = 1, 2, \ldots, k$. The intuition behind this neighborhood function is that neighboring entries should correspond to subsets of similar sizes. By setting the initial entry with a corresponding subset of size N', this allows the heuristic to examine entries with corresponding subsets of similar sizes. In the worst case, this neighborhood function may enumerate the full *DE_table*.

The PNLS heuristic biases the neighbor selection to avoid enumerating the full *DE_table*. A new neighboring entry is selected based on the size of the corresponding subset as well as the corresponding percentile function value. Subsets of size N' with improving percentile function value are considered first. The heuristic terminates when a threshold, given by T, of non-improvement neighboring searches are made.

The PNLS heuristic is initialized at a starting entry where the size constraint is at equality. To find such a starting entry, select one objective function i, and set the percentile function vector to be $(0, \ldots, p_i, \ldots, 0)$. An entry with subset of size N' can be found by increasing the percentile value p_i, which then can be used as the initial entry for the PNLS heuristic. This can be repeated for each of the k objective functions. Since the PNLS heuristic searches for the optimal solution in a state space of size $|S^{PO}|^k$, then it has a worst case running time of $O(|S^{PO}|^k)$, similar to the DE algorithm.

6.3.4 Greedy Reduction Algorithm

The idea of capturing a preferred subset of Pareto optimal solutions by optimizing the PPOSP was introduced in [103]. They describe and analyze the Greedy Reduction (GR) algorithm for obtaining a subset of Pareto optimal solutions from a larger set of such solutions. The GR algorithm executes in linear time, $O(|S^{PO}|/N')$. This

Table 6.1: Counter Example for the GR Algorithm.

Percentile Values			
f_1	f_2	f_3	q
1.0	1.0	0.1	2.1
0.9	0.9	0.2	2.0
0.5	0.5	0.9	1.9
0.8	0.4	0.6	1.8
0.4	0.3	1.0	1.7
0.6	0.2	0.8	1.6
0.7	0.1	0.7	1.5
0.1	0.8	0.5	1.4
0.2	0.7	0.4	1.3
0.3	0.6	0.3	1.2

chapter also provides computational results of applying the GR algorithm to five multi-objective optimization problems. The Pareto optimal solution sets for each of these problems were generated by using five interactive optimization methods. Several different values of N' were tested with the GR algorithm, which provides an efficient way to generate a subset of Pareto optimal solutions from a larger set.

The GR algorithm attempts to maximize the percentile function q using a greedy element elimination strategy. At each iteration, it only considers the best N' solutions according to the ordering of the percentile function values. It then finds the corresponding threshold vector that satisfies these N' solutions, and eliminates all the solutions that fail the threshold. This is repeated until no solutions remain. The maximum threshold vector obtained across all iterations is the resulting solution.

The drawback of such an approach is that solutions may be eliminated prematurely. Greedily selecting the top q values does not measure the potential contribution of individual percentile values. Table 6.1 provides a counterexample to the optimality result reported in Venkat et al. [103]. Each row in the table corresponds to an element in $\wp^3$. There are ten elements in the Pareto optimal solution set. If the decision-maker wants a reduced set of size $N' = 3$, the GR algorithm will fail to find the reduced subset, and hence, contradicts the optimality result of the GR algorithm reported in

Table 6.2: First Iteration of the GR Algorithm Applied to Table 6.1 Example.

Percentile Values			
f_1	f_2	f_3	q
1.0	1.0	0.1	2.1
0.9	0.9	0.2	2.0
0.5	0.5	0.9	1.9
0.8	*0.4*	*0.6*	*1.8*
0.4	*0.3*	*1.0*	*1.7*
0.6	*0.2*	*0.8*	*1.6*
0.7	*0.1*	*0.7*	*1.5*
0.1	*0.8*	*0.5*	*1.4*
0.2	*0.7*	*0.4*	*1.3*
0.3	*0.6*	*0.3*	*1.2*

Venkat et al. [103]. Table 6.2 depicts the first iteration of the GR algorithm. The top three solutions, rows 1, 2, 3, are selected, and $\mathbf{p}^{min} = (0.5, 0.5, 0.1)$ as indicated by the boxed values in Table 6.2. The remaining solutions are then eliminated from consideration, failing to meet the threshold. The algorithm terminates after the first iteration, returning the first three solutions with percentile function value 1.1. However, 1.1 is not the optimal value for this instance. In particular, if rows 1, 2, 4 are selected, then the optimal solution is found with $p^{min} = (0.8, 0.4, 0.1)$, and percentile function value 1.3.

6.4 Computational Results

This section reports computational results of the algorithms and heuristics described in Section 6.3, applied to five multi-objective optimization problems. Test Problem 1 consists of three non-linear convex objective functions with bounded constraints. Test Problem 2, adapted from [82], consists of three non-linear convex objective functions with non-linear constraints. Test Problem 3, taken from Van Veldhuizen [102], consists of three non-linear non-convex objective functions with bounded constraints. Test Problem 4 is a randomly generated Pareto optimal solution set in $\Re^4$, containing

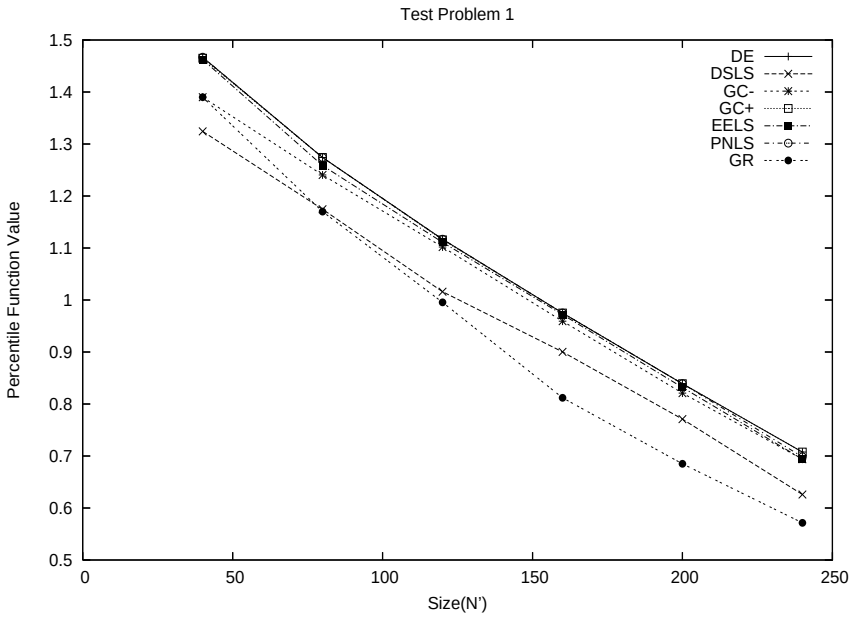

Figure 6.3: The PPOSP: Test Problem 1 Percentile Function Value Results.

2000 Pareto optimal solutions, while Test Problem 5 is a randomly generated non-Pareto set in $\Re^4$, with 2000 solutions over the same variable range for Test Problem 4. See Venkat et al. [103] for specific details of these problems.

An exhaustive enumeration procedure was executed to obtain the true Pareto optimal solution set for Test Problems 1, 2, 3. In particular, the feasible region is sampled via a fine grid to capture the true Pareto optimal solution sets. Because of the size of Test Problems 4, 5, only computational results with the five heuristics are reported. Other enumeration and approximation methods to generate the set of Pareto optimal solutions can be found in Ehrgott [32], Ehrgott and Gandibleux [33] and Miettinen [77].

The PPOSP is formulated for each test problem, where the percentile function is of the form, $q(p_1, p_2, \ldots, p_k) = \sum_{i=1}^{k} p_i$. Figures 6.3, 6.4, 6.5 report computational results using the DE algorithm and heuristics for Test Problems 1, 2, 3, respectively. The EELS heuristic and the PNLS heuristic were repeated ten times, using a new random initial solution for each run. The threshold used for the PNLS heuristic

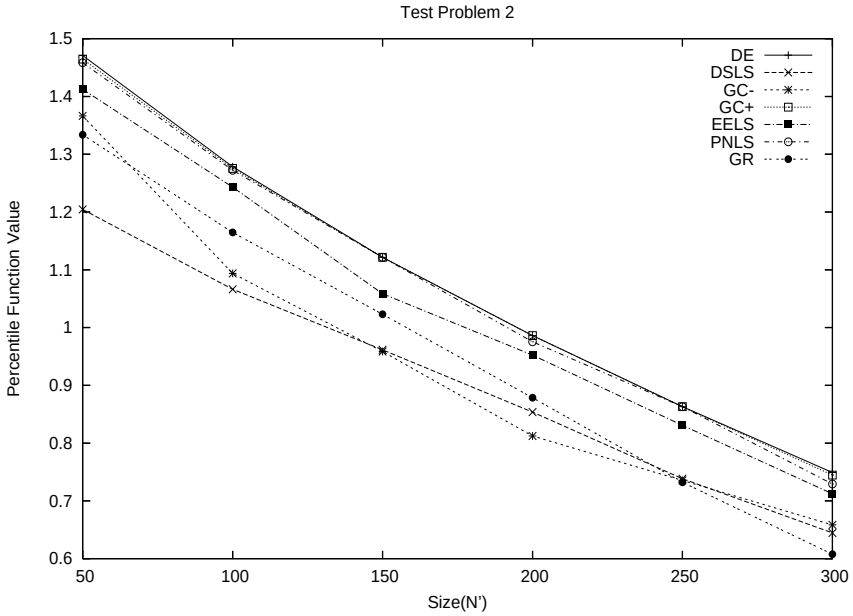

Figure 6.4: The PPOSP: Test Problem 2 Percentile Function Value Results.

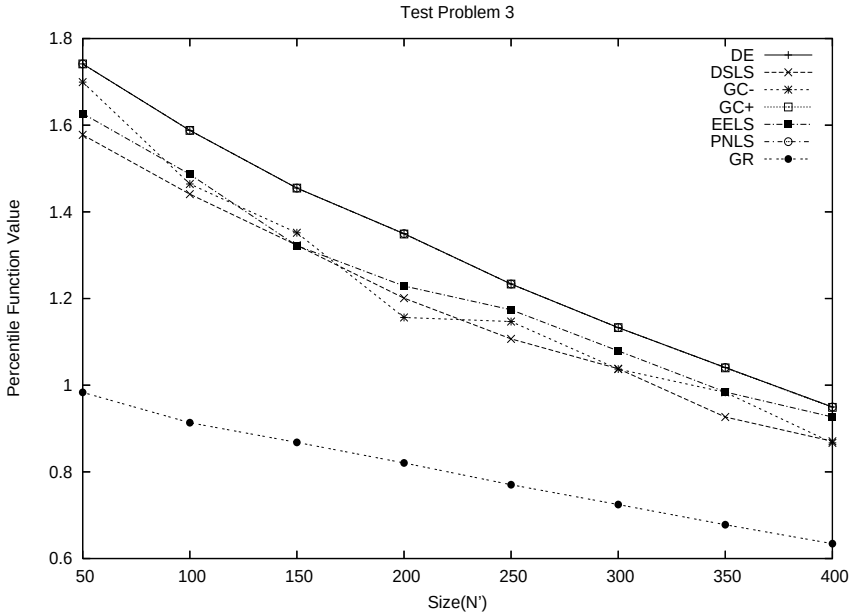

Figure 6.5: The PPOSP: Test Problem 3 Percentile Function Value Results.

Table 6.3: The PPOSP: Percentile Function Value Ratio

Test Problem 1

N'	DSLS/DE	GC-/DE	GC+/DE	EELS/DE	PNLS/DE	GR/DE
40	0.902	0.947	0.998	0.996	1.0	0.947
80	0.921	0.973	1.0	0.987	1.0	0.918
120	0.908	0.985	0.997	0.993	1.0	0.890
160	0.923	0.983	1.0	0.997	1.0	0.832
200	0.918	0.978	1.0	0.991	1.0	0.816
240	0.884	0.980	1.0	0.980	0.987	0.807

Test Problem 2

N'	DSLS/DE	GC-/DE	GC+/DE	EELS/DE	PNLS/DE	GR/DE
50	0.819	0.928	0.995	0.960	0.991	0.906
100	0.833	0.855	0.997	0.972	0.995	0.910
150	0.857	0.854	1.0	0.943	1.0	0.912
200	0.865	0.823	1.0	0.965	0.989	0.890
250	0.855	0.852	1.0	0.962	1.0	0.848
300	0.860	0.878	0.993	0.950	0.973	0.811

Test Problem 3

N'	DSLS/DE	GC-/DE	GC+/DE	EELS/DE	PNLS/DE	GR/DE
50	0.905	0.975	1.0	0.934	1.0	0.564
100	0.907	0.922	1.0	0.936	1.0	0.575
150	0.909	0.928	1.0	0.908	1.0	0.596
200	0.889	0.856	0.999	0.910	1.0	0.607
250	0.897	0.929	1.0	0.951	1.0	0.624
300	0.916	0.915	1.0	0.952	1.0	0.639
350	0.890	0.944	0.999	0.945	1.0	0.651
400	0.916	0.912	0.998	0.975	1.0	0.667

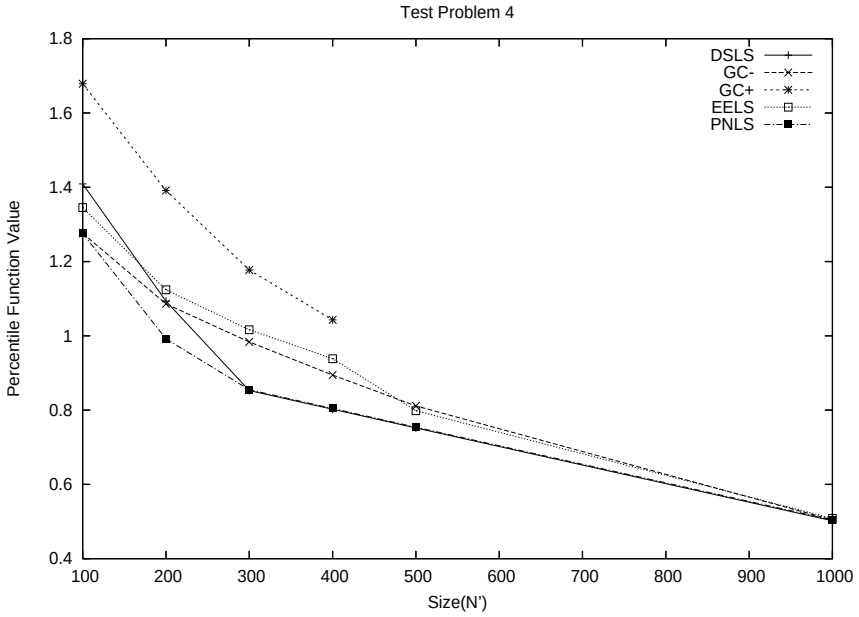

Figure 6.6: The PPOSP: Test Problem 4 Percentile Function Value Results.

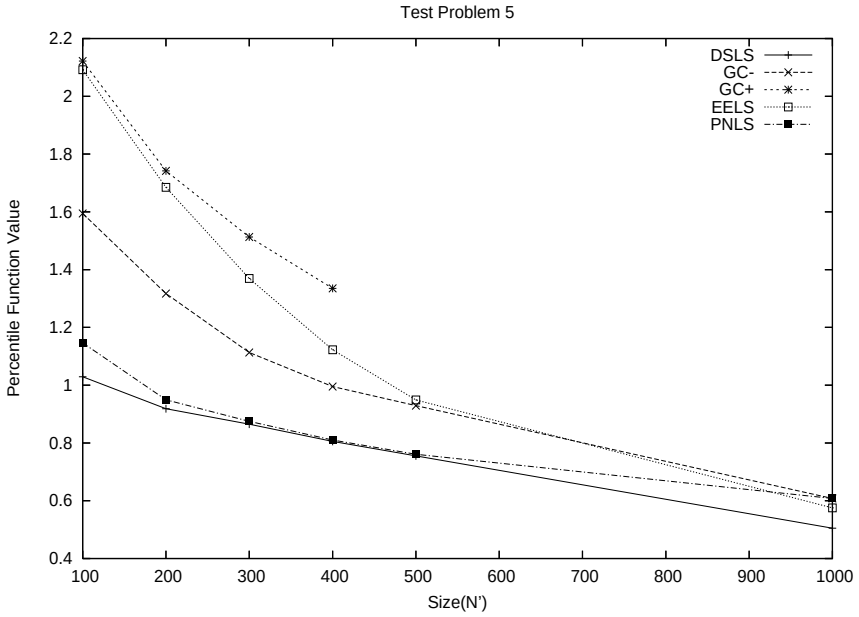

Figure 6.7: The PPOSP: Test Problem 5 Percentile Function Value Results.

was 1000 non-improving iterations. The DE algorithm and each of the local search heuristics where initialized as described in Section 6.3. Table 6.3 reports the ratio of the percentile function value found by each of the heuristics over the optimal percentile function value obtained by the DE algorithm. The numbers of Pareto optimal solutions generated with the sampling scheme for Test Problems 1, 2, 3, were 441, 650, and 1075, respectively. Different ranges of N' were used for each of the test problems. Size parameter $N' = 40$, 80, 120, 160, 200, 240, $N' = 50$, 100, 150, 200, 250, 300, and $N' = 50$, 100, 150, 200, 250, 300, 350, 400 were applied to Test Problems 1, 2, 3, respectively. Note that as N' increased to $|S^{PO}|$, the percentile function value solved by each of the heuristics converges. In particular, if $N' = |S^{PO}|$, then the optimal percentile function value will be $\sum_{i=1}^{k} \min_{\mathbf{P} \in S^{PO}} p_i$, which is depicted in Figures 6.6, 6.7 for the two larger test problems.

The computational results reported for Test Problems 1, 2, 3 suggest that the GC+ heuristic and the PNLS heuristic can be very effective in finding the optimal subset of Pareto optimal solutions. Figures 6.4, 6.5 depict a comparison of the computational results obtained from applying the heuristics to Test Problems 1, 2, 3. From Table 6.3, the GC+ heuristic found the optimal solutions 12 out of 20 experimental runs, and the PNLS heuristic found the optimal solutions 15 out of the 20 experimental runs, including all the optimal solutions for Test Problem 3 for each N'. The lowest GC+/DE ratio and PNLS/DE ratio across Test Problems 1, 2, 3 are 0.995 and 0.973, respectively. Although the EENS heuristic did not find the optimal solutions for Test Problems 1, 2, 3, it was still very efficient, always obtaining solutions within 10 percent of the optimal solutions. The GC- heuristic, the DSLS heuristic, and the GR algorithm always found solutions within 20 percent of the optimal solutions for Test Problems 1, 2. However, as the size of S^{PO} increased, the quality of solutions found by the GR algorithm degraded, as illustrated by Test Problem 3 (see Figure 6.5). The GC- heuristic and DSLS heuristic found solutions within 15 percent of the optimal

solutions for Test Problem 3, while the GR algorithm failed to find any solutions within 30 percent of the optimal solutions. Moreover, as N' increased, quality of solutions found by the GR algorithm degraded, while this is not the case for the heuristics introduced in this chapter.

Figures 6.6, 6.7 depict a comparison of the computational results obtain from applying the five heuristics to Test Problems 4, 5, respectively. These heuristics were applied to these two test problems with $N' = 100, 200, 300, 400, 500, 1000$. The PNLS heuristic and the EELS heuristic were repeated with ten different randomly generated initial solutions. The threshold used for the PNLS heuristic was 1000 non-improving iterations. The optimal solutions for Test Problems 4, 5 are unknown. However, by comparing the percentile function values, the GC+ heuristic clearly out performs the other four heuristics. The EELS heuristic obtained larger percentile function value in Test Problem 5; this is most likely due to the relaxation of the Pareto property in Test Problem 5. Since the EELS heuristic uses an element exchange neighborhood function, the Pareto property ensures that when exchanging an element from the preferred subset, some component of the threshold percentile vector must increase while others must either remain constant or decrease. However, without the Pareto restriction, the negative effect of exchanging a poor percentile vector into the preferred subset is mitigated (i.e., each components of the threshold percentile vector may all increase). It is not apparent whether the Pareto property has an effect on the other heuristics. Although the PNLS heuristic performed well for Test Problems 1, 2, 3, it performed poorly in Test Problems 4, 5, which is likely due to the large problem size.

Tables 6.4, 6.5 report the experimental running time for applying the algorithms and heuristics to the five test problems. All experiments were executed on a 997MHz Intel Pentium III processor. The GR algorithm and the GC- heuristic were the two fastest heuristics. The DE algorithm had the slowest experimental running time, which is not surprising since it had to perform an exponential time enumeration. Of

Table 6.4: Algorithms and Heuristics for the PPOSP: Average Running Time (CPU Seconds)

Test Problem 1

N'	DE	DSLS	GC-	GC+	EELS	PNLS	GR
40	49962	1.3	0.2	110	69	276	0.1
80	46268	1.7	0.2	451	134	278	0.21
120	39076	1.9	0.2	1120	162	316	0.2
160	28448	2.0	0.2	1891	193	329	0.2
200	22806	2.0	0.2	2762	176	340	0.15
240	14897	1.9	0.2	3773	180	350	0.3

Test Problem 2

N'	DE	DSLS	GC-	GC+	EELS	PNLS	GR
50	53312	2.3	0.3	279	154	726	0.2
100	134710	2.9	0.3	1238	300	580	0.2
150	136038	3.6	0.4	2930	431	758	0.1
200	142055	3.5	0.5	5941	512	474	0.1
250	144775	3.6	0.4	8682	533	609	0.1
300	104832	3.6	0.4	12436	543	553	0.1

Test Problem 3

N'	DE	DSLS	GC-	GC+	EELS	PNLS	GR
50	781127	4.3	0.1	108	333	445.2	0.2
100	886240	5.0	0.3	695	676	445.9	0.2
150	820453	5.8	0.4	1910	1529	534.4	0.3
200	967034	6.6	0.6	4343	1809	534.7	0.2
250	933062	7.3	0.7	8842	2119	628.6	0.2
300	932000	7.8	0.8	15444	2361	637.4	0.3
350	919328	8.2	1,0	25424	2579	765.3	0.2
400	1012952	8.7	1.2	36671	2701	662.6	0.2

the five heuristics, the GC+ heuristic had the slowest experimental running time, however it was also the most effective heuristic in finding optimal and near-optimal solutions. The GC+ heuristic also had memory limitations due to the recursive nature. On the other end of the quality performance trade-off spectrum, the PNLS heuristic was the fastest, but only managed to find solutions within 20 percent of the optimal solutions for the small problems. For Test Problems 4, 5, the PNLS heuristic found solutions that had significantly smaller percentile function value (i.e., up to 30 and 50 percent less than those solutions found by the GC+ heuristic in Test Problems 4, 5, respectively). The PNLS heuristic, the EELS heuristic, and the GC- heuristic provided a quality performance trade-off spectrum in decreasing experimental running time, respectively. For Test Problems 1, 2, 3, the quality of solutions is positively correlated with the increase of running time. However, for Test Problem 4, 5, this correlation did not follow for the PNLS heuristic. Note that although the experimental running time for the PNLS heuristic gracefully increased with the increase in the size of S^{PO}, the quality of solutions found were similar to those found by the GC- heuristic and the EELS heuristic, both of which had significantly faster experimental running times. Lastly, notice that as N' increased and approached $|S^{PO}|/2$, the experimental running time also increased, which corresponds to the worst case analysis where the number of possible subsets is maximized (i.e., $O(\frac{2^{|S^{PO}|}}{\sqrt{|S^{PO}|}})$).

6.5 Conclusion

Multi-objective optimization problems occur in numerous real-world applications. Solving such problems can yield large sets of Pareto optimal solutions. This chapter examined the question of identifying preferred subsets of Pareto optimal solutions. The formulation of the discrete optimization problem, PPOSP, is designed to assist a decision-maker in finding preferred subsets of Pareto optimal solutions. The PPOSP

Table 6.5: Algorithms and Heuristics for the PPOSP: Average Running Time (CPU Seconds)

Test Problem 4

N'	DSLS	GC-	GC+	EELS	PNLS
100	7	2.3	16180	748	1676
200	9	2.2	47090	1565	1793
300	11	2.2	86380	2442	1866
400	13	2.2	135500	3221	1933
500	15	2.1	-	3834	2011
1000	19	1.6	-	4866	2559

Test Problem 5

N'	DSLS	GC-	GC+	EELS	PNLS
100	7	2.4	2220	755	1743
200	9	2.3	19640	1638	1854
300	12	2.4	61790	2518	1818
400	14	2.3	117000	3459	1942
500	15	2.3	-	4195	2029
1000	21	1.8	-	5657	2549

is unique, in that it allows the decision-maker to obtain a desirable subset size N', based on threshold values for each objective functions. It does not require expert knowledge in finding such reduced preferred subset, which allows the decision-maker to focus on smaller sets of preferred Pareto optimal solutions. In addition, unlike typical value function approaches, the PPOSP is formulated (but not limited to) in the percentile space, which provides an ordinal approach in addressing the post-optimality selection problem.

The decision formulation of the PPOSP is formulated and proven to be *NP*-complete, which corrects the optimality results reported in Venkat et al. [103]. Two exact algorithms, the DE algorithm and the BC algorithm, are provided for solving the PPOSP to optimality. Five heuristics are also presented, which provide a spectrum of heuristics with varying trade-offs in solution quality and run time efficiency. The experimental results reported suggest that the GC+ heuristic can yield the best results, if running time can be sacrificed. Otherwise the EELS heuristic provided the best trade-off, efficiently returning quality solutions. The experimental results from

Test Problems 1, 2, 3 also suggest that the PNLS heuristic can be effective for smaller problems.

The heuristic presented in this chapter does not require the set of solutions to be Pareto. Although the decision problem for a non-Pareto set is also proven to be *NP*-complete, it is not clear what the impact of the Pareto property has on these heuristics. The Pareto property provides structure to the feasible solution set for the PPOSP. For bi-objective problems, the DSLS heuristic uses this structure to find the optimal solution. However, it is not apparent how one can exploit such structure in higher dimensional problems.

The PPOSP introduces a new approach to address the post-optimality selection problem. It provides a framework that defers the need of expert knowledge in the decision process, reducing the burden of the decision-maker to only focus on preferred reduced subsets of Pareto optimal solutions. The use of the percentile set provides one level of encapsulation. Providing higher levels of encapsulation, while retaining the consistency of the decision-maker preferences, is an area of current research activity. Another area of research is to address the scalability of the heuristics and algorithms higher dimensional problems. The ultimate goal of this effort is to design a fully automated post-optimality selection process.

Chapter 7

Summary

The research presented in this dissertation focuses on two topics in combinatorial optimization, designing efficient exact algorithms for several single machine scheduling problems, and formulating a discrete optimization problem for addressing the post-optimality selection problem.

The BB&R algorithms have been shown to outperform the current best algorithms in the literature for the $1|r_i|\sum U_i$, $1|r_i|\sum t_i$, and $1|ST_{sd}|\sum t_i$ scheduling problems. Computational results show that the BB&R algorithms are very effective, and that they are capable of solving even larger test instances than the ones reported in the literature. A new DBFS exploration strategy is also introduced and incorporated into the BB&R algorithms. By design, the DBFS exploration strategy works in conjunction with the memory-based dominance rules to explore fewer states. Chapter 3 and 4 show that the DBFS exploration strategy provides a significant computational speedup compared to DFS and best first search exploration strategies. Chapter 5 shows that the DBFS exploration strategy is comparable to the best first search strategy for the $1|ST_{sd}|\sum t_i$ scheduling problem. In addition, several new dominance rules and bounding schemes for these scheduling problems are also presented. The combination of explicit memorization of states, new exploration strategy, dominance rules, and improved bounds computation demonstrate that the BB&R algorithms are very efficient. These results show that the BB&R algorithms have the potential to solve other combinatorial problems.

Although the results of the BB&R algorithms for these three scheduling problems

presented in this dissertation are very promising, the BB&R algorithms do have their limitation. By explicitly storing every visited state, the BB&R algorithms can incur a significant memory overhead. This is most noticeable in Chapter 5 when the BB&R algorithm is used for solving the $1|ST_{sd}|\sum t_i$ scheduling problem. The time limitation imposed on the algorithm did not constrain the performance of the algorithm, whereas, the memory limitation caused many unsolved problem instances. Despite the negative results due to the memory limitation, the effect of the memory limitation could be potentially curtailed by stronger dominance rules and bounding schemes. With stronger dominance rules and bounding schemes, this could provide early pruning of the search tree reducing the number of explicitly stored states while boosting the overall performance of the algorithm. However, if there are fewer states stored, this can reduce the effectiveness of the memory-based dominance rules. The key on improving the performance of the BB&R algorithms is to find a balance among the different components such that each component can benefit one another. In addition, it is also worthwhile to incorporate multi-core computing architecture technologies with the BB&R algorithm and DBFS exploration strategy. By design, the DBFS exploration strategy contains the features needed to take advantage of a distributed environment. Distributed computing strategy can provides substantial improvements in both memory management and computational processing time.

The research effort presented in Chapter 6 on the PPOSP formulation addresses the second topic of this dissertation on post-optimality selection. The new PPOSP formulation provides a framework that reduces the burden on the decision-maker by using limited expert knowledge to find a preferred reduced subset of Pareto optimal solutions. A new ordinal ranking approach is used in the PPOSP formulation that provides one level encapsulation. The PPOSP formulation can be viewed as a specific normalization procedure by using the percentile set. Other normalization approaches can be beneficial and might result in different preferred subsets of Pareto optimal

solutions. Using different normalization strategies can provide one method of sensitivity analysis to the PPOSP formulation. This sensitivity analysis could be helpful in assessing the benefit in using this framework. Furthermore, it would also be interesting to consider applying other scalarizing functions to the percentile vectors. This can also be very helpful in assessing the benefits of using an ordinal ranking approach and can also enhance the significance of the general framework.

References

[1] A. Allahverdi, C.T. Ng, T.C.E. Cheng, and M.Y. Kovalyov. A survey of scheduling problems with setup times or cost. *European Journal of Operational Research*, 187:985–1032, 2008.

[2] V.A. Armentano and R. Mazzini. A genetic algorithm for scheduling on a single machine with set-up times and due dates. *Production Planning and Control*, 11:985–1032, 2008.

[3] E. Baker, A. Joseph, A. Mehrotra, and M. Trick (eds). *Extending the Horizons: Advances in Computing, Optimization, and Decision Techbologies Series*. Springer, 2007.

[4] P. Baptiste, J. Carlier, and A. Jouglet. A branch-and-bound procedure to minimize total tardiness on one machine with arbitrary release dates. *European Journal of Operational Research*, 158:595–608, 2003.

[5] P. Baptiste, C. Le Pape, , and L. Perify. Global constraints for partial csps: A case-study of resource and due date constraints. In M. Maher and J.F. Puget, editors, *Principles and Practice of Constraint Programming - CP98, Lecture Notes in Computer Science*, volume 1520, pages 87–101. Springer, 1998.

[6] P. Baptiste, L. Peridy, and E. Pinson. A branch and bound to minimize the number of late jobs on a single machine with release time constraints. *European Journal of Operational Research*, 144:1–11, 2003.

[7] A. Baykasoglu, S. Owen, and N. Gindy. A taboo search based approach to find pareto optimal set in multiple objective optimization. *Journal of Engineering Optimization*, 31:731–748, 1999.

[8] D. Bedworth and J. Bailey. *Integrated Production Control Systems: Management, Analysis, Design*. Wiley and Son, Inc., 1987.

[9] R. Bellman. Bottleneck problems and dynamic programming. *Mathematics*, 39:947–951, 1953.

[10] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[11] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.

[12] D. Brown and C. White III (eds). *Operations Research and Artificial Intelligence: The Integration of Problem-Solving Strategies*. Kluwer Academic, 1990.

[13] P. Brucker. *Scheduling Algorithms, 2nd Edition*. Springer, 1999.

[14] S. Chang, Q. Lu, G. Tang, and W. Yu. On decomposition of the total tardiness problem. *Operations Research*, 17(5):221–229, 1995.

[15] H. Cho, S. Oh, and D. Choi. A new evolutionary programming approach based on simulated annealing with local cooling schedule. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pages 598–602, Anchorage, AK, 1998. IEEE Press.

[16] C. Chu. A branch-and-bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics*, 39:265–283, 1992.

[17] C. Chu and M.C. Portmann. Some new efficient methods to solve the $n|1|r_i|\sum t_i$ scheduling problem. *European Journal of Operational Research*, 58:404–413, 1992.

[18] C.A. Coello. An updated survey of evolutionary multiobjective optimization techniques: State of the art and future trends. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 3–13, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press.

[19] C.A. Coello and C. Romeros (eds). *Evolutionary Algorithms and Multiple Objective Optimization, Multiple Criteria Optimization-State of the Art Annotated Bibliographic Surveys*. Kluwer Academic, New York, 2002.

[20] C.A. Coello, D. Van Veldhuizen, and G.B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic, New York, 2002.

[21] P. Czyżak and A. Jaszkiewicz. Pareto simulated annealing- a metaheuristic technique for multiple-objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis*, 7:34–47, 1998.

[22] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[23] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of Operations Research*, 2:393–410, 1954.

[24] I. Das. A preference ordering among various pareto optimal alternatives. *Structural Optimization*, 18(1):30–35, 1999.

[25] S. Dauzère-Pérès. Minimizing late jobs in the general one machine scheduling problem. *European Journal of Operational Research*, 81:134–142, 1995.

[26] S. Dauzère-Pérès and M. Sevaux. Using lagrangean relaxation to minimize the weighted number of late jobs on a single machine. *Naval Research Logistics*, 50:273–288, 2003.

[27] S. Dauzère-Pérès and M. Sevaux. An exact method to minimize the number of tardy jobs in single machine scheduling. *Journal of Scheduling*, 7:405–420, 2004.

[28] K. Deb. *Evolutionary Algorithms for Multi-Criterion Optimization in Engineering Design*. Wiley and Son, Inc., 1999.

[29] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.P. Schwefel, editors, *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, Paris, France, 2000. Springer. Lecture Notes in Computer Science No. 1917.

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[31] J. Du and J. Y-T Leung. Minimizing total tardiness on one machine is *np*-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

[32] M. Ehrgott. *Multicriteria Optimization*. Springer, 2005.

[33] M. Ehrgott and X. Gandibleux (eds). *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*. Kluwer Academic, 2002.

[34] H. Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17(4):701–715, 1969.

[35] C.M. Fonseca and P.J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Genetic Algorithms: Proceedings of the Fifth International Conference*, pages 416–423. Morgan Kaufmann, 1993.

[36] C. Gagné, W.L. Price, and M. Gravel. Comparing an aco algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times. *Journal of the Operational Research Society*, 53(8):895–906, 2002.

[37] X. Gandibleux, N. Mezdaoui, and A. Fréville. A tabu search procedure to solve multiobjective combinatorial optimization problems. In R. Caballero, F. Ruiz, and R. Steuer, editors, *Advances in Multiple Objective and Goal Programming*, volume 455, pages 291–300. Springer, 1997.

[38] M. Garey and D. Johnson. *Computers and Intractability: Guide to the Theory of NP-Completeness*. WH Freeman and Company, 1979.

[39] R. Garfinkel and G. Nemhauser. *Integer Programming*. John Wiley Press, 1972.

[40] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[41] B. Golden, S. Raghava, and E. Wasil (eds). *The Next Wave in Computing, Optimization, and Decision Technologies*. Springer, 2005.

[42] C. Gomes. Challenges and opportunities in planning and scheduling. *The Knowledge Engineering Review*, 15:1–10, 2000.

[43] C. Gomes. On the intersection of ai and or. *The Knowledge Engineering Review*, 16:1–4, 2001.

[44] R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals Discrete Mathematics*, 4:287–326, 1979.

[45] S.R. Gupta and J.S. Smith. Algorithms for single machine total tardiness scheduling with sequence dependent setups. *European Journal of Operational Research*, 175:722–739, 2006.

[46] T. Hanne. Global multiobjective optimization using evolutionary algorithms. *Journal of Heuristics*, 6:347–360, 2000.

[47] M.P. Hansen. Tabu searh in multiobjective optimization: Mots. In *Proceedings of the 13th International Conference on Multiple Criteria Decision Making (MCDM-97)*, Cape Town, South Africa, 1997.

[48] H. Ishibuchi, T. Yoshida, and T. Murata. Balance Between Genetic Search and Local Search in Memetic Algorithms for Multiobjective Permutation Flowshop Scheduling. *Evolutionary Computation, IEEE Transactions on*, 7(2):204–223, 2003.

[49] A. Jaszkiewicz. *Multiple Objective Metaheuristic Algorithms for Combinatorial Optimization, Habilitation Thesis*. Poznan University of Technology, Poznan, Poland, 2001.

[50] A. Jouglet, P. Baptiste, and J. Carlier. Branch-and-bound algorithms for total weighted tardiness. In J. Y-T. Leung, editor, *Hangbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 307–328. CRC Press, 2004.

[51] A. Rinnooy Kan. *Machine Scheduling Problem: Classification, Complexity and Computation*. Nijhoff, Hague, 1976.

[52] G.K. Kao and S.H. Jacobson. Post-optimality algorithms and heuristic for multi-objective optimization. In *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, 2006.

[53] G.K. Kao and S.H. Jacobson. Finding preferred subsets of pareto optimal solutions. *Computational Optimization and Applications*, 40:73–95, 2008.

[54] G.K. Kao, E.C. Sewell, and S.H. Jacobson. Minimizing total tardiness for the single machine with sequence dependent setup time problem using the bb&r algorithm. In *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, 2008.

[55] G.K. Kao, E.C. Sewell, and S.H. Jacobson. A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem. *Journal of Scheduling*, (to appear).

[56] G.K. Kao, E.C. Sewell, S.H. Jacobson, and S.N. Hall. The distributed best first search exploration strategy: An illustrative example with the $1|r_i|\sum u_i$ scheduling problem. In *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, 2008.

[57] E.M. Kasprzak and K.E. Lewis. Pareto analysis in multiobjective optimization using the colinearity theorem and scaling method. *Structural and Multidisciplinary Optimization*, 22(3):208–218, 2001.

[58] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one-machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126, 1978.

[59] J. Knowles and D. Corne. M-PAES: A memetic algorithm for multiobjective optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 325–332, La Jolla Marriott Hotel La Jolla, California, USA, 6-9 2000. IEEE Press.

[60] T. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[61] P. Korhonen and M. Halme. Supporting the decision maker to find the most preferred solutions for a molp-problem. In *Proceedings of the 9th International Conference on Multiple Criteria Decision Making*, pages 173–183, Fairfax, Virginia, 1990.

[62] C. Koulamas. The total tardiness problem: Review and extensions. *Operations Research*, 42:1025–1041, 1994.

[63] J. Lasserre and M. Queyranne. Generic scheduling polyhedra and a new mixed integer formulation for single machine scheduling. In *Proceedings of the 2nd Integer Programming Conference*, pages 136–149, Pittsburgh, PA, 1992.

[64] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Hold, Rinehart, and Winston, 1976.

[65] E. Lawler. A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1990.

[66] E. Lawler. Knapsack-like scheduling problems, the moore-hodgson algorithm and the "tower of sets" property. *Mathematical and Computer Modelling*, 20:91–106, 1994.

[67] Y.H. Lee, K. Bhaskaran, and M. Pinedo. A heuristic to minimize the total weighted tardiness with sequence dependent setups. *IIE Transactions*, 29:45–52, 1997.

[68] J. Lenstra, A. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:342–362, 1977.

[69] S.W. Lin and K.C. Ying. Solving single-machine total weighted tardiness problems with sequence-dependent setup times by meta-heuristics. *International Journal of Advance Manufacturing Technology*, 34:1183–1190, 2007.

[70] X. Luo and C. Chu. A branch and bound algorithm of the single machine schedule with sequence-dependent setup times for minimizing maximum tardiness. *European Journal of Operational Research*, 180:68–81, 2007.

[71] X. Luo, C. Chu, and C. Wang. Some dominance properties for single-machine tardiness problem with sequence-dependent setup. *International Journal of Production Research*, 44:3367–3378, 2006.

[72] X. Luo and F. Chu. A branch and bound algorithm of the single machine schedule with sequence dependent setup times for minimizing total tardiness. *Applied Mathematics and Computation*, 183:575–588, 2006.

[73] C.A. Mattson, A.A. Mulur, and A. Messac. Smart pareto filter: Obtaining a minimal representation of multiobjective design space. *Engineering Optimization*, 36:721–740, 2004.

[74] A. Messac, A. Ismail-Yahaya, and C.A. Mattson. The normalized normal constraint method for generating the pareto frontier. *Structural and Multidisciplinary Optimization*, 25:86–98, 2003.

[75] A. Messac and C.A. Mattson. Normal constraint method with guarantee of even representation of complete pareto frontier. *AIAA Journal*, 42:2101–2111, 2004.

[76] R. M'Hallah and R.L. Bulfin. Minimizing the weighted number of tardy jobs on a single machine with release dates. *European Journal of Operational Research*, 176:727–744, 2007.

[77] K.M. Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer Academic, 1999.

[78] K.M. Miettinen and M.M. Mäkelä. Interactive multiobjective optimization system www-nimbus on the internet. *Computers and Operations Research*, 27:709–723, 2000.

[79] J.M. Moore. One machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968.

[80] T. Morin and R. Marsten. Branch and bound strategies for dynamic programming. *Operations Research*, 24:611–627, 1976.

[81] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[82] S.C. Narula, L. Kirilov, and V. Vassilev. An interactive algorithm for solving multiple objective nonlinear programming problems. In *Proceedings of 10th International Joint Conference on Multiple Criteria*, pages 119–127, Taipei, Taiwan, 1983.

[83] S. Nash and A. Sofer (eds). *The Impact of Emerging Technologies on Computer Science and Operations Research*. Kluwer Academic, 1995.

[84] G. Nemhauser. *Introduction to Dynmaic Programming*. John Wiley Press, 1966.

[85] S.S. Panwalkar, R.A. Dudek, and M.L. Smith. Sequencing research and the industrial scheduling problem. In E. Elmaghraby, editor, *Symposium on the Theory of Scheduling and its Applications*, pages 29–38, Berlin, 1973. Spinger.

[86] L. Pèridy, E. Pinson, and D. Rivreau. Using short-term memory to minimize the weighted number of late jobs on a single machine. *European Journal of Operational Research*, 148:591–603, 2003.

[87] M.L. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, Heidelberg, 2005.

[88] C.N. Potts and L.N. VanWassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters*, 1:177–182, 1982.

[89] G.L. Ragatz. A branch and bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times. In *Proceedings of the 24th Annual Meeting of the Decision Sciences Institute*, pages 1375–1377, Baton Rouge, LA, 1993.

[90] P.A. Rubin and G.L. Ragatz. Scheduling in a sequence dependent setup environment with genetic search. *Computers and Operations Research*, 22:85–99, 1995.

[91] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[92] S. Salhi and N.M. Queen. A Hybrid Algorithm for Identifying Global and Local Minima When Optimizing Functions with Many Minima. *European Journal of Operational Research*, 155(1):51–67, 2004.

[93] D.w. Sellers. A survey of approaches to the job shop scheduling problem. In *28th Southeastern Symposium on System Theory*, pages 396–400, Baton Rouge, LA, 1996. IEEE Computer Society.

[94] M. Sevaux and S. Dauzère-Pérès. Genetic algorithms to minimize the weighted number of late jobs on a single machine. *European Journal of Operational Research*, 151(2):296–306, 2000.

[95] A. Souissi and C. Chu. Minimizing total tardiness on a single machine with sequence-dependent setup times. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, pages 1481–1485, Hague, Netherlands, 1996. IEEE Computer Society.

[96] W. Szwarc, F. Della Croce, and A. Grosso. Solution of the single-machine total tardiness problem. *Journal of Scheduling*, 2(2):55–71, 1999.

[97] W. Szwarc, A. Grosso, and F. Della Croce. Algorithmic paradoxes of the single-machine total tardiness problem. *Journal of Scheduling*, 4(2):93–104, 2001.

[98] K.C. Tan and R. Narasimhan. Minimizing Tardiness on a Single Processor with Sequence-Dependent Setup Times: a Simulated Annealing Approach. *Omega*, 25(6):619–634, 1997.

[99] K.C. Tan, R. Narasimhan, P.A. Rubin, and G.L. Ragatz. A Comparison of Four Methods for Minimizing Total Tardiness on a Single Processor with Sequence Dependent Setup Times. *Omega*, 28(3):313–326, 2000.

[100] E.L. Ulungu, J. Teghem, P. Fortemps, and D. Tuyttens. Mosa method a tool for solving multiobjective combinatorial optimization problems. *Journal of Multi-Criteria Decision Analysis*, 8:221–236, 1999.

[101] V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[102] D.A. Van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations, Ph.D Thesis*. Department of Electrical and Computer Engineering, Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, 1999.

[103] V. Venkat, S.H. Jacobson, and J.A. Stori. A post-optimality analysis algorithm for multi-objective optimization. *Computational Optimization and Applications*, 28:357–372, 2004.

[104] L.A. Wolseyn. *Integer Programming*. Wiley-Intercsience Publication, 1979.

[105] R. Zhou and E.A. Hansen. Stuctured duplicate detection in external-memory graph search. In *Proceedings of 19th National Conference on Artificial Intelligence (AAAI-04)*, pages 683–688, San Jose, CA, 2004.

[106] R. Zhou and E.A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 90–98, Monterrey, CA, 2005.

Author's Biography

Gio K. Kao was born on August 12th, 1980 to Leo and Christina Kao in the city of Hong Kong, China. At the age of 9, he moved to Palo Alto, CA in the United States. In August of 1998, Gio started his undergraduate studies in Computer Science at the University of Illinois at Urbana-Champaign. Gio recieved a B.S. in Computer Science with a minor in Mathematics in May 2002. After graduating he continued his studies and started in August of 2002 at the University of Illinois at Urbana-Champaign with his Doctorate of Philosophy in Computer Science studies under the guidance of Prof. Sheldon H. Jacobson. While doing his graduate studies on problems at the interfaces of Computer Science and Operations Research, Gio received the Sandia National Laboratories Fellowship from the Department of Energy from 2003 to 2007. Following his completion of his Ph.D., Gio will join Sandia National Laboratories in Albuquerque, NM, to further his research.