# CYCLIC BEST FIRST SEARCH IN BRANCH-AND-BOUND ALGORITHMS

BY

WENDA ZHANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Industrial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Teaching Assistant Professor Douglas King, Chair
Professor Sheldon Jacobson, Director of Research
Professor Xin Chen
Assistant Professor Lavanya Marla

# Abstract

In this dissertation, we study the application of a search strategy called cyclic best first search (CBFS) in branch-and-bound (B&B) algorithms. First, we solve a one machine scheduling problem with release and delivery times with the minimum makespan objective with a B&B algorithm using a variant of CBFS called CBFS-depth and a modified heuristic for finding feasible schedules. Second, we investigate the conditions of the search trees that may lead to CBFS-depth outperforming BFS in terms of the average number of nodes explored to prove optimality. Finally, we present a B&B algorithm using CBFS for a close-enough traveling salesman problem that demonstrates the benefit of using CBFS even if it does not improve the number of nodes explored to prove optimality. Overall, we show that using CBFS has a number of advantages to the performance of a B&B algorithm in comparison to the other search strategies given the right problems.

*To my parents and the memory of grandma.*

# Acknowledgements

I would like to thank my advisor Dr. Sheldon Jacobson, whose knowledge and experience guided me through years of struggle in finding my footing in research. This dissertation would not have happened without your help putting me on the path forward every time I went astray.

I would also like to thank Dr. Jason Sauppe, who provided many great comments on the experiments that I should run and the issues that I have not investigated fully. Most importantly, thank you for still being so patient with me even after dealing with all of my careless writing errors.

Thanks to Dr. Douglas King, Dr. Xin Chen and Dr. Lavanya Marla for sacrificing their time to sit on my committee and read this work.

Thanks to my girlfriend Yijue for always being there when I needed support. Somehow she found the time to help me achieve a healthier life style while pursuing her own Doctoral degree.

Thanks to my parents for their constant support even though I have not been able to go back home to visit them in more than 3 years. I especially thank them for their cooking recipes without which I don't think I can survive on another continent.

Finally, thanks to my cat Alfi for being a lovely companion who constantly reminds me that whatever my problems are, I should always look past them and focus on his needs.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The *branch-and-bound* (B&B) algorithm [27] has been widely-used for solving optimization problems exactly (without loss of generality, we assume the optimization problem is a minimization problem). The algorithm builds a search tree of nodes that represent regions of the solution space. Each node can be explored to either find the best solution in the corresponding solution space or generate child nodes that further divide the solution space. Moreover, the algorithm can detect regions of the solution space that do not lead to an optimal solution. These regions can then be pruned (i.e. removed from further consideration). We say a node is processed if it is either explored or pruned without exploration. When all nodes are processed, the algorithm either terminates with an optimal solution or concludes that the problem is infeasible.

The procedure used by a B&B algorithm to determine the order in which nodes in a search tree are processed is called a *search strategy*. Three strategies (and their variations) have been frequently used: *Depth First Search* (DFS), *Breadth First Search* (BrFS) and *Best First Search* (BFS). DFS selects unprocessed nodes to be processed in the reverse order that they are generated [41]. DFS is known to have low memory requirement to perform and finds complete solutions quickly because the search dives

to the bottom of the search tree quickly. However, the search strategy cannot move on from a poor region of the search tree to a promising region, where a good feasible solution may be used to prune all nodes in the poor region without exploration. As a result, DFS can perform badly when the distribution of good feasible solutions is not even across the search tree.

BrFS selects unprocessed nodes following the order that they are generated. The order of the search often leads to poor performance in B&B algorithms as complete solutions may not be generated until late in the search process which limits pruning and requires large amount of memory to store unprocessed nodes [34]. However, when the breadth of the search is needed and there are pruning rules to reduce the exploration of nodes that do not lead to good solutions, BrFS can still be applied as part of the B&B algorithm [38].

BFS stores all unprocessed nodes and tries to identify the node that is the most likely to lead to an optimal solution. This is achieved by applying a *measure-of-best function* $\mu$ to the unprocessed nodes and selecting the node with the minimum value of $\mu$. A frequently used function for minimization problems is $\mu_{LB}$, which returns a lower bound on the value of the feasible solutions that can be found within the subtree rooted at a node. In this work, if not specified otherwise, we assume BFS uses $\mu_{LB}$. Regardless of the measure-of-best function, BFS requires exponential memory with respect to the depth of the search tree to store all the unprocessed nodes in the worst case. Dechter and Pearl [17] shows that under some assumptions (notably no ties in the measure-of-best function), BFS explores the smallest number of nodes of any search strategy.

On Mixed Integer Programming (MIP) problems, a hybrid search strategy of BFS and DFS (BFS with diving) has been shown to be the most effective for a general purpose solver [1, 28]. In its basic form, BFS with diving uses DFS for the search until

some criteria are met, and the search strategy selects a new node using BFS, from which node DFS is used again. Solving child node immediately after its parent node means the solver can use the solution information from the parent node for solving the child node, which reduces the solving time of the child node. This advantage is called a "warm start" [1]. This search strategy is the basis for the search strategies used by both commercial and non-commercial MIP solvers today (i.e. CPLEX, SCIP).

On combinatorial optimization problems, where there is no general purpose solver and a complete B&B algorithm has to be created, the choice of the search strategy is decided by the authors. The "warm start" that makes BFS with diving effective in MIP problems may not be available on these problems. As a result, when there is no special structure of the search tree that can be utilized by DFS or BrFS and no severe space constraint, BFS is frequently chosen since it tends to explore fewer nodes than DFS and BrFS [7, 15].

In recent years, a new search strategy called *Cyclic Best First Search* (CBFS) has been applied to many optimization problems, including scheduling [23, 38, 39, 32, 47], graph coloring [33], traveling salesman [48], and MIP problems [34] (notably, we showed that the BFS with diving strategy can be represented as a CBFS strategy). The CBFS strategy partitions a search tree by assigning unprocessed nodes to contours using a *labeling function* $\kappa$, where each contour is a set of unprocessed nodes. Then, at each iteration of the B&B algorithm, the search strategy applies a measure-of-best function $\mu$ to a contour for node selection. Different assignments of nodes to contours can lead to significantly different search orders.

In this dissertation, we will explore the impact of CBFS on two B&B algorithms for two combinatorial optimization problems, respectively. One problem is a one machine scheduling problem and the other is a variant of the traveling salesman problem called close-enough traveling salesman problem. Additionally, we investigate the performance

of CBFS with a particular labeling function, called CBFS-depth, compared with BFS on a variety of problems.

## 1.1 Outline

In Chapter 2, we presents detailed descriptions of CBFS, including some labeling functions that lead to different behaviors of CBFS. In particular, we show the labeling functions that allow CBFS to behave as BFS, BrFS, DFS and BFS with diving. Furthermore, we present the labeling function of a variant of CBFS called CBFS-depth that partitions unprocessed nodes into contours by their depth levels in the search tree. This variant of CBFS has been successfully applied to several combinatorial optimization problems [23, 38, 39, 32].

In Chapter 3, we demonstrate the effectiveness of CBFS-depth on a B&B algorithm for a one-machine scheduling problem with release and delivery times with the minimum makespan objective, as well as a variation to this problem that requires a delay between the completion of one job and the start of another (delayed precedence constraints). We also propose an improved heuristic for finding feasible solutions in addition to the use of CBFS-depth in our B&B algorithm. Computational experiments demonstrate that our B&B algorithm lead to a substantial improvement in running time and number of iterations on the one-machine problem instances both with and without delayed precedence constraints.

Given that CBFS-depth explores fewer nodes than BFS on the one machine problem and on several other problems [23, 38, 39, 32], we are interested in the conditions of the search trees that may lead to CBFS-depth outperforming BFS in terms of the average number of nodes explored to prove optimality. In Chapter 4, we present the results of the investigation. A search tree model for B&B algorithms based on the distributions of

nodes with different lower bounds is proposed. An estimation of the expected number of nodes expanded by CBFS-depth based on the search tree model is discussed and a B&B algorithm was applied on randomly generated search trees according to the proposed model with both search strategies to study how the average number of nodes expanded is affected by the search tree. Finally, both search strategies are tested on some optimization problems to validate the observation from the generated search trees.

The conditions where CBFS-depth explores fewer nodes than BFS is not the only scenarios in which CBFS is a better choice. In Chapter 5, we present a case in which CBFS is better than BFS in terms of the memory requirement and the ability to find good feasible solutions early, given a good choice of the labeling function, even if CBFS explores more nodes than BFS. We present a B&B algorithm with CBFS for a close-enough traveling salesman problem (CETSP), which is a generalization of the Traveling Salesman Problem that requires a salesman to, instead of visiting the exact location of each customer, just get close enough to each customer. A B&B algorithm has been proposed in [15] to find the exact solution to this problem by constructing and examining partial sequences until optimality of a sequence is proven. We propose improvements to this B&B algorithm, including a variant of CBFS with a labeling function that takes advantage of the problem specific structure to group unexplored nodes to obtain good feasible solutions early, a new branching vertex selection scheme, a method to avoid unnecessary computation, a method to improve the quality of feasible solutions, and a method to reduce the space requirement of the algorithm. Numerical experiments show that the improved B&B algorithm finds good solutions faster and uses less space, which results in better performance overall than the existing algorithm.

# Chapter 2

# Preliminaries

## 2.1 B&B algorithm and cyclic best first search

A generic optimization problem that B&B algorithms are used to solve can be written as follows:

$$\min Z := f(x)$$
$$\text{s.t.} \quad x \in X$$

where the solution space $X$ is the set of all feasible solutions and $f : X \to \mathbb{R}$ is the objective function. The goal is to find an optimal solution $x^* \in X$ such that $Z^* = f(x^*) \leq f(x)$ for all $x \in X$.

The B&B algorithm maintains a search tree of nodes. Each node $l$ represents a partial solution to the original optimization problem and a region of the original problem solution space $X(l)$. The algorithm explores a node $l$ by solving a relaxation problem which has solution space $X_{RL}(l) \supseteq X(l)$. Denote the optimal objective value of the relaxation as $f^*_{RL}(l)$. $f^*_{RL}(l)$ provides a valid lower bound on the best solution in the solution space $X(l)$, and, if it belongs to the solution space $X(l)$, also a feasible

solution to the original problem.

The algorithm also keeps track of the best feasible solution for the original problem that has been found during the search, the objective value of which we denote as $\hat{Z}$. We call this solution the *incumbent* solution. If $f^*_{RL}(l) \geq \hat{Z}$ or the relaxation is infeasible, node $l$ can be removed from the search tree (pruned). If $f^*_{RL}(l) < \hat{Z}$ and the solution to the relaxation is feasible to the original problem, we set $\hat{Z} := f^*_{RL}(l)$ and update the incumbent solution.

If a node $l$ is not pruned, we branch from it by generating $b$ child nodes $l_r$ for $r = 1, 2, \ldots, b$ to further divide the region of solution space. Each child node $l_r$ is assigned an inherited lower bound $f_{LB}(l_r) := f^*_{RL}(l)$. An unprocessed node $l$ may be pruned without exploration if a new incumbent solution is found and $f_{LB}(l) \geq \hat{Z}$ is true for the new incumbent objective value $\hat{Z}$. After the algorithm has explored or pruned all nodes, the best feasible solution is proved to be optimal for the original problem. If such a solution does not exist, the original problem is infeasible.

Pseudocode for the B&B algorithm with CBFS as the search strategy is shown in Algorithm 1. The CBFS strategy maintains a set of non-empty contours $\mathcal{C}$ (Line 2, Algorithm 1) where each contour $C$ contains some unprocessed nodes. Each contour is associated with a numbered label and we denote a contour with label $i$ as $C_i$. In one iteration of the algorithm, a measure-of-best function $\mu$ is used to select an unprocessed node from a non-empty contour (Line 4, Algorithm 1). The process is similar to BFS but limited to the unprocessed nodes in one contour.

When child nodes are generated, a labeling function $\kappa$ is used to assign a label to each child node, and that node is then inserted into the contour with the same label (Lines 14-17, Algorithm 1). The non-empty contour selected in each iteration is the one with the smallest label greater than the label of the contour selected in the previous iteration, and if there is no such contour, the one with the smallest overall label is

**Algorithm 1:** Branch-and-Bound Algorithm with CBFS

```
1  Given original problem node l_root
2  Set Ẑ := +∞, i := κ(l_root), C_i := {l_root} and 𝒞 := {C_i}
3  while 𝒞 ∉ ∅ :
4      Choose l = arg min_{k∈C_i} μ(k)
5      Remove l from C_i
6      if C_i = ∅ : Set 𝒞 = 𝒞\{C_i}
7      if f_LB(l) ≥ Ẑ :  Prune l and go to line 18
8      Explore l to obtain f*_RL(l)
9      if relaxation is infeasible or f*_RL(l) ≥ Ẑ :  Prune l and go to line 18
10     if the relaxation solution is feasible to original problem :
11         Prune l and update Ẑ when necessary
12     else:
13         Generate child nodes
14         for each generated child node l_r :
15             Set f_LB(l_r) := f*_RL(l)
16             Insert l_r into contour C_{κ(l_r)}
17             if C_{κ(l_r)} ∉ 𝒞 : Insert C_{κ(l_r)} into 𝒞
18     if there exists C_j ∈ 𝒞 with j > i : Set i = min{j | C_j ∈ 𝒞 and j > i}
19     else: Set i = min{j | C_j ∈ 𝒞}
20 return Ẑ
```

selected (Lines 18-19, Algorithm 1). As a result, the search strategy cycles through all non-empty contours until there are no longer any non-empty ones.

## 2.2 Labeling functions

The labeling function $\kappa$ is critical to the CBFS strategy: it determines how nodes are assigned to contours and hence determines the order in which nodes are explored. [34] shows that given the right labeling function, CBFS can simulate the behavior of any other search strategy. Theorems 2.1, 2.2, 2.3 and 2.4 provide the labeling functions that allow CBFS to simulate the behavior of BFS, BrFS, DFS and BFS with diving proved in [34].

**Theorem 2.1.** *Using $\kappa_{BFS}(l) = 0$ for all nodes $l$, CBFS will emulate BFS.*

**Theorem 2.2.** *Let $l_j$ be the $j^{th}$ node generated by BrFS; then, using $\kappa_{BrFS}(l_j) = j$ for all $j$, CBFS will emulate BrFS.*

**Theorem 2.3.** *For any node $l$, let $\Delta(l)$ be an upper bound on the number of nodes contained in the tree rooted at $l$ that satisfies $\Delta(l) \geq 1 + \sum_{j=1}^{r} \Delta(l_j)$, where $l_1, l_2, \ldots, l_r$ are the children of $l$. For each child $l_j$ of $l$, define*

$$\kappa_{DFS}(l_j) = \kappa_{DFS}(l) + 1 + \sum_{k=1}^{j-1} \Delta(l_k).$$

*Using this labeling function for all nodes $l$, CBFS will emulate DFS.*

In the last case, bounds $\Delta_j$ can be computed if upper bounds on the branching factor and subtree depth are known. For example, given a binary branching strategy and a finite number of branching choices, then $\Delta_j = 2^u$, where $u$ is the number of remaining branching choices at $l_j$.

Finally, a labeling function is given which shows that the BFS with diving strategy which chooses the best child node for diving is a special case of CBFS:

**Theorem 2.4.** *The BFS with diving strategy can be emulated by CBFS where the root node is placed in contour 0, and contours for all other nodes are computed via the following labeling function for all nodes $l$:*

$$\kappa_{dive}(l) = \begin{cases} \kappa_{dive}(pred(l)) + 1 & \text{if } \mu(l) \geq \mu(l') \ \forall \ l' \in sib(l) \\ 0 & \text{otherwise.} \end{cases}$$

*where $pred(l)$ gives the parent of $l$ in the search tree, and $sib(l)$ gives the set of siblings of $l$, that is, nodes of $pred(l)$. Ties in $\mu$ for sibling nodes are assumed to be broken arbitrarily, or using the desired rules in a BFS with diving strategy.*

The variant of CBFS that has been shown to be successful on a number of combinatorial optimization problems is the CBFS-depth strategy. The labeling function for CBFS-depth is

$$\kappa_{depth}(l) := d(l), \tag{2.1}$$

where $d(l)$ is the depth level of node $l$ in the search tree. Therefore, CBFS-depth selects a node to explore from one depth level each iteration and selects a node from the next depth level that still has unprocessed node the next iteration.

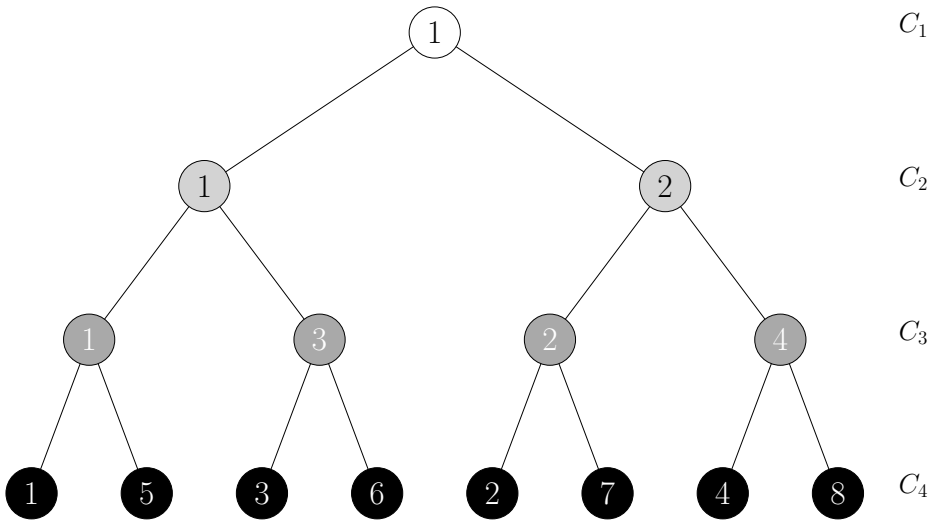



Figure 2.1: The nodes processed in each cycle

For example, starting with the root node, Figure 2.1 shows the order in which nodes are explored in each cycle by CBFS-depth on a search tree with four depth levels. The number in the nodes indicates which cycle a node is processed. The nodes are shaded based on their contours (depth levels) which are indicated at the right side of the tree. The particular nodes explored in each cycle may be different from the ones shown in the figure, but the figure shows the non-empty contours examined in each cycle.

# Chapter 3

# CBFS in the One-machine Scheduling Problem with Delayed Precedence Constraints

## 3.1 Introduction

The one-machine scheduling problem with release and delivery times with the minimum makespan objective, also known as $1|r_i, q_i|C_{max}$, can be described as follows. Assuming all parameters are integers, let $J = \{1, 2, ..., n\}$ be a set of $n$ jobs to be scheduled on a single machine without preemption. For each job $i \in J$, let $r_i$ be the release time (or head) to represent the time when job $i$ becomes available, $p_i$ be the processing time on the machine, and $q_i$ be the delivery time (or tail). A solution is given by a sequence of all the jobs in $J$, which is also called a schedule. Given a schedule of jobs $S$, let $s_i(S)$ denote the starting time of job $i$ in $S$. The completion time $c_i(S)$ of job $i$ in $S$ is given by $c_i(S) := s_i(S) + p_i$. Then the objective is to find a schedule $S$ that minimizes the makespan $C_{max}(S) := \max_{i \in J}\{c_i(S) + q_i\}$, i.e., the time by which all

jobs are delivered. For the remainder of the dissertation, when the context is clear, we simplify the notation to $s_i, c_i$ and $C_{max}$, respectively.

It has been shown that $1|r_i, q_i|C_{max}$ is NP-complete in the strong sense [20]. Numerous studies have been done to solve the problem exactly [6, 29, 26, 13] while others investigate heuristics for the problem [24, 36].

Carlier [13] formulates an efficient branch-and-bound algorithm, `Car`. The algorithm allows the presence of standard precedence constraints

$$s_i + p_i \leq s_j \quad \text{for some pairs of jobs } (i,j) \in J' \subset J \times J \text{ with } i \neq j. \tag{3.1}$$

Between jobs $i$ and $j$, constraint (3.1) means job $i$ is to precede job $j$ by setting the starting time of job $j$ to be at or after the completion time of job $i$.

Some work has been done to improve the lower bound used in `Car`. In particular, Pan and Shi [35] presents a new lower bound based on the minimum makespan of a selected subset of jobs. Gharbi and Labidi [21] presents more lower bound algorithms that improve the preemptive schedule lower bound used in `Car`. Finally, Briand et al. [11] presents an integer linear programming (ILP) formulation.

Dauzere-Peres and Lasserre [16] introduces a new type of precedence constraint for the problem, namely:

$$s_i + l_{ij} \leq s_j \quad \text{for some pairs of jobs } (i,j) \in J' \subset J \times J \text{ with } i \neq j \tag{3.2}$$

where $l_{ij} \in \mathbb{Z}_0^+$ satisfies $l_{ij} \geq p_i$. Note that (3.2) is a generalized version of (3.1), as job $j$ can only start after $l_{ij} - p_i$ time has passed since the completion time of job $i$. When that delay is 0, the two constraints are equivalent. Constraint (3.2) is referred to as the Delayed Precedence Constraint (DPC), with $1|r_i, q_i, dpc|C_{max}$ denoting this version of the one-machine problem. $1|r_i, q_i, dpc|C_{max}$ is a better representation of

the one-machine scheduling problem in a multiple machine system such as job shop scheduling [16, 7] than $1|r_i, q_i|C_{max}$. Algorithms for a multiple machine system can then take advantage of $1|r_i, q_i, dpc|C_{max}$ to find better solutions.

Unlike $1|r_i, q_i|C_{max}$, when DPC constraints are present, it is possible for a released job to not start immediately after the previous job in the schedule is processed due to the delay constraint. Balas et al. [7] shows that in such a situation, the branching scheme in `Car` does not work. The authors then present a branch-and-bound algorithm, `Bal`, that adds a new branching scheme for situations in which the scheme in `Car` does not work. `Bal` operates the same way as `Car` on instances without DPC, however.

Computational experiments in several studies have shown `Car` and `Bal` to be effective in practice. Carlier [13] uses `Car` to solve $1|r_i, q_i|C_{max}$ instances with up to 1000 jobs, and Balas et al. [7] uses `Bal` to solve $1|r_i, q_i|C_{max}$ instances of 100 jobs. All test instances are randomly generated. The number of nodes explored by `Car` and `Bal` to solve an instance of $1|r_i, q_i|C_{max}$ or $1|r_i, q_i, dpc|C_{max}$ rarely exceeds the number of jobs to be scheduled, which is not often seen on branch-and-bound algorithms where the number of nodes explored is often substantially higher. Sadykov and Lazarev [37] uses a different scheme to generate $1|r_i, q_i|C_{max}$ instances up to 300 jobs. For all problem sizes tested, the average running time of `Car` remains consistent with the results reported in Carlier [13] and Balas et al. [7] (well within 1 second in all cases).

Several other branch-and-bound algorithms, including one proposed by McMahon and Florian [29] and one using constraint programming, were tested in Sadykov and Lazarev [37] for $1|r_i, q_i|C_{max}$, but no algorithm was shown to be superior to `Car` on all problem sizes.

Pan and Shi [35] generates 500000 instances of $1|r_i, q_i|C_{max}$ using the same method as Carlier [13] and demonstrates that the 101 instances unsolved within 2 minutes by `Car` can be solved within 2 minutes by the algorithm proposed in the paper. However,

Pan and Shi [35] does not provide an average performance comparison with `Car`.

Gharbi and Labidi [21] uses the same method as in Pan and Shi [35] to generate 221 instances that are unsolved within 2 minutes by `Car`. The authors show that their best method averages 5 seconds to solve these instances, but do not report results for the average performance of their method over all generated instances.

Briand et al. [11] shows that solving the ILP formulation requires about 1/3 of the running time of `Car` on selected groups of instances. However, the average running time of solving the ILP formulation on all instances of the same size is shown to be comparable to `Car` with problem sizes up to 700 jobs, and up to 2 to 70 times slower than `Car` with problem sizes of 1000 to 3100 jobs.

To the best of our knowledge, both `Car` and `Bal` are still considered state-of-the-art algorithms for the two problems.

Due to the efficiency of `Car` and `Bal`, both variants of the one-machine scheduling problem are important building blocks in developing algorithms for more complex problems such as the job shop scheduling problem [13, 7, 35, 21], as the construction of schedules for each machine in a multiple machine problem can often be represented by $1|r_i, q_i|C_{max}$ or $1|r_i, q_i, dpc|C_{max}$. The problems are also useful in real-life industrial applications as shown in Sourirajan and Uzsoy [40], where $1|r_i, q_i, dpc|C_{max}$ is used to represent a manufacturing process. Furthermore, algorithms that solve $1|r_i, q_i|C_{max}$ also solve a scheduling problem with release dates, due dates and a maximum lateness objective, which is equivalent to $1|r_i, q_i|C_{max}$ [36].

The main contributions of this work are two improvements that works for both `Car` and `Bal`: (1) a modification of its Longest Tail Heuristic (`LTH`) which is used to generate feasible schedules, and (2) an alternative search strategy, cyclic best-first search (CBFS), which is used to select the next node to explore in the branch-and-bound search tree. When the two proposed ideas are integrated in `Car`, the computational

results show significant performance advantage in both running time and the number of nodes explored on a large number of instances; when the ideas are integrated in `Bal`, the average reduction in running time and the number of nodes explored is not as substantial but still evident, which could be attributed to the lower bound algorithm not able to take DPCs into account. In both cases, the improved algorithms have an order of magnitude improvement in running time and the number of nodes explored on selected groups of challenging instances.

The rest of the chapter is organized as follows. Because `Bal` solves $1|r_i, q_i|C_{max}$ instances the same way as `Car`, we will use `Bal` to represent both algorithms for the remainder of this chapter. Section 3.2 presents additional notation and provides an overview of `Bal`. Section 3.3 states and analyzes the heuristic used to generate a feasible schedule in `Bal`. A new heuristic is then proposed based on the discussion of the existing one. In Section 3.4 the behavior of BFS with different tie-breaking rules for node selection in the one-machine problem search tree is demonstrated and the cyclic best-first search strategy is presented as an alternative to BFS. Section 3.5 reports computational results that demonstrate the effectiveness of the algorithm. Conclusions and future research directions are outlined in Section 3.6.

## 3.2 Balas' Branch-and-Bound Algorithm

This section describes `Bal` for $1|r_i, q_i, dpc|C_{max}$ after introducing some additional notation. Let $l$ be a node in the search tree. Then, for each job $i \in J$, define:

$$\pi_l(i) := \{j \in J : j \text{ is required to precede } i \text{ in node } l\},$$
$$\sigma_l(i) := \{j \in J : j \text{ is required to follow } i \text{ in node } l\}.$$

For a node $l$ in the search tree, $\pi_l(i)$ and $\sigma_l(i)$ store the corresponding precedence constraints (including the branching decisions leading to that node). Note that if job $i$ precedes job $j$, then because of (3.2), $r_i + l_{ij} \leq r_j$ can be assumed to hold. If the current $r_j$ violates this inequality, we can set $r_j := r_i + l_{ij}$ without changing the structure of the problem. Thus, $\pi_l(i)$ and $\sigma_l(i)$ can be used to strengthen the head and tail of jobs before node $l$ is explored.

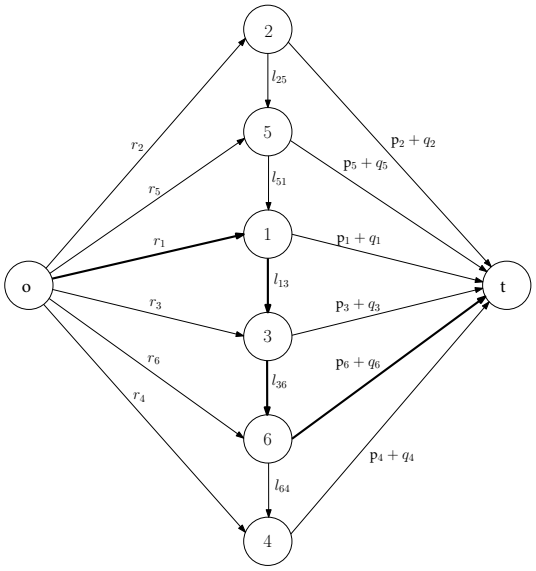



Figure 3.1: Graph Representation of a Schedule

We briefly describe the graph representation of a feasible schedule for the one-machine scheduling problem as it provides some useful insights for later discussion. Let a schedule $S$ be a feasible solution at node $l$ in the search tree. Then the solution can be represented by a directed graph $G(S) := (N, E)$. Vertex set $N$ consists of all jobs in $J$, a start $o$ and a sink $t$, such that $N = J \cup \{o\} \cup \{t\}$. The edge set $E(S)$ is defined as

$$E(S) := \{e_{oi} | i \in J\} \cup \{e_{it} | i \in J\} \cup \{e_{ij} | i, j \in J, \text{ job } i \text{ precedes job } j \text{ in } S\},$$

where $e_{oj}$ has weight $r_j$, $e_{it}$ has weight $p_i + q_i$ and $e_{ij}$ has weight $l_{ij}$. If $i \in \pi_l(j)$, then the weight of the edge $e_{ij}$ is $l_{ij}$ as in (3.2). If $i \notin \pi_l(j)$, then $l_{ij} := p_i$.

The length of the longest path (or paths, as there could be more than one) from $o$ to $t$ is the makespan $C_{max}$. We call such paths the critical paths for a schedule. Critical paths play a key role in `Bal`. Figure 3.1 shows a graph representation of a schedule with 6 jobs. Edges between vertices not scheduled next to each other are not shown to simplify the image. A critical path $\{e_{o1}, e_{13}, e_{36}, e_{6t}\}$ is marked in the graph with heavy lines.

Let $I$ be a critical path of $G(S)$, given by

$$I := \{e_{oi_1}, e_{i_1 i_2}, ..., e_{i_p t}\},$$

where $J_I := \{i_1, i_2, ..., i_p\} \subseteq J$. For $I$ to be a critical path, $s_{i_1} = r_{i_1}$ must be true. If $r_{i_1} < s_{i_1}$, some job $i_0$ scheduled before job $i_1$ must have $r_{i_0} + l_{i_0 i_1} = s_{i_1} > r_{i_1}$. Otherwise, job $i_1$ could start before $s_{i_1}$. In this case, $I' := (I \backslash e_{oi_1}) \cup \{e_{oi_0}, e_{i_0 i_1}\}$ would be a longer path than $I$, which contradicts $I$ being a critical path.

Note that if $r_{i_0} + l_{i_0 i_1} = r_{i_1}$ for some job $i_0$ scheduled before job $i_1$, then $I' := (I \backslash e_{oi_1}) \cup \{e_{oi_0}, e_{i_0 i_1}\}$ is also a critical path. For the branching scheme of `Bal` to work, it is necessary that no such $I'$ exists [13] and no additional job can be added to a critical path. Therefore, for all jobs $i$ scheduled before the first job $i_1$ in a critical path $I$, the inequality

$$r_i + l_{ii_1} < s_{i_1} \tag{3.3}$$

must hold. From here on, when discussing critical paths, we assume that they satisfy (3.3).

`Bal` is described in Algorithm 2. At each iteration, `Bal` generates a feasible schedule $S$ with `LTH` (line 6). A post processing procedure (line 7) is called to reduce the number

**Algorithm 2:** Balas' Branch-and-Bound algorithm

```
1  Let L be an empty list that stores active nodes. Insert original problem into L
2  Ĉ = ∞
3  while L ≠ ∅ :
4      Select a node l from L to explore
5      Update heads and tails of all jobs based on precedence constraints
6      Obtain a feasible schedule S by LTH
7      Postprocess schedule S
8      if there is any tail changes in postprocessing : go to 6
9      if No branching can be done from S :
10         if C(S) < Ĉ : Update current best solution by setting Ĉ = C(S)
11     else if strong branching condition is satisfied :
12         Generate l_1, l_2 with strong branching rule
13         Insert l_1, l_2 into L
14     else:
15         Generate l_1, l_2 with weak branching rule
16         Insert l_1, l_2 into L
17     Remove l from L
18 Return Ĉ
```

of critical paths that has to be examined by the branching scheme. The branching scheme (from line 9 to line 16) is then applied to $S$ to examine the critical paths and determine how new nodes are created. For additional details of Algorithm 2, see Balas et al. [7].

## 3.3 The Heuristic

We first examine the Longest Tail Heuristic (`LTH`) in line 6 of Algorithm 2. At each iteration of `LTH`, the job with the longest tail is chosen from the set of *available* jobs to be scheduled, until all jobs have been scheduled. At any iteration, a job is considered *available* if it is released and all its predecessors are already scheduled. This is repeated until all jobs have been scheduled. In this section, we first present `LTH` proposed by Balas et al. [7] and derive a bound for the solution of `LTH` on $1|r_i, q_i, dpc|C_{max}$ instances.

Then, we describe in detail a modified version of `LTH`.

### 3.3.1 Longest Tail Heuristic

The `LTH` used in `Bal` takes into account DPCs by updating the heads of all successors of a job when that job is scheduled, and the pseudo-code is presented in Algorithm 3.

**Algorithm 3:** Longest Tail Heuristic

1 Let $S := \{i \in J : i \text{ has been scheduled}\}$
2 Set $\tau := 0, S := \varnothing, C := 0, r'_i := r_i, \ \forall\, i \in J$
3 **while** $S \neq J$**:**
4  Set $Q := \{i \in J \setminus S \mid \pi(i) \subseteq S\}$
5  **if** $r'_i \leq \tau$ for some $i \in Q$**:**
6  Set $k := \arg\max_{i \in Q}\{q_i \mid r'_i \leq \tau\}$
7  **else:**
8  Set $k := \arg\max_{i \in Q}\{q_i \mid r'_i = \min_{j \in Q} r'_j\}$
9  Set $s_k := \max\{\tau, r'_k\}, S := S \cup \{k\}, \tau := s_k + p_k$
10  **for each** $j \in \sigma(k)$**:**
11  Set $r'_j := \max\{r'_j, s_k + l_{kj}\}$
12  **if** $\tau + q_k > C$**:** Set $C := \tau + q_k$
13 Return $C$

In Algorithm 3, the return value $C$ represents the makespan of the schedule obtained. Note that $r'_i$ represents the updated head of job $i$ during scheduling. This is different from line 5 in Algorithm 2, which is done before `LTH`. Note also that $\pi(i)$ and $\sigma(i)$ are used to represent the precedence constraints in current problem when the node is not specified.

Theoretical bounds on the solution of `LTH` for $1|r_i, q_i|C_{max}$ have been reported. Kise and Uno [24] provides a bound of $C/C^* \leq 2-3/(P-1)$, where $C^*$ represents the optimal makespan and $P = \sum_{i \in J} p_i$. The worst-case performance can be achieved through a two-job example: suppose $r_1 = 0, r_2 = 1, p_1 = P_C - 1, p_2 = 1, q_1 = 0$ and $q_2 = P_C - 1$, for some $P_C > 1$. Then `LTH` gives the schedule $\{1, 2\}$ with makespan $C = 2P_C - 1$, while the optimal schedule is $\{2, 1\}$ with makespan $C^* = P_C + 1$. Moreover, Potts [36]

derives several bounds that relate to the heads and tails of particular jobs, such as

$$C - C^* \leq q_{i_p}. \tag{3.4}$$

Recall that $i_p$ is the last job in a critical path $I$. However, the inequality in (3.4) is based on the condition

$$r_k \geq s_{i_1}, \text{ for all } k \in J_I, \tag{3.5}$$

which follows from the requirement on critical paths. Since critical paths must satisfy (3.3), the machine must be idle for some time before $s_{i_1}$. Thus, job $k$ would be scheduled before job $i_1$ by `LTH` if $r_k < s_{i_1}$. When DPCs are present, `LTH` can only guarantee

$$r'_k \geq s_{i_1}, \text{ for all } k \in J_I. \tag{3.6}$$

Therefore, we derive a weaker bound for `LTH` on $1|r_i, q_i, dpc|C_{max}$, given in Lemma 3.1.

**Lemma 3.1.** *For the makespan $C$ and a critical path $I$ from the solution of `LTH` on $1|r_i, q_i, dpc|C_{max}$, the following statements must be true:*

*(i) if (3.5) holds for all $k \in J_I$, then the bound (3.4) remains valid.*

*(ii) if (3.5) does not hold for all $k \in J_I$, then the bound is*

$$C - C^* \leq r_{i_1} + q_{i_p} - r_{k^*}, \tag{3.7}$$

*where $k^* = \arg\min_{k \in J_I} r_k$.*

*Proof.* A lower bound for the length of a path $I$ in $S$ is

$$h(I) = \min_{k \in J_I} r_k + \sum_{k \in J_I} p_k + \min_{k \in J_I} q_k.$$

The makespan $C$ of a path $I$ can be represented by

$$C = r_{i_1} + \sum_{k \in J_I} p_k + q_{i_p}. \tag{3.8}$$

Since $I$ is a critical path for schedule $S$, then the following inequality must hold for the optimal makespan $C^*$

$$C^* \geq h(I). \tag{3.9}$$

For case (i), inequality (3.5) holds, which implies that $r_{i_1} = \min_{k \in J_I} r_k$. Therefore, (3.9) leaves

$$C^* \geq h(I) \geq r_{i_1} + \sum_{k \in J_I} p_k. \tag{3.10}$$

Combining (3.10) with (3.8) leads to $C - C^* \leq q_{i_p}$.

For case (ii), because $k^* = \arg\min_{k \in J_I} r_k$, we can replace $r_{i_1}$ with $r_{k^*}$ in (3.10) to get

$$C^* \geq h(I) \geq r_{k^*} + \sum_{k \in J_I} p_k. \tag{3.11}$$

Combining (3.11) with (3.8) leads to $C - C^* \leq r_{i_1} + q_{i_p} - r_{k^*}$, which completes the proof. ∎

### 3.3.2 A Modified Longest Tail Heuristic

`LTH` favors jobs with long tails by selecting from all available jobs the one with the longest tail. However, the set of jobs available for scheduling can be expanded to include some jobs that have not yet been released (but have all predecessors scheduled), assuming one is willing to introduce some idle time in the sequence. Algorithm 4 presents the approach. Because the new heuristic is a modification of `LTH`, we will call it Modified Longest Tail Heuristic or `MLTH` for short.

**Algorithm 4:** Modified Longest Tail Heuristic

1 Set $\tau := 0, S := \varnothing, C := 0, r'_i := r_i,\ \forall\ i \in J$
2 # acquire initial schedule
3 **while** $S \neq J$ **:**
4  Set $Q := \{i \in J \setminus S \mid \pi(i) \subseteq S\}$
5  Set $k := \arg\max_{i \in Q}\{q_i \mid r'_i \leq \tau\}$
6  Set $l := \arg\min_{i \in Q}\{r'_i \mid r'_i > \tau\}$
7  **if** $q_l > q_k$ **:**
8  Set $\tau := r'_l$
9  **for each** $j \in \sigma(k)$ **:**
10  Set $r'_j := \max\{r'_j, \tau + l_{kj}\}$
11  Set $S := S \cup \{k\}, \tau := \tau + p_k$
12 Compute makespan $C$ and $s_i, r'_i,\ \forall\ i \in J$ of schedule $S$
13 # reschedule jobs in initial schedule
14 **if** Schedule $S$ is not valid for branching **:**
15  Reschedule jobs in $S$
16 Return $C$

`MLTH` consists of two parts. The first part, from line 2 to line 12 of Algorithm 4, constructs a feasible schedule $S$, favoring jobs with long tails; $S$ is called the *initial schedule*. The second part of Algorithm 4 (after line 13) ensures that the schedule $S$ satisfies the inequality (3.6), which we will discuss in Section 3.3.3.

In Algorithm 3, $\tau$ represents the current time (the completion time of the last scheduled job). At each iteration, $\tau$ is used to determine whether a job is available; $\tau$ gets updated after a job is scheduled. In Algorithm 4, $\tau$ again represents the current time and is used to identify the set of jobs that are available (line 5), but $\tau$ may be updated without scheduling a job. Specifically, given job $k$ as the job with the longest tail among all released jobs (line 5) and job $l$ as the next job to be released (line 6), if $q_l > q_k$, we update $\tau := r'_l$ (line 8) instead of not changing the value of $\tau$. Let $\tau_0$ be $\tau$ prior to the update. Then in the next iteration, more jobs may be considered released with a larger time counter $\tau = r'_l + p_k$, instead of $\tau = \tau_0 + p_k$. Therefore, the job with the longest tail is selected from a larger set of jobs. As jobs are scheduled, $\tau$ can be

updated repeatedly in line 8, leading to additional jobs being considered for scheduling next. Once the initial schedule is completed, we can then compute the starting time $s_i$ and update the release time $r'_i$ for each job $i \in J$ and the makespan $C$ (line 12).

### 3.3.3 Rescheduling Delayed Jobs

Algorithm 4 includes unreleased jobs when choosing the job to be scheduled next (line 5). In an iteration, an unreleased job $i_u$ can be scheduled by `MLTH` before a released job $i_k$ because $q_{i_u} > q_{i_k}$. As a result, the initial schedule we obtained in `MLTH` may contain job $i_k \in J_I$ in a critical path $I$ such that (3.6) is not satisfied. We will refer to job $i_k$ as a *delayed job*. Note that delayed jobs will not appear in `LTH`. More importantly, the branching scheme of `Bal` requires (3.6) to hold (See Balas et al. [7]). Therefore, delayed jobs must be handled before the schedule can be used in branching.

The issue is addressed by the second part of Algorithm 4. Line 14 checks for the presence of delayed jobs in the initial schedule $S$. Every delayed job $i_k \in J_I$ in the initial schedule $S$ is rescheduled as follows.

- *Step 1.* Find a pair of jobs $j_1$ and $j_2$ such that the completion time $c_{j_1}$ of job $j_1$ is strictly less than the starting time $s_{j_2}(S)$ of job $j_2$ with $r'_{i_k} < s_{j_2}(S)$. We can insert job $i_k$ between any pair of jobs $j_1$ and $j_2$. However, recall the requirement on critical paths in (3.3), which means that we can set job $j_1$ to be the job before job $i_1$ and set job $j_2 := i_1$. If job $i_1$ is the first job in schedule $S$, then set job $j_1 := o$.

- *Step 2.* Schedule job $i_k$ between jobs $j_1$ and $j_2$. Let the new schedule be $S'$. Then set the starting time $s_{i_k}(S') := s_{j_1}(S) + p_{j_1}$ if $s_{j_1}(S) + p_{j_1} \geq r'_{i_k}$ and $s_{i_k}(S') := r'_{i_k}$ otherwise.

The rescheduling steps are repeated until no delayed job can be found in any critical

path. Then the schedule would be valid for the branching process. The procedure, termed rescheduling (Algorithm 4, line 15), is outlined in Algorithm 5.

**Algorithm 5:** Rescheduling Procedure

```
1 Given initial schedule S
2 while there exists a delayed job i_k in some critical path I of S with jobs
    J_I = {i_1, i_2, ..., i_p} :
3     if job i_1 is the first job in S : Set j_1 := o
4     else: Set j_1 to be the job before i_1
5     Schedule job i_k after job j_1 and before i_1
6     Update the schedule S, the makespan C and s_i(S), r'_i, ∀ i ∈ J
7 Return C
```

Properties of the rescheduling procedure are given in Theorem 3.2.

**Theorem 3.2.** *The following statements are true for rescheduling a delayed job $i_k$ in a critical path $I$ for schedule $S$ with makespan $C$:*

*(1) there always exists a pair of jobs $j_1$ and $j_2$ between which job $i_k$ can be inserted;*

*(2) let $S'$ be the schedule after rescheduling job $i_k$ and $\lambda := s_{i_1}(S) - s_{i_k}(S')$ . Then the makespan $C'$ of $S'$ is bounded by*

$$\max\{C - p_{i_k}, C - \lambda\} \leq C' \leq \max\{C, C + p_{i_k} - \lambda\}.$$

*Proof.* (1) follows directly from the restriction on critical paths. For the remainder of this proof, we will substitute job $j_2$ with job $i_1$.

For (2), suppose $I' := \{e_{oi'_1}, e_{i'_1 i'_2}, ..., e_{i'_n t}\}$ is a critical path for the new schedule $S'$. Then the new makespan $C'$ can be represented by information on job $i'_n$,

$$C' = s_{i_n}(S') + p_{i_n} + q_{i_n}.$$

This means that we can use the change in starting time (of the last job in a critical path) from $S$ to $S'$ to determine the new makespan. The starting time change can be categorized into two cases after rescheduling job $i_k$. We will discuss these cases separately.

- *Case 1.* If $\lambda \geq p_{i_k}$, then inserting job $i_k$ between jobs $j_1$ and $i_1$ will not change the starting time of any other job before job $i_k$ in $S$. Starting time $s_i(S)$ for a job $i$ after job $i_k$ will not increase. Since no job has greater starting time in $S'$ than in $S$, we have $C' \leq C$.

  Note that for `MLTH`, $q_{i_{k-1}} \geq q_{i_k}$ must be true, where $i_{k-1}$ is the last job processed by the machine in $I$ before job $i_k$. Then

  $$s_{i_k}(S') + p_{i_k} + q_{i_k} < s_{i_{k-1}}(S') + q_{i_{k-1}} \leq C' \tag{3.12}$$

  must hold. This means that job $i_k$ cannot be the last job of a critical path in $S'$. For all other jobs, the starting time can be reduced by at most $p_{i_k}$ as the processing time of job $i_k$ in $S$ is freed by rescheduling. This implies that $C' \geq C - p_{i_k}$. Therefore, $C'$ is bounded by $C - p_{i_k} \leq C' \leq C$.

- *Case 2.* If $\lambda < p_{i_k}$, then the starting time of jobs between jobs $i_1$ and $i_k$ is increased by $p_{i_k} - \lambda$. Inequality (3.12) still holds, which means that job $i_k$ is still not the last job in a critical path in $S'$. The starting times of jobs after job $i_k$ can become larger, smaller or stay the same. In particular, the starting time can increase by $p_{i_k} - \lambda$, if a job has predecessors between job $i_1$ and $i_k$ and the starting time of that job is increased by $p_{i_k} - \lambda$ in $S'$. The starting time can also decrease by at most $\lambda$ for the same reason as in Case 1. Therefore, $C'$ is bounded by $C - \lambda \leq C' \leq C + p_{i_k} - \lambda$.

Combining these two cases gives the desired inequality. ∎

Theorem 3.2 guarantees that a delayed job can be rescheduled, and bounds the makespan of the schedule after one rescheduling procedure. Note that once no additional rescheduling is needed, the resulting schedule satisfies (3.6), and the bound in Lemma 3.1 holds. Assessing whether `MLTH` will terminate in a finite number of rescheduling steps is needed as well. Ideally, each rescheduling effort will reduce the number of delayed jobs in the schedule or at least will not create new delayed jobs. However, this cannot be guaranteed. First, job $i_k$ after rescheduling may still be a delayed job if job $i_k$ is part of a critical path $I'$ in $S'$. Second, jobs after job $i_k$ in $S$ may also become delayed jobs. For instance, let jobs $j_1$ and $j_2$ be scheduled after job $i_k$ in $S$, where $s_{j_1}(S) = r_{j_1}$ and a DPC exists between jobs $i_k$ and $j_2$ such that $s_{i_k}(S) + l_{i_k j_2} > r_{j_1}$ so job $j_2$ is scheduled after job $j_1$ in $S$. Let job $j_1$ be the first job of a critical path that includes job $j_2$. After rescheduling, if the new updated head of job $j_2$ satisfies $r'_{j_2} = s_{i_k}(S') + l_{i_k j_2} < r_{j_1}$, and jobs $j_1$ and $j_2$ are still part of a critical path that starts with job $j_1$, then job $j_2$ becomes a delayed job.

However, the rescheduling procedure will terminate in $O(n^2)$ number of reschedules as the following theorem shows.

**Theorem 3.3.** *Algorithm 5 will generate a schedule with no delayed job after at most $O(n^2)$ number of iterations.*

*Proof.* The rescheduling procedure can be viewed as placing a delayed job before some jobs with a larger head. Note that a schedule that is ordered by head $r_i$ in ascending order does not have any delayed jobs, and rescheduling procedures can be viewed as steps in a sorting algorithm to create the ascending order by the head of each job. Each rescheduling can at least swap the position between a delayed job and its immediate predecessor in a schedule, which resembles a step in the Bubble Sort algorithm. Because

the Bubble Sort algorithm has a worst case performance of $O(n^2)$, equivalently it takes at most $O(n^2)$ rescheduling steps (where $n$ is the number of jobs to schedule) to obtain a schedule with jobs sorted in ascending order by head. After this number of iterations, it is guaranteed that no delayed job exists in the schedule. ∎

## 3.4 Search Strategy

The second aspect of our algorithm that differs from `Bal` is the search strategy (line 4 in Algorithm 2). For a given search tree, the search strategy guides the branch-and-bound algorithm to the next node to explore. In `Bal`, the best-first search (BFS) strategy is used to select the node with the smallest lower bound to explore. However, we have observed that for many instances of $1|r_i, q_i|C_{max}$ and $1|r_i, q_i, dpc|C_{max}$, a large portion of the search tree shares the same lower bound which is obtained by solving the preemptive version of $1|r_i, q_i|C_{max}$ (DPC are treated as standard precedence constraints in the preemptive version).

Given a search tree where all the nodes share the same lower bound, if a last-in-first-out (LIFO) tie-breaking rule is applied with BFS and the right child is always created after the left child, then BFS selects the right child node that was just created as long as there is one, similar to depth-first search. On the other hand, if a first-in-first-out (FIFO) tie-breaking rule is applied with BFS, the search strategy explores all nodes at the same depth before any nodes at greater depth, similar to breath-first search.

In a search tree for $1|r_i, q_i|C_{max}$ or $1|r_i, q_i, dpc|C_{max}$ instance with a large number of nodes sharing a lower bound, the last-in-first-out (LIFO) tie-breaking rule allows BFS to choose nodes deep in the search tree quickly while the opposite is true for BFS with the FIFO rule. In our computational experiments, BFS with the LIFO rule has been observed to have substantially better running time and number of iteration than

BFS with the FIFO rule, which suggests that it may be beneficial to reach deep in the search tree early.

However, the intensity (the amount of time spent exploring the same region of the search space) of BFS with LIFO can lead to long searches without finding an optimal solution.

Consider then the CBFS-depth strategy. In contrast to BFS with LIFO, by choosing a node from different subsets of unexplored nodes at each iteration, the search is more diversified. At the same time, by cycling through contours based on depth levels of the search tree, CBFS-depth also retains some intensity in the search that makes BFS with LIFO effective. Therefore, we chose CBFS-depth in our branch-and-bound algorithm.

## 3.5 Computational Results

We create a branch-and-bound algorithm by integrating `MLTH` and the CBFS-depth strategy with `Bal`. The resulting algorithm, referred to as `LDepth`, can be applied to both $1|r_i, q_i|C_{max}$ and $1|r_i, q_i, dpc|C_{max}$. We also create two additional branch-and-bound algorithms for comparison: `BalDepth` which uses `LTH` and CBFS-depth, and `LBFS` which uses `MLTH` and BFS. These algorithms can be used to distinguish the individual performance benefits of using `MLTH` and CBFS-depth. Randomly generated instances are used to test the performance of these algorithms with `Bal`. In this section, we explain several implementation details of the algorithms and how instances are generated. We then compare the computational results of all algorithms.

### 3.5.1 Implementation Details and Problem Generation

Our implementation of `LDepth` and `LBFS` uses both `LTH` and `MLTH` to find a feasible schedule. At each node, the makespan of the initial schedule of `MLTH` is compared

with the makespan of the schedule generated by `LTH`. If the schedule generated by `LTH` has smaller makespan, then that schedule will be used in branching. On the other hand, when the initial schedule from `MLTH` has smaller makespan, the algorithm applies the rescheduling procedure to any delayed jobs in critical paths. The makespan after rescheduling is compared against the makespan of the schedule generated by `LTH` again. The schedule with smaller makespan is used in branching. In our computational results, `MLTH` found a smaller makespan than `LTH` more than 50% of the time (Tables A.1 and A.2). Note that the LIFO tie-breaking rule is used for search strategies in all three algorithms.

Test instances (with and without DPCs) are randomly generated using the same scheme used in Balas et al. [7]. Let $n$ be the number of jobs. Set the release time $r_i$ to be $r_i \sim \mathrm{DU}(1, r_{max})$, the processing time $p_i$ to be $p_i \sim \mathrm{DU}(1, p_{max})$, and the delivery time $q_i$ to be $q_i \sim \mathrm{DU}(1, q_{max})$, where $\mathrm{DU}(a, b)$ denotes the discrete uniform distribution from $a$ to $b$. For all instances, $p_{max} = 50$ is used. Set $r_{max} = q_{max} = \frac{1}{50}nkp_{max}$, where $k$ is a coefficient.

For generating DPCs, with probability $p$ a precedence constraint will be generated between jobs $i$ and job $j$, with delay $l_{ij} \sim \mathrm{DU}(1, \frac{1}{50}nkp_{max})$. If $l_{ij} \leq p_i$, the delay is reset with $l_{ij} = l_{ij} + p_i$, as suggested by Balas et al. [7].

The algorithms are implemented in C++ as single thread operations and all computational experiments were performed on a desktop machine with an Intel Core i5-7600K 3.8GHz quad-core processor and 8 GB of available memory. The implementation terminates if the running time reaches 3600 seconds.

Additionally, we refer to one iteration of the algorithm as each time either `LTH` or `MLTH` needs to be applied to a node, which either leads to further branching or an optimal solution for the node. Because the most time-consuming aspects of the algorithm (finding feasible solutions and branching) are included in each iteration, we

believe iterations to be a better indicator of performance than the number of nodes in the search tree as used in Balas et al. [7] and Carlier [13]. In our experiments, we observed that the number of nodes in a search tree is close to 2 times the number of iterations, as most nodes lead to branching. Therefore, the number of nodes in the search trees are not reported separately in this section. The implementation terminates if the number of iteration exceeds 100,000.

### 3.5.2 Instances without DPC

The four algorithms were first run on $1|r_i, q_i|C_{max}$. For each combination of $n$ and $k$, 1000 instances were generated. The results are reported in Tables 3.1, 3.2 and 3.3.

Table 3.1: Standard $1|r_i, q_i|C_{max}$ instances, $k = 15$

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 50$ | `Bal` | 411 | 982 | 983 | 988 | 991 | 9 | 12.2 | 5.5 |
| | `BalDepth` | 409 | 981 | 983 | 988 | 991 | 9 | 12.2 | 4.0 |
| | `LBFS` | 853 | 989 | 992 | 995 | 997 | 3 | 4.2 | 2.9 |
| | `LDepth` | 855 | 988 | 992 | 995 | 997 | 3 | 4.1 | 2.9 |
| $n = 100$ | `Bal` | 263 | 992 | 993 | 993 | 995 | 5 | 21.0 | 17.3 |
| | `BalDepth` | 263 | 992 | 993 | 993 | 995 | 5 | 21.0 | 14.5 |
| | `LBFS` | 835 | 994 | 995 | 996 | 996 | 4 | 5.0 | 4.2 |
| | `LDepth` | 835 | 994 | 995 | 996 | 996 | 4 | 5.0 | 4.7 |
| $n = 200$ | `Bal` | 208 | 992 | 995 | 996 | 996 | 4 | 30.7 | 69.1 |
| | `BalDepth` | 208 | 992 | 995 | 996 | 996 | 4 | 30.7 | 66.1 |
| | `LBFS` | 906 | 996 | 998 | 998 | 998 | 2 | 3.7 | 5.4 |
| | `LDepth` | 906 | 996 | 998 | 998 | 998 | 2 | 3.7 | 5.6 |
| $n = 500$ | `Bal` | 195 | 999 | 999 | 999 | 1000 | 0 | 33.3 | 407.5 |
| | `BalDepth` | 195 | 858 | 999 | 999 | 1000 | 0 | 33.4 | 408.5 |
| | `LBFS` | 996 | 999 | 1000 | 1000 | 1000 | 0 | 1.6 | 5.8 |
| | `LDepth` | 996 | 999 | 1000 | 1000 | 1000 | 0 | 1.6 | 6.5 |
| $n = 1000$ | `Bal` | 198 | 878 | 998 | 998 | 998 | 2 | 31.9 | 1626.1 |
| | `BalDepth` | 198 | 878 | 998 | 998 | 998 | 2 | 31.9 | 1532.7 |
| | `LBFS` | 995 | 998 | 998 | 998 | 998 | 2 | 1.7 | 13.5 |
| | `LDepth` | 995 | 998 | 998 | 998 | 998 | 2 | 1.7 | 18.5 |

Table 3.2: Standard $1|r_i, q_i|C_{max}$ instances, $k = 20$

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 50$ | `Bal` | 492 | 981 | 987 | 990 | 993 | 7 | 11.3 | 3.8 |
| | `BalDepth` | 493 | 981 | 987 | 990 | 993 | 7 | 11.3 | 3.0 |
| | `LBFS` | 985 | 992 | 995 | 996 | 998 | 2 | 2.1 | 1.6 |
| | `LDepth` | 985 | 992 | 995 | 996 | 998 | 2 | 2.1 | 1.7 |
| $n = 100$ | `Bal` | 242 | 999 | 999 | 999 | 999 | 1 | 14.9 | 11.5 |
| | `BalDepth` | 242 | 999 | 999 | 999 | 999 | 1 | 14.9 | 9.3 |
| | `LBFS` | 998 | 999 | 999 | 999 | 999 | 1 | 1.5 | 1.2 |
| | `LDepth` | 998 | 999 | 999 | 999 | 999 | 1 | 1.5 | 1.3 |
| $n = 200$ | `Bal` | 215 | 999 | 999 | 999 | 999 | 1 | 22.9 | 49.5 |
| | `BalDepth` | 215 | 999 | 999 | 999 | 999 | 1 | 22.9 | 48.7 |
| | `LBFS` | 998 | 1000 | 1000 | 1000 | 1000 | 0 | 1.4 | 1.5 |
| | `LDepth` | 998 | 1000 | 1000 | 1000 | 1000 | 0 | 1.4 | 1.9 |
| $n = 500$ | `Bal` | 207 | 906 | 999 | 999 | 999 | 1 | 29.6 | 349.1 |
| | `BalDepth` | 207 | 906 | 999 | 999 | 999 | 1 | 29.5 | 347.8 |
| | `LBFS` | 999 | 1000 | 1000 | 1000 | 1000 | 0 | 1.4 | 4.0 |
| | `LDepth` | 999 | 1000 | 1000 | 1000 | 1000 | 0 | 1.4 | 4.8 |
| $n = 1000$ | `Bal` | 207 | 869 | 999 | 999 | 999 | 1 | 32.2 | 1622.3 |
| | `BalDepth` | 207 | 869 | 999 | 999 | 999 | 1 | 32.2 | 1566.3 |
| | `LBFS` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.2 | 5.6 |
| | `LDepth` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.2 | 6.9 |

Table 3.3: Standard $1|r_i, q_i|C_{max}$ instances, $k = 25$

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 50$ | `Bal` | 930 | 994 | 997 | 998 | 999 | 1 | 5.1 | 2.0 |
| | `BalDepth` | 930 | 934 | 997 | 998 | 999 | 1 | 5.1 | 1.4 |
| | `LBFS` | 996 | 997 | 999 | 999 | 1000 | 0 | 1.5 | 1.1 |
| | `LDepth` | 996 | 997 | 999 | 999 | 1000 | 0 | 1.5 | 1.2 |
| $n = 100$ | `Bal` | 748 | 995 | 996 | 999 | 999 | 1 | 6.9 | 4.5 |
| | `BalDepth` | 748 | 995 | 996 | 999 | 999 | 1 | 6.9 | 3.2 |
| | `LBFS` | 998 | 998 | 999 | 999 | 999 | 1 | 1.5 | 1.2 |
| | `LDepth` | 998 | 998 | 999 | 999 | 999 | 1 | 1.5 | 1.2 |

(continued on next page)

Table 3.3: Standard $1|r_i, q_i|C_{max}$ instances, $k = 25$ (continued)

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 200$ | `Bal` | 627 | 1000 | 1000 | 1000 | 1000 | 0 | 7.8 | 13.8 |
| | `BalDepth` | 627 | 1000 | 1000 | 1000 | 1000 | 0 | 7.8 | 12.7 |
| | `LBFS` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.3 | 1.2 |
| | `LDepth` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.3 | 1.6 |
| $n = 500$ | `Bal` | 428 | 997 | 998 | 998 | 998 | 2 | 11.9 | 125.8 |
| | `BalDepth` | 428 | 997 | 998 | 998 | 998 | 2 | 11.9 | 126.4 |
| | `LBFS` | 999 | 999 | 999 | 999 | 999 | 1 | 1.3 | 2.8 |
| | `LDepth` | 999 | 999 | 999 | 999 | 999 | 1 | 1.3 | 3.2 |
| $n = 1000$ | `Bal` | 394 | 1000 | 1000 | 1000 | 1000 | 0 | 13.8 | 609.7 |
| | `BalDepth` | 394 | 1000 | 1000 | 1000 | 1000 | 0 | 13.7 | 573.8 |
| | `LBFS` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.2 | 5.4 |
| | `LDepth` | 1000 | 1000 | 1000 | 1000 | 1000 | 0 | 1.2 | 7.0 |

Instances are grouped by the number of iterations needed to solve them. Tables 3.1, 3.2 and 3.3 report the number of instances in each group for each combination of parameters. For example, each entry in the third column shows how many instances the corresponding algorithm solves within 10 iterations; each entry in the forth column shows how many instances are solved within 100 iterations, and so forth. The "Sol" column contains the number of instances that are solved by an algorithm within the time and iteration limits and the "Unsol" column contains the number of instances not solved by an algorithm within the given limits. Finally, the last two columns record the shifted geometric mean of the iteration counts using a shift of 10 (see Achterberg and Wunderling [2]) and the shifted geometric mean of the running time (in milliseconds) also using a shift of 10. The shifted geometric mean $\bar{x}$ for a set of numbers $x_1, x_2, ..., x_n$ is defined as

$$\bar{x} = \left( \prod_{i=1}^{n} (x_i + s) \right)^{\frac{1}{n}} - s,$$

where $s$ is the shift.

Note that if a problem is not solved (either by reaching the iteration or time limit) by an algorithm, the iteration number and running time when the algorithm stopped is used when taking the mean, which introduces a bias against the algorithm with fewer unsolved instances. Because the number of unsolved instances is small relative to the overall number of instances, the impact is not significant. If an algorithm terminates due to reaching either iteration limit or time limit on an instance, the ratio between running time and the number of iterations approximately follows the ratio between the average running time and the number of iterations of the corresponding group of instances. In addition, running times less than one millisecond are rounded up to one millisecond.

The performance of `Bal` is similar to what has been reported in previous studies [13, 37]. The majority of the instances can be solved within 100 iterations and the mean iteration numbers are low. However, instances where `Bal` required 1000 or more iterations to solve appear in all combinations of parameters.

It can be observed that `LDepth` solved more instances and also solved them in fewer iterations than `Bal`, as the majority of the instances are solved within 10 iterations. Moreover, for large $n$ ($n = 500, 1000$), the advantage of `LDepth` becomes more prominent, particularly with regard to the mean running time. However, the reduction in the number of iterations and running time can mostly be attributed to `MLTH` as there is little difference between the performance of `LBFS` and `LDepth` or between the performance of `Bal` and `BalDepth`. The reduction is substantial if comparing `LBFS` or `LDepth` with `Bal` or `BalDepth`.

For further comparison between `Bal` and `LDepth`, instances that can be solved within one second by both algorithms are discarded. For the remaining 2341 out of 15000 instances, the comparison results are reported in Table 3.4.

Table 3.4: Selected instances, Standard $1|r_i, q_i|C_{max}$

| | | LDepth | Bal | | | | |
|---|---|---|---|---|---|---|---|
| Bracket | Instances | Unsol | Unsol | Faster | Slower | Time | Iter |
| All | 2341 | 16 | 35 | 6 | 2320 | 535.4 | 25.5 |
| $[1, 3600]$ | 2326 | 1 | 20 | 6 | 2320 | 565.6 | 33.4 |
| $[10, 3600]$ | 108 | 1 | 20 | 2 | 106 | 1842.1 | 97.3 |
| $[100, 3600]$ | 10 | 1 | 8 | 1 | 9 | 28618.8 | 3487.3 |

Table 3.4 groups the instances by running time. In column 1, the label "All" denotes the set of all 2341 instances not discarded. The label "$[t, 3600]$" denotes the subset of "All" instances for which at least one of the two algorithms requires $t$ seconds or more to solve, and at least one of the two algorithms is able to solve. Column 2 reports the number of instances in each subset. Column 3 reports the number of unsolved instances in each subset for `LDepth`. Columns 4 to 8 report the results for `Bal`. Column 4 is the number of unsolved instances. Columns 5 and 6 report the number of instances that `Bal` solved faster and slower than `LDepth`, respectively. If one algorithm solves an instance and the other does not, then the former algorithm is considered to be faster than the other on this instance. Columns 7 and 8 report the ratio of the shifted geometric mean in running time and iteration between `Bal` and `LDepth`. Similar to how unsolved instances are handled in previous tables, the iteration and running time when the algorithm terminates are used for those instances.

It can be observed that `LDepth` has a clear advantage over `Bal`. Out of the 2326 instances solved by at least one of the two algorithms (the row of "$[1, 3600]$"), 2320 instances were solved by `LDepth` faster than `Bal`. In particular, `Bal` used on average 33.4 times the number of iterations and 565.6 times the running time than `LDepth` on these 2326 instances. There are also 15 instances that cannot be solved by either algorithm. In all 15 instances, both algorithms found solutions with the same makespan when they terminated.

### 3.5.3 Instances with DPC

The four algorithms were then run on $1|r_i, q_i, dpc|C_{max}$. For each combination of $n$, $k$, and $p$, 1000 instances were generated. The results are reported in Table 3.5 and 3.6. We also experimented on instances generated with $p > 0.02$ as shown in Balas et al. [7] but all algorithms performed similarly and most instances were solved within 10 iterations. Therefore, those instances are omitted.

Tables 3.5 and 3.6 are organized the same way as Tables 3.1, 3.2 and 3.3. The performance of `Bal` in most instances is similar to what has been shown in Balas et al. [7].

Table 3.5: $1|r_i, q_i, dpc|C_{max}$ instances, $k = 10$

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 50$, $p = 0.01$ | `Bal` | 122 | 951 | 959 | 965 | 966 | 34 | 28.5 | 14.8 |
| | `BalDepth` | 122 | 948 | 964 | 970 | 971 | 29 | 27.5 | 17.2 |
| | `LBFS` | 521 | 958 | 966 | 972 | 974 | 26 | 15.9 | 9.9 |
| | `LDepth` | 523 | 955 | 972 | 979 | 980 | 20 | 15.0 | 9.2 |
| $n = 50$, $p = 0.02$ | `Bal` | 457 | 984 | 985 | 986 | 989 | 11 | 13.3 | 6.0 |
| | `BalDepth` | 457 | 983 | 987 | 988 | 990 | 10 | 13.2 | 7.2 |
| | `LBFS` | 819 | 990 | 990 | 991 | 992 | 8 | 6.3 | 3.2 |
| | `LDepth` | 830 | 991 | 993 | 994 | 994 | 6 | 6.1 | 3.2 |
| $n = 100$, $p = 0.01$ | `Bal` | 265 | 996 | 996 | 996 | 996 | 4 | 14.5 | 13.9 |
| | `BalDepth` | 265 | 996 | 996 | 996 | 996 | 4 | 14.5 | 15.0 |
| | `LBFS` | 816 | 998 | 998 | 998 | 998 | 2 | 5.2 | 4.3 |
| | `LDepth` | 816 | 998 | 998 | 998 | 998 | 2 | 5.2 | 4.8 |
| $n = 100$, $p = 0.02$ | `Bal` | 942 | 1000 | 1000 | 1000 | 1000 | 0 | 4.7 | 4.2 |
| | `BalDepth` | 943 | 1000 | 1000 | 1000 | 1000 | 0 | 4.7 | 4.1 |
| | `LBFS` | 980 | 1000 | 1000 | 1000 | 1000 | 0 | 3.1 | 2.2 |
| | `LDepth` | 980 | 1000 | 1000 | 1000 | 1000 | 0 | 3.1 | 3.0 |

Table 3.6: $1|r_i, q_i, dpc|C_{max}$ instances, $k = 15$

| Parameters | Algorithm | No. of instances solved within No. Iter | | | | | Unsol | Avg. Iter | Avg. Time |
|---|---|---|---|---|---|---|---|---|---|
| | | $< 10$ | $< 100$ | $< 1000$ | $< 10000$ | Sol | | | |
| $n = 50$, $p = 0.01$ | `Bal` | 548 | 925 | 996 | 996 | 997 | 3 | 9.6 | 3.6 |
| | `BalDepth` | 548 | 994 | 996 | 996 | 997 | 3 | 9.6 | 4.5 |
| | `LBFS` | 925 | 999 | 1000 | 1000 | 1000 | 0 | 3.1 | 1.4 |
| | `LDepth` | 925 | 1000 | 1000 | 1000 | 1000 | 0 | 3.1 | 2.1 |
| $n = 50$, $p = 0.02$ | `Bal` | 917 | 996 | 997 | 998 | 999 | 1 | 5.0 | 1.9 |
| | `BalDepth` | 916 | 996 | 997 | 998 | 999 | 1 | 5.0 | 2.4 |
| | `LBFS` | 972 | 999 | 999 | 999 | 1000 | 0 | 2.7 | 1.2 |
| | `LDepth` | 972 | 999 | 999 | 999 | 1000 | 0 | 2.7 | 1.5 |
| $n = 100$, $p = 0.01$ | `Bal` | 875 | 1000 | 1000 | 1000 | 1000 | 0 | 5.2 | 4.0 |
| | `BalDepth` | 875 | 1000 | 1000 | 1000 | 1000 | 0 | 5.2 | 3.6 |
| | `LBFS` | 980 | 1000 | 1000 | 1000 | 1000 | 0 | 2.5 | 1.5 |
| | `LDepth` | 980 | 1000 | 1000 | 1000 | 1000 | 0 | 2.5 | 2.3 |
| $n = 100$, $p = 0.02$ | `Bal` | 992 | 1000 | 1000 | 1000 | 1000 | 0 | 3.1 | 2.8 |
| | `BalDepth` | 992 | 1000 | 1000 | 1000 | 1000 | 0 | 3.1 | 2.4 |
| | `LBFS` | 997 | 1000 | 1000 | 1000 | 1000 | 0 | 2.1 | 1.4 |
| | `LDepth` | 997 | 1000 | 1000 | 1000 | 1000 | 0 | 2.1 | 2.1 |

It can be observed that `LDepth` solved more instances and solved more instances quickly than `Bal`, though the difference between `LDepth` and `Bal` in mean iteration and running time is less prominent than the difference reported in Section 3.5.2. However, CBFS-depth contributes more to the overall performance gain than on instances without DPCs as some instances are solved with `BalDepth` but not `Bal`, and `LDepth` but not `LBFS` when $n = 50$ and $k = 10$.

Similar to Section 3.5.2, instances that can be solved by both `Bal` and `LDepth` within one second are discarded, which leaves 67 instances. The comparison results are reported in Table 3.7.

Table 3.7 is organized the same way as Table 3.4. Out of the 67 instances, 27 cannot be solved by either algorithm. `LDepth` solved 25 instances unsolved by `Bal`, and solved 11 instances in fewer iterations than `Bal`. `Bal` only solved 1 instance unsolved by

LDepth, and solved 3 instances in fewer iterations. Out of the 40 instances solved by at least one of the two algorithms (the row of "[1, 3600]"), Bal used on average 336.9 times the number of iterations and 301.2 times the running time than LDepth.

Table 3.7: Selected instances, $1|r_i, q_i, dpc|C_{max}$

| | | LDepth | Bal | | | | |
|---|---|---|---|---|---|---|---|
| Bracket | Instances | Unsol | Unsol | Faster | Slower | Time | Iter |
| All | 67 | 28 | 53 | 4 | 36 | 25.7 | 31.1 |
| [1, 3600] | 40 | 1 | 26 | 4 | 36 | 301.2 | 336.9 |
| [10, 3600] | 32 | 1 | 26 | 1 | 31 | 887.1 | 988.9 |
| [100, 3600] | 2 | 0 | 2 | 0 | 2 | 6884.5 | 5085.2 |

Moreover, out of the 27 instances that cannot be solved by either algorithm, LDepth found strictly better feasible solutions than Bal in 15 instances, and both algorithms found solutions with the same makespan in the other 12 instances.

Table 3.8: Selected instances, $1|r_i, q_i, dpc|C_{max}$

| | | LDepth | LBFS | | | | |
|---|---|---|---|---|---|---|---|
| Bracket | Instances | Unsol | Unsol | Faster | Slower | Time | Iter |
| All | 47 | 28 | 36 | 3 | 16 | 3.4 | 3.5 |
| [1, 3600] | 19 | 0 | 8 | 3 | 16 | 20.3 | 24.7 |

We can also compare LDepth and LBFS in the same way and the results are shown in Table 3.8, organized in the same way as in 3.7. There are a total of 8 instances that LDepth solves and LBFS does not and 8 other instances that LDepth is faster than LBFS.

## 3.6 Conclusion and Future Work

This chapter presents an improved branch-and-bound algorithm (LDepth) based on the Bal algorithm of Balas et al. [7]. MLTH is proposed based on the existing longest tail

heuristic LTH to generate feasible schedules, and a rescheduling process is presented to make sure that schedules created by MLTH are valid in the branching process. Both LTH and MLTH are used in LDepth to generate good feasible solutions. The behavior of BFS with both FIFO and LIFO tie-breaking rules on the one-machine scheduling problem is discussed and the cyclic best-first search strategy with the depth contour (CBFS-depth) is presented for LDepth to balance the diversity and intensity of the search. Computational results are reported showing that LDepth outperforms Bal on both $1|r_i, q_i|C_{max}$ and $1|r_i, q_i, dpc|C_{max}$. On a number of instances, an order of magnitude improvement is observed in terms of iteration number and running time of the algorithm.

There are a number of directions that future research on this subject can take. First, the one-machine problem discussed in this chapter appears in the Shifting Bottleneck, SB, method (Adams et al. [3]) that solves the job shop scheduling problem approximately as subproblems. Balas et al. [7] has shown that using Bal to solve $1|r_i, q_i, dpc|C_{max}$ leads to better performance of SB over other methods solving job shop scheduling problem. Future work can examine the performance of SB with LDepth.

Different methods for updating heads and tails in $1|r_i, q_i|C_{max}$ have been proposed, such as the *edge-finding* technique (see Baptiste et al. [8]). A branching scheme that allows better progress than Bal when conditions for the branching scheme in Car are not satisfied, is possible. Research can be performed to find ways to discover new methods for the one-machine scheduling problem.

Future research can introduce better lower bound methods into the algorithm. In particular, the lower bound method proposed in Pan and Shi [35] can be incorporated to further improve performance. Additionally, this lower bound could potentially be adopted to solve $1|r_i, q_i, dpc|C_{max}$ in addition to $1|r_i, q_i|C_{max}$.

# Chapter 4

# The Number of Nodes Explored by CBFS

## 4.1 Introduction

In Chapter 3, we have observed that the numbers of nodes explored to prove optimality can be smaller than BFS on the problems $1|r_i, q_i|C_{max}$ and $1|r_i, q_i, dpc|C_{max}$. In fact, in the B&B algorithms proposed using CBFS-depth [23, 38, 39, 32], the numbers of nodes explored to prove optimality is frequently smaller than that of BFS.

On the other hand, CBFS-depth is known to explore no less number of nodes than BFS on some other problems [33, 34]. To keep the analysis independent of specific problems, we will use the number of nodes explored to prove optimality to represent the performance of the search strategies. The reasons for the variation in the performance of CBFS-depth in comparison to BFS on different problems have not been studied further. In this chapter, our goal is then to investigate the performance variation by looking into the characteristics of a search tree (not specific to any particular problem) that can lead to CBFS-depth exploring fewer nodes than BFS to prove optimality.

The study of the performances of search strategies regarding the number of nodes explored in general terms has been done for DFS and BFS. Vempaty et al. [42] studies the time and space complexity of search strategies including BFS and DFS on a tree model where nodes with the same costs are in the same depth level. The cost of a node can be considered as a lower bound used as $\mu_{LB}$ by BFS and the solution value if the node is a solution. As a result, once a solution is found, only nodes in smaller depth levels have to be explored. Each node in the same depth level is assumed to have the same probability (referred to as solution density) of being a solution. Given a factor on the number of child nodes that can be generated from a parent node, as well as the solution densities (probabilities of the nodes containing the optimal solution), the authors discuss the expected number of nodes to be explored by different search strategies. Zhang and Korf [46] proposes a random search tree model with random non-negative edge costs (the cost of a node is the sum of the costs of edges on the path to reach the node) and variable branching factor to study the expected number of nodes to be explored to prove optimality of the two search strategies. In more recent years, studies on search strategies have focused on computational experiments [28, 1]. Although Bourgeois et al. [10] studies the average-case performance of BFS on a maximum independent set problem.

In terms of CBFS-depth, Morrison et al. [34] shows that the worst-case performance in terms of the number of explored nodes of CBFS-depth is significantly worse than that of BFS. Despite the worst-case performance, the authors suggest that the reason for CBFS-depth strategy exploring fewer nodes than BFS on some problems is that, by grouping nodes into different contours, it can be used to distinguish nodes with the same $\mu$ value which are not distinguished by BFS. In particular, we expand on this idea and seek to answer the following question: if the assumption of the absence of ties in measure-of-best function is not met, and BFS cannot be proven to explore

the smallest number of nodes of any search strategy, can CBFS-depth explore fewer nodes than BFS given some characteristics of the measure-of-best function values of the nodes in the search tree.

To facilitate the investigation, we first propose a search tree model that allows control over the distribution of nodes with different measure-of-best function value, and the average-case performance (the expected number of nodes explored to prove optimality) are measured on the search trees. Then the observations based on the results from the search tree model can be tested on optimization problems.

Let `B&B-CD` denote a B&B algorithm with the CBFS-depth strategy and `B&B-BFS` denote a B&B algorithm with the BFS strategy, and let the measure-of-best function return a lower bound for each node. First, we present assumptions of the B&B algorithm and the search tree model that are necessary for our analysis. Second, the search tree model is presented. Third, we study the average number of nodes explored by both `B&B-BFS` and `B&B-CD` on search trees generated using the proposed model to determine the scenarios in which `B&B-CD` explores fewer nodes than `B&B-BFS`. Finally we test the effectiveness and the limitation of the conclusions from generated search trees on several problems.

The rest of this chapter is organized as follows. In Section 4.2, assumptions necessary for our analysis on the B&B algorithm and the associated search trees are presented, and a search tree model is proposed. In Section 4.3 we propose a way to estimate the expected number of nodes explored by `B&B-CD` on the search tree model. In section 4.4, numerical results of applying `B&B-BFS` and `B&B-CD` on the search trees generated using the proposed model are presented. The impact of nodes with different lower bounds on the performance of `B&B-BFS` and `B&B-CD` is discussed and tested on several problems, including a scheduling problem and some mixed integer programming (MIP) problems. Section 4.5 concludes the chapter and briefly discusses future

research possibilities.

## 4.2 Assumptions and Search Tree Definition

Morrison et al. [34] provides a worst-case bound for the number of nodes generated by CBFS in comparison to the number of nodes explored by BFS, which we restate in Theorem 4.1.

**Theorem 4.1.** *Assume that $\mu$ is an admissible measure-of-best function (i.e. provides a valid lower bound for the region of the solution space represented by a node) for a B&B algorithm that does not use dominance relations, and assume that there does not exist any pair of nodes $l, l'$ for which $\mu(l) = \mu(l')$. Furthermore, assume that there is exactly one optimal solution in $X$, and let $l_1, l_2, \ldots, l_t$ be the sequence of nodes explored by BFS using $\mu$. Then for a fixed labeling function $\kappa$, CBFS generates children at no more than $t$ times the number of contours that are non-empty during at least one iteration of the B&B algorithm.*

Theorem 4.1 is too conservative to predict the performance of CBFS in practice. In the case of CBFS-depth, the number of non-empty contours can be as large as the maximum depth level of the search tree. Moreover, the assumption that all nodes have distinct lower bound values is rarely satisfied. Therefore, instead of analyzing the worst-case performance in general, given a set of search trees, we are interested in the average number of nodes explored to prove optimality by `B&B-BFS` and `B&B-CD`, as it may better reflect the effectiveness of the search strategies in practice.

In this section we propose a search tree model that we can use to randomly generate search trees on which to apply `B&B-BFS` and `B&B-CD`. We can also provide an estimation of the expected number of nodes explored given a search tree model as well. However, assumptions have to be made for the B&B algorithm and the search tree model to

emphases the aspect of the search tree that can lead to difference in the performance between `B&B-BFS` and `B&B-CD`.

**Assumption 4.2.** *The measure-of-best function used in CBFS-depth and BFS is* $\mu_{LB}(l) := f_{LB}(l)$ *for a node* $l$*.*

**Assumption 4.3.** *The only pruning strategy used is pruning a node* $l$ *when either* $f_{LB}(l) \geq \hat{Z}$*,* $f^*_{RL}(l) \geq \hat{Z}$*, or the relaxation solution is feasible to the original problem. We assume no relaxation is infeasible, so no pruning can be done by infeasibility.*

**Assumption 4.4.** *The branching is binary (i.e., exactly two child nodes are generated when branching).*

Assumptions 4.2, 4.3 and 4.4 restrict the complexity of the B&B algorithms to concentrate on the difference in search strategies. In particular, Assumptions 4.2 and 4.3 indicate that all nodes with $f_{LB} < Z^*$ have to be explored by any search strategy. Therefore, the number of nodes with $f_{LB} \geq Z^*$ that a search strategy has to explore determines its performance.

When Assumptions 4.2 and 4.3 hold, we can make the following observation on the number of nodes explored by `B&B-CD` and `B&B-BFS` on a search tree.

**Theorem 4.5.** *Given Assumptions 4.2 and 4.3, on a search tree where optimal solutions are found on nodes with* $f_{LB} < Z^*$*,* `B&B-BFS` *explores no more nodes than* `B&B-CD`*.*

*Proof.* Under Assumptions 4.2 and 4.3, once an optimal solution is found, there is no need to explore nodes with $f_{LB} \geq Z^*$. As a result, because `B&B-BFS` explores all nodes with $f_{LB} < Z^*$ before any nodes with $f_{LB} = Z^*$, if an optimal solution can be obtained by exploring some nodes with $f_{LB} < Z^*$, then `B&B-BFS` will not explore any node with $f_{LB} \geq Z^*$. Because nodes with $f_{LB} < Z^*$ must be explored by any search strategy, `B&B-BFS` explores no more nodes than `B&B-CD`. ∎

On the other hand, if no solutions are found by exploring nodes with $f_{LB} < Z^*$, then `B&B-BFS` may explore more nodes with $f_{LB} = Z^*$ than the number of nodes with $f_{LB} \geq Z^*$ explored by `B&B-CD` to find an optimal solution. Therefore, based on Theorem 4.5, if `B&B-CD` explores fewer nodes than `B&B-BFS` on a search tree, solutions in this search tree can only be found by exploring nodes with $f_{LB} = Z^*$.

Let $E_{CD}$ be the expected number of nodes explored to prove optimality by `B&B-CD` and $E_{BFS}$ be the expected number of nodes explored to prove optimality by `B&B-BFS`. Since we are interested in the characteristics of the search trees where $E_{CD} < E_{BFS}$ is possible, we have the following assumption on search trees.

**Assumption 4.6.** *Optimal solutions can only be found by exploring nodes with* $f_{LB} = Z^*$.

Finally, the following assumptions limit the variability in a search tree that needs to be accounted for in the comparison between search strategies.

**Assumption 4.7.** *Feasible solutions are only found in the maximum depth level.*

**Assumption 4.8.** *Suboptimal solutions cannot be used for pruning.*

If Assumptions 4.7 and 4.8 hold, then there are no suboptimal feasible solutions to the original problem to be found in any depth level except in the maximum depth level and no way to prune any nodes with $f_{LB} > Z^*$ until an optimal solution is found. As such, there is no need to distinguish between nodes whose lower bounds are greater than $Z^*$. Thus, the search tree can be viewed to have only three types of nodes based on the lower bound: nodes with $f_{LB} < Z^*$, nodes with $f_{LB} = Z^*$ and nodes with $f_{LB} > Z^*$. The assumptions also guarantee a complete binary tree, so that we know the exact number of nodes in each depth level during each cycle prior to the discovery of an optimal solution.

Note that B&B-BFS does not explore any nodes with $f_{LB} > Z^*$ and hence does not benefit from having suboptimal feasible solutions that may be used to prune some nodes with $f_{LB} > Z^*$. On the other hand, if some nodes with $f_{LB} > Z^*$ are pruned, it is less likely that B&B-CD examines a depth level that only has unprocessed nodes with $f_{LB} > Z^*$. Therefore, B&B-CD can potentially explore fewer nodes if suboptimal solutions can be used for pruning.

We are now ready to define the search tree model on which to analyze the performance of B&B-CD and B&B-BFS. We will refer to the nodes with $f_{LB} < Z^*$ as $L$-type nodes, the nodes with $f_{LB} = Z^*$ as $E$-type nodes and the nodes with $f_{LB} > Z^*$ as $G$-type nodes.

**Definition 4.9.** *Let $T(d, p, q, r)$ be a set of search trees determined by the following parameters:*

- *$d$ is the maximum depth of the search tree.*
- *$p$ is the probability of an $E$-type node producing an optimal solution in depth level $d$.*
- *$q_i := q^{i-1}$ is the probability that an $L$-type node in depth level $i = 1, 2, \ldots, d-1$ produces $f^*_{RL} < Z^*$.*
- *$r_i := r^{d-i}$ is the probability that an $L$-type or an $E$-type node in depth level $i = 1, 2, \ldots, d-1$ produces $f^*_{RL} > Z^*$.*
- *By Assumption 4.8, all nodes in depth level $d$ have either an optimal solution or a suboptimal solution. Therefore, in depth level $d$, an $E$-type node has probability $1 - p$ to produce $f^*_{RL} > Z^*$ and all $L$-type and $G$-type nodes produce $f^*_{RL} > Z^*$.*

*We also make the following comments:*

- *The root node $l_{root}$ is considered an L-type node.*
- *Since the relaxation of an L-type node in depth level $i = 1, 2, \ldots, d-1$ has three possibilities: $f^*_{RL} < Z^*$, $f^*_{RL} = Z^*$ or $f^*_{RL} > Z^*$, the choice of $q$ and $r$ are only valid when the following inequality*

$$q_i + r_i \leq 1 \tag{4.1}$$

*is satisfied for all $i = 1, 2, \ldots, d-1$.*
- *An L-type node in depth level $i$ has probability $1 - q_i - r_i$ to produce $f^*_{RL} = Z^*$. An E-type node in depth level $i$ has probability $1 - r_i$ to produce $f^*_{RL} = Z^*$.*
- *An E-type node in depth level $i$ has probability 0 to produce $f^*_{RL} < Z^*$, and a G-type node has probability 0 to produce $f^*_{RL} \leq Z^*$.*

## 4.3 Estimate $E_{CD}$

Before we apply B&B algorithms on the search trees from $T(d, p, q, r)$, we first present an estimation of $E_{CD}$ on those search trees. This also provides theoretical information on the performance of CBFS-depth when we test it against BFS in Section 4.4.1.

Before finding an optimal solution, `B&B-CD` examines each depth level that still has unprocessed nodes in each cycle. For example, in the first cycle, all depth levels have unprocessed nodes. In the second cycle, all depth levels except the first level (root node level) have unprocessed nodes. Then if no optimal solution is found, contour $i$ has unprocessed nodes in the first $2^{i-1}$ cycles.

The number of nodes explored in the $k^{th}$ cycle is $d - \lceil \log_2 k \rceil$, where $\lceil \log_2 k \rceil$ is the smallest integer greater than or equal to $\log_2 k$ and represents the number of depth

levels that no longer have any unprocessed nodes in the $k^{th}$ cycle. Let $D_j$ be the number of nodes that have been explored in the first $j$ cycles. Then $D_j$ can be written as

$$D_j := \sum_{k=1}^{j} \left(d - \lceil \log_2 k \rceil\right). \tag{4.2}$$

Let $E_{CD} := E_{CD}^{(1)} + E_{CD}^{(2)}$ where $E_{CD}^{(1)}$ and $E_{CD}^{(2)}$ are the expected number of nodes explored by `B&B-CD` to find an optimal solution and the expected number of nodes explored after an optimal solution is found, respectively. Theorem 4.10 provides an expression of $E_{CD}^{(1)}$.

**Theorem 4.10.** *Let the probability of exploring a node with an optimal solution in depth level $d$ during cycle $j$ be $\rho_j$. In particular, let $\rho_{2^{d-1}} = 1$ as the algorithm terminates after the last unprocessed node in the search tree is explored. Then*

$$E_{CD}^{(1)} = \sum_{j=1}^{2^{d-1}-1} \left( D_j \rho_j \prod_{k=1}^{j-1} (1 - \rho_k) \right). \tag{4.3}$$

*Proof.* The first $j$ cycles explore $D_j$ nodes in total. The probability that the node explored in depth level $d$ in the first cycle produces an optimal solution is $\rho_1$. The probability that the node explored in depth level $d$ in the $i^{th}$ cycle produces an optimal solution is $\rho_j \prod_{k=1}^{j-1}(1 - \rho_k)$. The probability that all the nodes in the search tree are explored is $\rho_{2^{d-1}} \prod_{k=1}^{2^{d-1}-1}(1 - \rho_k)$. Given the number of nodes explored in previous cycles in (4.2) and the probabilities of finding an optimal solution in each cycle, we can derive the expectation as (4.3). ∎

### 4.3.1 The probabilities of selecting an $E$-type node

The probability $\rho_i$ is the product of $p$ and the probability of an $E$-type node being selected in depth level $d$ cycle $j$. In particular, we want to know the probability that

level $d$ has no $L$-type nodes and at least one $E$-type node in cycle $j$, for all valid $j$. As a result, it is necessary to know the probability of each combination of different types of nodes in depth level $i$ cycle $j-1$, as well as the probability of each combination of different types of nodes in depth level $i-1$ cycle $j$ for all valid pairs of $i$ and $j$.

Let $N_L^{i,j}$, $N_E^{i,j}$ and $N_G^{i,j}$ be the number of unprocessed nodes of $L$-type, $E$-type, and $G$-type respectively, in depth level $i$ cycle $j$ when a node is to be selected from the depth level. Then the probability $\tau_{i,j,l,e,g} := P(N_L^{i,j} = l, N_E^{i,j} = e, N_G^{i,j} = g)$ is the probability that there are $l$ unprocessed $L$-type nodes, $e$ unprocessed $E$-type nodes and $g$ unprocessed $G$-type nodes in depth level $i$ cycle $j$ when a node is to be selected.

The number of nodes of each type in depth level $i$ cycle $j$ depends on the number of nodes of each type in depth level $i$ cycle $j-1$ and the child nodes added to depth level $i$ cycle $j$ after exploration of a node in cycle $j-1$. There can either be no node added to depth level $i$ cycle $j$, or two nodes that can be $L$-type, $E$-type or $G$-type. For instance, if there are two $L$-type nodes added to depth level $i$ cycle $j$, in order to have exactly $l$ unprocessed $L$-type nodes, $e$ unprocessed $E$-type nodes and $g$ unprocessed $G$-type nodes in depth level $i$ cycle $j$ means that there must be exactly $l-1$ $L$-type nodes in depth level $i$ cycle $j-1$, assuming $l-1>0$, and the number of $E$-type and $G$-type nodes unchanged (one $L$-type nodes is selected for exploration in depth level $i$ cycle $j-1$).

Conditioning on this observation, the probability $\tau_{i,j,l,e,g}$ can be written as

$$\begin{aligned}
\tau_{i,j,l,e,g} = {} & P\left(N_L^{i,j-1} = l+1, N_E^{i,j-1} = e, N_G^{i,j-1} = g | A_0^{i-1,j}\right) * P\left(A_0^{i-1,j}\right) \\
& + P\left(N_L^{i,j-1} = l-1, N_E^{i,j-1} = e, N_G^{i,j-1} = g | A_L^{i-1,j}\right) * P\left(A_L^{i-1,j}\right) \\
& + P\left(N_L^{i,j-1} = l+1, N_E^{i,j-1} = e-2, N_G^{i,j-1} = g | A_E^{i-1,j}\right) * P\left(A_E^{i-1,j}\right) \\
& + P\left(N_L^{i,j-1} = l+1, N_E^{i,j-1} = e, N_G^{i,j-1} = g-2 | A_G^{i-1,j}\right) * P\left(A_G^{i-1,j}\right),
\end{aligned} \tag{4.4}$$

where $A_0^{i-1,j}$ is the event that there are no child node is generated in depth level $i-1$ cycle $j$. Similarly, $A_L^{i-1,j}$, $A_E^{i-1,j}$ and $A_G^{i-1,j}$ are the events that the child nodes generated in depth level $i-1$ cycle $j$ is of $L$-type, $E$-type, and $G$-type, respectively. Note that the equation does not work for the edge cases with $l, e, g \leq 1$. However, the edge cases are covered when we estimate the value of $\tau_{i,j,l,e,g}$ with $\hat{\tau}_{i,j,l,e,g}$ in Appendix B.

The event of a particular combination of nodes occurring in depth level $i$ cycle $j-1$ and the event of a particular combination of nodes occurring in depth level $i-1$ cycle $j$ are not independent. For example, given a partial search tree in which 2 $L$-type nodes are generated for depth level 3 cycle 1, the nodes in depth level 2 must both be $L$-type nodes as well, which also implies that an $L$-type node is selected in depth level 2 cycle 2. Then the probability of getting two new $E$-type nodes in depth level 3 in cycle 2 is different from when we do not know which type of node is selected in depth level 2 cycle 2.

Finding the probability of getting an $E$-type node in depth level $d$ cycle $j$ requires keeping track of the probabilities of all possible combinations of nodes in previous levels, which would be impractical for trees of reasonable sizes.

If independence between combinations of different types of nodes in depth level $i$ cycle $j-1$ and depth level $i-1$ cycle $j$ is assumed, the probability of selecting an $E$-type node in depth level $i$ cycle $j$, which we refer to as $\hat{\tau}_{i,j,l,e,g}$, can be estimated with

$$\begin{aligned}\hat{\tau}_{i,j,l,e,g} = &\ \hat{\tau}_{i,j-1,l+1,e,g} * P\left(A_0^{i-1,j}\right) \\ &+ \hat{\tau}_{i,j-1,l-1,e,g} * P\left(A_L^{i-1,j}\right) \\ &+ \hat{\tau}_{i,j,l+1,e-2,g} * P\left(A_E^{i-1,j}\right) \\ &+ \hat{\tau}_{i,j,l+1,e,g-2} * P\left(A_G^{i-1,j}\right)\end{aligned} \tag{4.5}$$

The probabilities obtained in this way do not lead to the correct expectation $E_{CD}$,

but as we will show, on randomly generated search trees from $T(d, p, q, r)$ with the parameters that we have tested, the probabilities lead to close estimations of $E_{CD}$. We will use $\hat{E}_{CD}^{(1)}$ to represent this estimation.

The detail of computing $\hat{\tau}_{i,j,l,e,g}$ can be found in Appendix B. Once the values of $\hat{\tau}_{i,j,l,e,g}$ are obtained for all $i, j, l, e$ and $g$, we can estimate the probability $\rho_j$ to be

$$\rho_j = p\left(\hat{\tau}_{d,j,0,*,*} - \hat{\tau}_{d,j,0,0,g_{max}^{d,j}}\right) \tag{4.6}$$

where $g_{max}^{d,j}$ is the maximum number of unprocessed nodes in depth level $d$ cycle $j$, and $\hat{\tau}_{d,j,0,*,*}$ is the probability that $l = 0$ in depth level $d$ cycle $j$. We can compute the value of $\hat{E}_{CD}^{(1)}$ once we substitute (4.6) in Theorem 4.10.

### 4.3.2 Estimate $E_{CD}^{(2)}$

Once an optimal solution is obtained, any unprocessed $L$-type nodes have to be explored. Therefore, we need to calculate the expected number of $L$-type nodes after an optimal solution is found.

Only $L$-type nodes can have $f_{RL}^* < Z^*$ and generate $L$-type child nodes. The expected number of $L$-type nodes in the search tree, which we refer to as $E_Q$, can then be written as

$$E_Q = 1 + \sum_{i=2}^{d} 2^{i-1} \prod_{j=1}^{i-1} q_j. \tag{4.7}$$

The estimated probability of selecting an $L$-type node in depth level $i$ cycle $j$ can be written as

$$\hat{P}(N_L^{i,j} > 0) = 1 - \hat{\tau}_{i,j,0,*,*}. \tag{4.8}$$

We can estimate the expected number of $L$-type nodes selected before an optimal solution is found, by adding the probabilities $\hat{P}(N_L^{i,j} > 0)$ for all $i$ and $j$ before iteration

$\hat{E}^{(1)}_{CD}$. In particular, we can find the number of cycles $N_{cycle}$ it takes to explore $\hat{E}^{(1)}_{CD}$ nodes, and use the following formula to obtain an estimate of $E^{(2)}_{CD}$, which we refer to as $\hat{E}^{(2)}_{CD}$:

$$\hat{E}^{(2)}_{CD} = E_Q - \sum_{j=1}^{N_{cycle}} \sum_{i=\lceil \log_2 j \rceil + 1}^{d} \hat{P}(N_L^{i,j} > 0). \tag{4.9}$$

Combining the values of $\hat{E}^{(1)}_{CD}$ and $\hat{E}^{(2)}_{CD}$ lead to an estimate of the expectation $E_{CD}$, which we refer to as $\hat{E}_{CD}$.

## 4.4 Numerical Experiments

In this section, we first compare the average number of nodes explored by `B&B-BFS` and `B&B-CD` on search trees randomly generated based on the tree model in Definition 4.9. The implications of the performance difference between the two search strategies are discussed. Moreover, the one machine scheduling problem $1|r_i, q_i|C_{max}$ and some MIP problems are used to demonstrate how the analysis on the randomly generated search trees can be used to evaluate whether `B&B-CD` may be able to outperform `B&B-BFS`.

An important aspect of the search strategies BFS and CBFS-depth that is not specified yet is the tie-breaking rules. On one hand, the tie-breaking rule does not affect the performance of CBFS-depth on the random search trees we defined as the probabilities are the same in each depth level where tie-breaking is needed. On the other hand, the tie-breaking rule can affect the number of $E$-type nodes that BFS has to explore based on Assumptions 4.3 and 4.6.

It can be observed that, given a search tree of nodes with the same $f_{LB}$, BFS with First-In-First-Out (FIFO) tie breaking rule behaves like breadth first search, and BFS with Last-In-First-Out (LIFO) tie breaking rule behaves like DFS. As a result, both rules have the advantages and disadvantages of the search strategies they resemble. For

BFS to not be affected by these potential extreme behaviors, we choose the arbitrary tie-breaking rule for `B&B-BFS` in our experiments.

All numerical experiments were done on a Desktop with an Intel Core i7 3.6GHz quad core processor and 16GB of RAM; all implementations were written in C++.

### 4.4.1 Randomly generated search trees

We first applied `B&B-BFS` and `B&B-CD` on randomly generated search trees from $T(d, p, q, r)$.

For each node in depth level $i < d$ of the search tree, its relaxation solution $f^*_{RL}$ was determined with the following rules: with probability $q_i$, let $f^*_{RL} := Z^* - 1$; with probability $r_i$, let $f^*_{RL} := Z^* + 1$; otherwise, let $f^*_{RL} := Z^*$. In the maximum depth level of the search tree, if a node does not produce an optimal solution, then it has a suboptimal solution with objective value $Z^* + 2$ so that finding such a solution does not allow for pruning other nodes from the search tree, as required by Assumption 4.8.

Let $d = 12$ and $p = 0.05$. We test pairs of $q$ and $r$ in the range between 0.1 and 1 that satisfy inequality (4.1). For each valid pair of $q$ and $r$, 1000 instances are generated. Because the number of $E$-type nodes in the maximum depth level is limited, in rare occasions there can be no node in the generated search with an optimal solution, in which case all nodes in the search will have to be explored before the algorithm terminates. We will indicate the combinations of parameters in which this occurs.

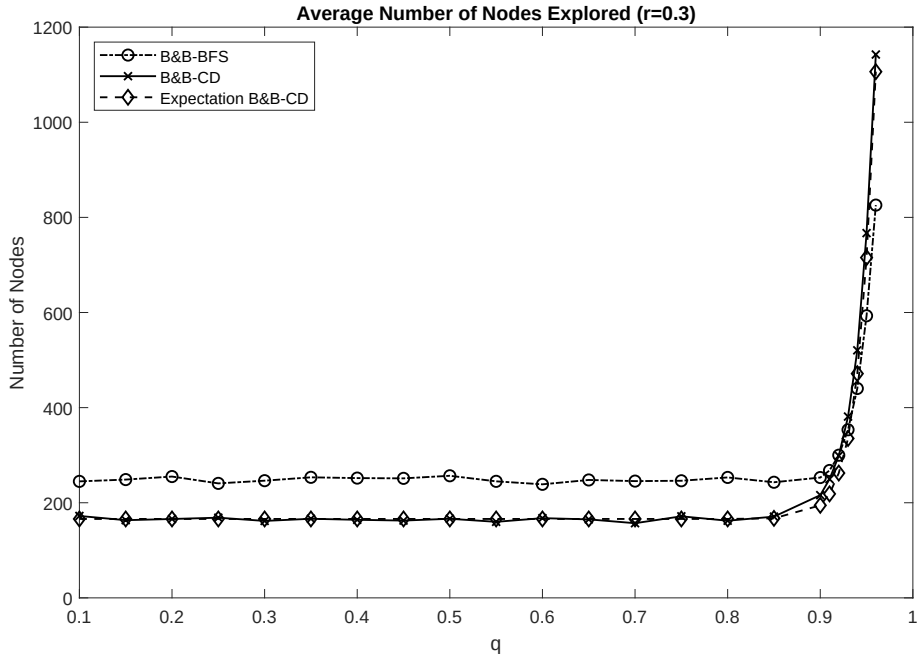



Figure 4.1: Comparison of `B&B-BFS`, `B&B-CD` and CBFS-depth expectation by $q$

Figure 4.1 shows the average number of nodes explored by `B&B-BFS` and `B&B-CD` as well as the estimation $\hat{E}_{CD}$ as $q$ varies with $r = 0.3$ fixed. When $q$ is small, `B&B-CD` is consistently better than `B&B-BFS`. As $q$ becomes larger, `B&B-CD` eventually explores more nodes on average than `B&B-BFS`.

On the other hand, the estimation never deviates far from the average results from simulation which shows that the probabilities we calculated should be close to the true probabilities on a randomly generated search tree $t \in T(d, p, q, r)$. In Appendix C, we present a comparison of the computed probability of selecting an $E$-type node in depth level $d$ cycle $j$ against the proportion of the search trees in the simulation in which an $E$-type node is explored in depth level $d$ cycle $j$. The results indicate that the estimation does not stray away from the proportions obtained in the simulation.

Figure 4.2 shows how the probabilities of selecting an $E$-type node in depth level $d$

for each cycle (as we calculated) changes as $q$ changes. As $q$ grows, `B&B-CD` has decreasing probabilities of selecting an $E$-type node in early cycles. Therefore, `B&B-CD` needs to take additional cycles to find an optimal solution. It then becomes a disadvantage that each cycle has to go through all depth levels that are not empty yet as the only remaining nodes in some levels may be of $E$-type or $G$-type. It explains the reason that `B&B-CD` explores more nodes than `B&B-BFS` when $q$ is large.

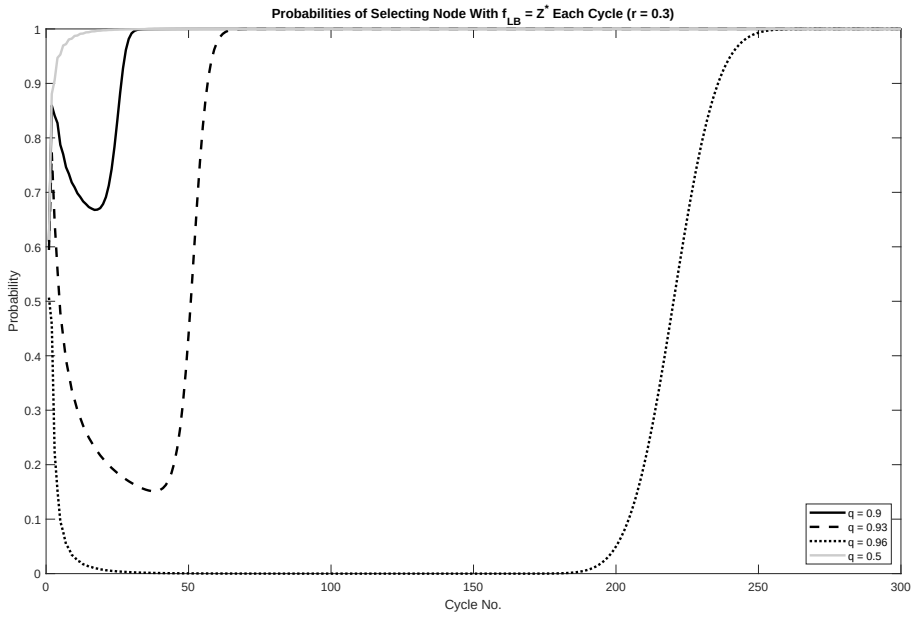



Figure 4.2: Probabilities of selecting $E$-type nodes in level $d$ during each cycle ($r = 0.3$)

Figure 4.3 shows the average number of nodes explored by `B&B-BFS` and `B&B-CD` as well as the estimation $\hat{E}_{CD}$ when $r$ changes and $q = 0.5$. Similar to the previous comparison, `B&B-CD` consistently explores fewer nodes than `B&B-BFS` when $r$ is small. The sharp increase in average number of nodes explored when $r = 0.8$ can be attributed to many search trees not having any node with an optimal solution of $Z^*$ which leads to all 4095 nodes in the search trees explored. When taking the average of all instances where an optimal solution of $Z^*$ is found, we find that the average number of nodes

explored by B&B-BFS is 190.2 while the average number of nodes explored by B&B-CD is 225.0. It is consistent with our previous discussion that, with large $r$, B&B-BFS explores fewer nodes than B&B-CD. B&B-CD cycles through all non-empty depth levels and is more likely to explore many $G$-type nodes with large $r$.

On the other hand, the estimation remains accurate and the probabilities we calculated should be close to the true probabilities on any particular search tree $t \in T(d, p, q, r)$.

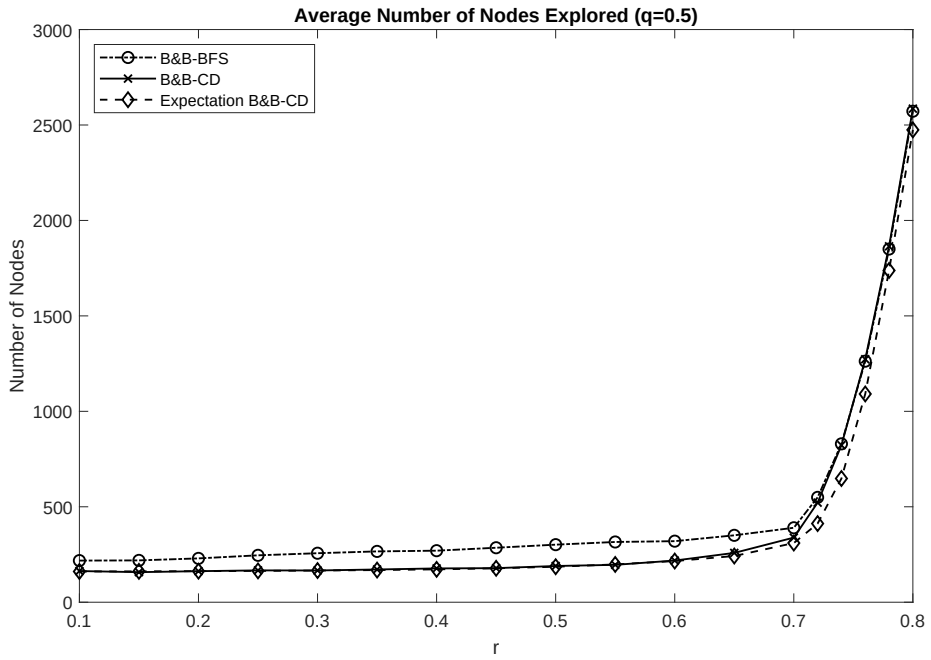



Figure 4.3: Comparison of B&B-BFS, B&B-CD and CBFS-depth expectation by $r$

Figure 4.4 shows how the probabilities of selecting an $E$-type node in depth level $d$ each cycle (as we calculated) changes as $r$ changes. The increase of $r$ leads to smaller probabilities of selecting an $E$-type node in all cycles. As a result, $G$-type nodes are more likely to be selected in early cycles with large $r$, which also makes B&B-CD less effective than B&B-BFS when $r$ is small.

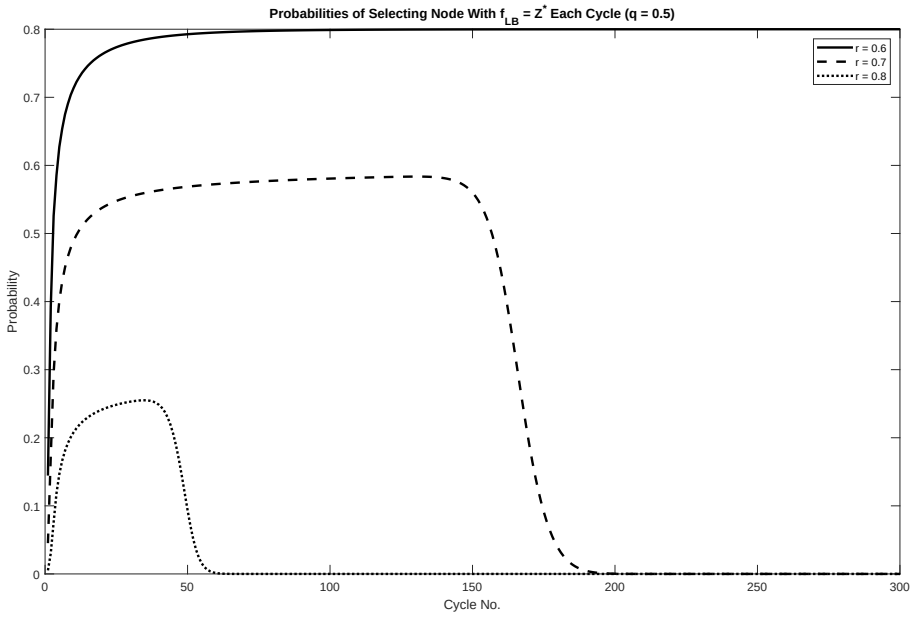


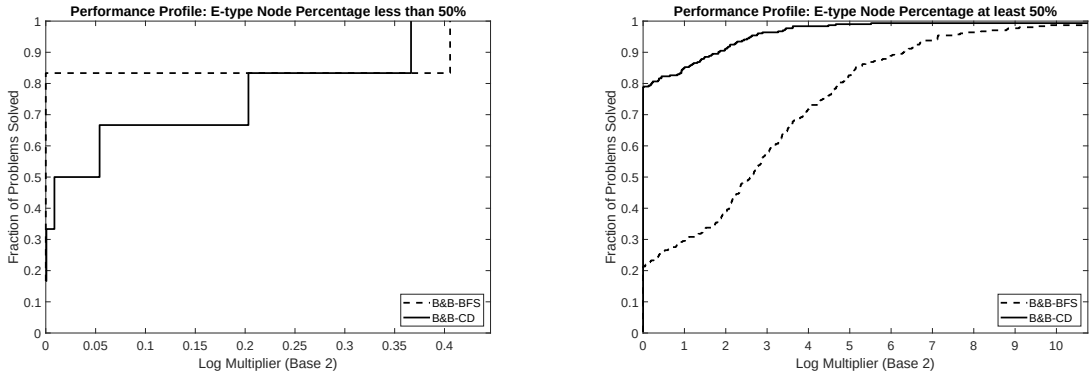

Figure 4.4: Probabilities of selecting $E$-type nodes in level $d$ during each cycle ($q = 0.5$)

We choose to show the results of the search strategies with $r = 0.3$ and $q = 0.5$ as they are representative of the behavior we have observed in other pairs of $q$ and $r$. In summary, when the number of $E$-type nodes is high in the regions of the search tree where optimal solutions can be found, `B&B-CD` may explore fewer nodes than `B&B-BFS`.

### 4.4.2 One machine scheduling problem

Chapter 3 shows that the search trees of instances of $1|r_i, q_i, dpc|C_{max}$ often contain a large number of $E$-type nodes, and uses CBFS-depth as the search strategy for the proposed algorithm. For convenience, in the context of the one machine scheduling problem discussed in this chapter, we will use `B&B-CD` to represent the B&B algorithm `LDepth`, and `B&B-BFS` to represent `LBFS`. With the algorithms, we can investigate the search trees and see if the observations from generated search trees remain useful.

We generate 2000 instances of the problem (details on problem generation can be




(a) $E$-type node percentage less than 50% (b) $E$-type node percentage more than 50%

Figure 4.5: Performance profiles from the one machine scheduling problems

found in Section 3.5.1) where 1000 instances are generated with $n = 50, k = 10$ and $p = 0.01$, and 1000 instances are generated with $n = 50, k = 10$ and $p = 0.02$. The arbitrary tie-breaking rule is used following our previous experiments. We set the time limit to 3600 seconds and number of nodes explored limit to 100000. Since most of the instances are solved with a few nodes explored, to better demonstrate the difference in performance between the search strategies, we only use instances that take at least 0.1 second to solve and are solved by at least one of the two algorithms, which results in 311 instances.

Figure 4.5 shows the performance profile [18] of `B&B-CD` and `B&B-BFS` on the 311 instances using the number of nodes explored as metric. A performance profile plots the fraction of instances ($y$-axis) solved by `B&B-BFS` and `B&B-CD` within a given multiple ($x$-axis) of the fastest one. A higher line to the left side of the plot indicates the ability to solve more instances faster (better efficiency) and a higher line to the right side indicates the ability to solve more instances (robustness). For all profiles presented, the log of the multiplier is used on the $x$-axis.

The instances are divided into two groups based on the percentage of $E$-type nodes in the generated search tree. The percentage is computed using the generated search

trees from B&B-BFS, since we are interested to see if the percentage of $E$-type nodes can be used as an indicator for the performance of B&B-CD.

First, there are 6 instances on which less than 50% of the generated nodes are $E$-type nodes in Figure 4.5a. On these instances B&B-BFS explores fewer nodes than B&B-CD in 5 of them. Based on the performance profile, B&B-CD is not as efficient as B&B-BFS on these instances. On the other hand, Figure 4.5b shows the performance profile of the algorithms on 305 instances on which at least 50% of the generated nodes are $E$-type nodes. There are 240 instances where B&B-CD explores fewer nodes than B&B-BFS and it can be observed that B&B-CD performs substantially better than B&B-BFS in terms of number of nodes explored on these instances. The results comply with our observation on the generated search trees, that B&B-CD is more likely to outperform B&B-BFS when the proportion of $E$-type nodes is high.

We have also experimented with our estimated expectation. The assumptions for the generated search trees are difficult to satisfy as there is no obvious maximum depth level and there are suboptimal solutions. But it would be interesting to see if, given just the information about the search trees generated with B&B-BFS, there exists a realistic scenario where the estimated expected number of nodes explored by B&B-CD is smaller than the average number of nodes explored by B&B-BFS.

We took all instances in which more than 50% of the nodes generated by B&B-BFS are $E$-type nodes and the average depth level is 7, the average proportion of $L$-type nodes is 0.003 and the average proportion of $G$-type nodes is 0.05. We then set $d = 7$, $q_i = 0.003$ and $r_i = 0.05$ for $i = 2, 3, \ldots, 6$ while setting $q_1 = 1$ and $r_1 = 0$.

Figure 4.6 shows the change of the estimated expectation as the probability $p$ changes as well as the actual average number of nodes explored by both search strategies. The solid line represents the average number of nodes explored by B&B-BFS over all selected instances and the dash line represents the average number of nodes explored

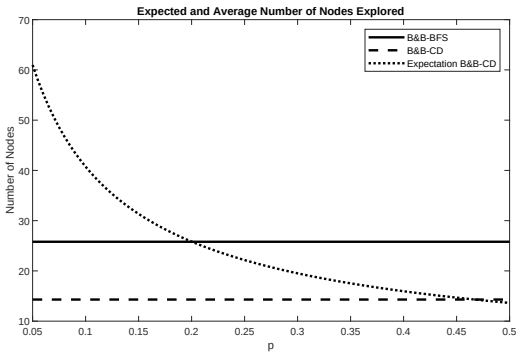



Figure 4.6: Expected and Average Number of Nodes Explored

by `B&B-CD`. The dotted line represents the estimated expectation. Note that we see the expectation become smaller than the average number of nodes explored by `B&B-BFS` for $p \geq 0.2$. It is an indication that a reasonable $p$ value, `B&B-CD` can explore fewer nodes than `B&B-BFS` on average, and we should try using `B&B-CD` on this problem.

### 4.4.3 Mixed integer programming problems

We also tested both BFS and CBFS-depth on 87 MIP problems from the MIPLIB 2010 library [25] "benchmark" instances with CPLEX 12.8 in a similar way as in Morrison et al. [34]. In the context of solving MIP problems, we use `B&B-CD` to represent CPLEX with CBFS-depth and `B&B-BFS` to represent CPLEX with BFS. Search strategies were implemented using callback functions, which disabled the parallel search and dynamic search options; to further isolate the effect of search strategies, advanced start methods and cut generation were disabled. A 3600 seconds time limit is imposed on all problems. However, given the complex nature of the solving process in CPLEX, it remains difficult to represent the impact of the proportion of $E$-type nodes alone on the performance of `B&B-CD` and `B&B-BFS`. Therefore, we only use one instance from the "benchmark" instances to demonstrate that impact.

In particular, on problem "lectsched-4-obj", B&B-BFS fails to find the optimal solution within 1 hour while B&B-CD is able to solve the problem in 7.71 seconds. With the optimal solution we found by using B&B-CD, we examined the search tree after 1-hour search of B&B-BFS. A total of 880258 nodes are created, with 4 $L$-type nodes, 832440 $E$-type nodes and 47814 $G$-type nodes. This supports previous observations that B&B-CD may be a better choice than B&B-BFS on a search tree with a large number of $E$-type nodes.

It should be noted that there are several other problems where B&B-CD outperforms B&B-BFS, but there are only a few $E$-type nodes in those search trees created by B&B-BFS. One advantage of using CBFS-depth that is not discussed in this chapter is that by exploring nodes deep in a search tree early, B&B-CD may find a feasible solution earlier than B&B-BFS, which may spend a long time exploring the nodes in the middle of a search tree [34]. Finding feasible solutions early may help some pruning rules in CPLEX to prune additional nodes including some $L$-type nodes. We will discuss this advantage of CBFS more in Chapter 5.

### 4.4.4 Discussions

Given that BFS is used in a B&B algorithm, the observations in this section is useful to determine if an alternative choice, CBFS-depth, may result in fewer number of nodes explored.

The assumptions we have for the algorithm and search trees in order to analyze the performance of both search strategies on the generated search trees are difficult to satisfy in practice. However, the experiments provide valuable information regarding the relation between the distribution of different types of nodes in search trees and the performance of the search strategies. As our experiments with the one machine scheduling problem show, there is a clear distinction in the distribution of different

types of nodes in the search trees between the instances where B&B-CD outperforms B&B-BFS and the instances where B&B-CD does not. The distribution of $E$-type nodes in the search trees serves as an indicator of whether B&B-CD has potential to outperform B&B-BFS.

Similar to the one machine scheduling problem, we have observed that the search trees generated by the same B&B algorithm on many instances of the same problem group, such as the Simple Assembly Line Balancing problem, can have similar distribution of $E$-type nodes in the search trees that B&B-CD can take advantage of [38].

Therefore, given a problem instance where B&B-BFS proves optimality but does not perform well, we can check if the optimal solutions come from $E$-type nodes and check if the portion of $E$-type nodes in the search tree is large. When the answers to both questions are positive, we can experiment with B&B-CD on the instance, or similar instances of the same problem group.

On the other hand, on a problem instance where B&B-BFS cannot prove optimality, we can still experiment with B&B-CD if a large portion of the search tree has the same $f_{LB}$. If the $f_{LB}$ for the instance is tight (i.e., all the nodes with the same $f_{LB}$ are $E$-type nodes) and optimal solutions can only be produced from $E$-type nodes, B&B-CD can still outperform B&B-BFS.

## 4.5 Conclusion And Future Work

In this chapter, a search tree model for B&B algorithms based on the distribution of nodes with different lower bounds is proposed so that we can use randomly generated search trees to study the performance of search strategies CBFS-depth and BFS in terms of the number of nodes explored to prove optimality. We first showed that, without sophisticated pruning rules beyond the lower bound, optimal solutions must

come from nodes with $f_{LB} = Z^*$ in order for CBFS-depth to have an opportunity to explore fewer nodes than BFS. An estimation of the expected number of nodes explored by CBFS-depth based on the parameters of the search tree is proposed. Generated search trees and several problems are used to demonstrate the search tree conditions under which CBFS-depth is a better choice than BFS. In particular, the results show that one key to the success of CBFS-depth is the large number of nodes with $f_{LB} = Z^*$ on the levels where an optimal solution may exist.

The analysis in this chapter aims to inform future research work on the performance of search strategies like CBFS-depth. We have discussed the performance of `B&B-CD` and `B&B-BFS` under the assumptions presented in Section 4.2. When some of the assumptions are relaxed, more research can be done to determine how the performance of the search strategies change. With the numerical experiments in the previous section, we have already observed that `B&B-CD` may still outperform `B&B-BFS` when the distribution of different types of nodes in the search trees are not in favor of CBFS-depth. We can examine the impacts of having additional pruning rules, having suboptimal solutions, or having multiple depth levels in which optimal solutions are possible. These analyses will be valuable to guide the selection of search strategies or the construction of new search strategies in B&B algorithms for particular problems.

# Chapter 5

# CBFS in the Close-enough Traveling Salesman Problem

## 5.1 Introduction

The traveling salesman problem (TSP) is a well known and well studied optimization problem. In its basic form, TSP finds a shortest Hamiltonian cycle (a closed loop that visits all the vertices exactly once) in a complete and undirected graph $G = (V, E)$ where $V$ represents a set of vertices and $E$ the set of edges connecting the vertices [5]. The length of each edge $e_{ij}$ where $i, j \in V$ is given by some distance function $dist : V \times V \to \mathbb{R}^+$.

The close-enough traveling salesman problem (CETSP) is a generalization of the traveling salesman problem (TSP) where, instead of visiting each vertex $v \in V$, the salesman must visit a specific region that contains a vertex $v \in V$. The region that has to be visited for a vertex $v$ is called a covering region $N(v)$ of that vertex, and a covering region is considered visited (covered) if at least one point in the region is visited. Given a depot $v_0 \in V$, the objective of the problem is to find a tour with the

shortest distance traveled that visits the covering regions of all vertices that starts and ends at the depot. The covering region of a vertex $v$ is commonly defined as a circular disk with radius $r$ centered at $v$, and we will define covering region the same way in this work. When the radii of all covering regions are zero, the problem is reduced to a TSP. However, when the radii are not zero, the exact point in each covering region that is visited by a tour has to be determined in addition to the sequence that the covering regions are visited, which makes CETSP difficult to solve.

CETSP often arises when visiting a customer (vertex) does not necessarily mean visiting the exact location of that customer. For example, a drone that tries to visit a set of sensors to read the data only need to get close enough to each sensor to read its data remotely.

A number of heuristics to solve CETSP have been proposed. In Gulczynski et al. [22] and Dong et al. [19], a set of supernodes is found so that each covering region contains at least one supernode, which is solved as a TSP to generate a feasible solution to CETSP. The solution to TSP is then improved while maintaining feasibility to CETSP. Yuan et al. [45] proposes a two-phase algorithm to decompose CETSP into a combinatorial problem (which finds conventional TSP solution for the vertices without the covering regions) and a continuous optimization problem (which improves the tour considering the covering regions). In Mennell [30] and Mennell et al. [31], Steiner zones (overlaps of covering regions) are used as a graph reduction method to reduce the number of regions to consider. The authors also introduced a heuristic that first reduces CETSP to a Generalized TSP (GTSP) by discretizing the covering regions and then solve the GTSP with a genetic algorithm. Yang et al. [44] solves CETSP with arbitrary neighborhoods with an algorithm based on particle swarm optimization and a genetic algorithm. Finally, Wang et al. [43] proposes a heuristic that solves a set covering problem to find an initial set of Steiner zones that can be used to create a feasible

solution to CETSP. A variable neighborhood search scheme is then used to improve the feasible solution and select a different set of Steiner zones to solve for a new feasible solution of CETSP. Using this scheme, the author manages to improve the quality of feasible solutions on many benchmark problems.

In terms of exact approaches, there have only been a few. Behdani and Smith [9] proposes a few mixed integer programming formulations based on different discretization schemes of the covering regions. The most efficient formulation is a two-stage one that first identifies the order in which the covering regions are visited and then optimizes the corresponding tour. The authors developed 240 test instances with between 7 to 21 vertices, but were unable to prove optimality on them. A different discretization scheme is proposed in Carrabs et al. [14] which results in tighter upper and lower bounds for the same instances.

Coutinho et al. [15] proposes a branch-and-bound (B&B) algorithm to solve this problem, which we will refer to as `Cout`. In this algorithm, each node in the search tree represents a subset of vertices in a fixed sequence and branching is done by inserting a not-yet-sequenced vertex into all possible places of the existing sequence. Given a fixed sequence, a second-order cone program (SOCP) is solved to produce the optimal tour length for that sequence. Two branching variable selection schemes are proposed for instances with constant and arbitrary radii on all covering regions, respectively. The authors suggest Best First Search (BFS) as the search strategy for node selection as it outperforms other search strategies (Depth First Search and Breadth First Search) they have examined. In particular, BFS selects from all unexplored nodes the one with the smallest lower bound. `Cout` was able to solve all instances developed in Behdani and Smith [9] as well as some large instances with up to 1000 vertices introduced in Mennell [30] depending on the covering region radii.

In this chapter, we present an improved version of `Cout`. In particular, we propose

a new search strategy derived from the cyclic best first search (CBFS) framework [34] which we will refer to as CBFS-CV that takes advantage of the information obtained in the search. Moreover, we propose a branching vertex selection scheme that unifies the two used in `Cout`. Additional improvement measures for the efficiency of `Cout` are also discussed, including avoiding redundant computation, improving the quality of obtained feasible solutions, reducing the space needed for storing unexplored nodes as well as improving the implementation of the algorithm. The result of all improvement measures is demonstrated with numerical experiments, where solved instances are solved with less running time and space requirement, and new best known feasible solutions are found on several instances not yet solved.

The chapter is organized as follows. Section 5.2 describes CETSP and `Cout` in more details. Section 5.3 discusses the new search strategy used to replace BFS in the B&B algorithm. Further improvement measures are presented in Section 5.4. In Section 5.5, numerical results are presented to show the benefit of the improvement. In Section 5.6, conclusion and future research directions are discussed.

## 5.2 Preliminaries

### 5.2.1 Problem description

In this chapter, we define the CETSP in two-dimensional (2D) or three-dimensional (3D) Euclidean space and the Euclidean distance is used as the distance function. Let $V = \{v_0, v_1, ..., v_n\}$ be a set of vertices with $v_0$ being the depot. Each vertex $v_i$ for $i \neq 0$ has a disk (ball if defined in 3D space) centered at $v_i$ with radius $r_i$. The objective of the problem is to construct the shortest tour that starts and ends at $v_0$ and visits all covering regions. The shortest tour is obtained by finding the sequence of points

$P = (p_0, p_{k_1}, p_{k_2}..., p_{k_n})$ that forms the tour where each point $p_{k_i}$ in $P$ is a point in the covering region $N(v_{k_i})$ of vertex $v_{k_i} \in V$. We call the points in $P$ turning points. The sequence of points $P$ also corresponds to a sequence $S = (v_0, v_{k_1}, v_{k_2}, ..., v_{k_n})$ that represents an ordering of the vertices in $V$.

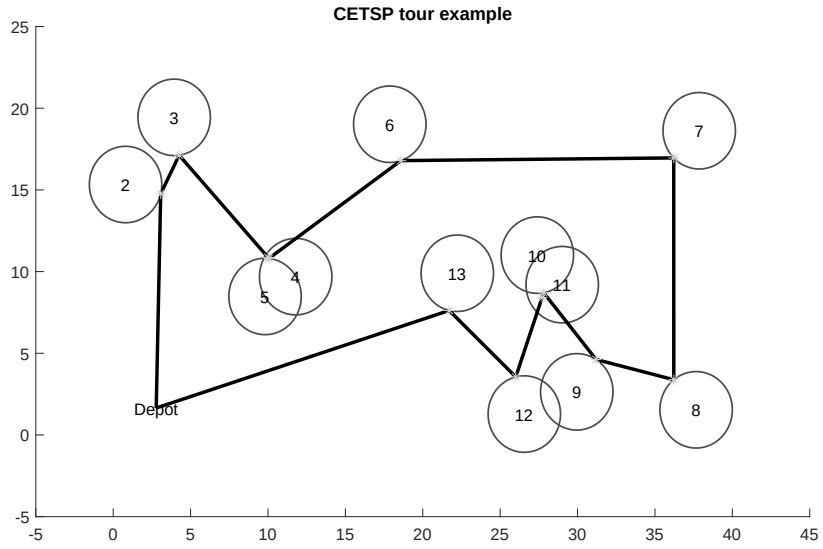



Figure 5.1: Example of a feasible solution to a CETSP instance

Figure 5.1 shows an example of a feasible solution to a 2D CETSP instance. The covering region associated with each vertex are the circles, and the asterisks represent the intersection of the tour with each region. Note that the turning point for a given vertex $v$ can be a single point (e.g. for covering region 2) or an infinite number of points (e.g. for covering region 4) as long as it is a point in the covering region $N(v)$ that is also on the path of the tour.

### 5.2.2 A B&B algorithm

Let $T^*(S)$ be an optimal tour that visits the covering regions $N(v)$ for $v \in S$ in the ordering of a sequence $S$. Given a sequence $S$ of some vertices, the problem of finding

$T^*(S)$ can be formulated and solved efficiently as a second-order cone programming (SOCP) problem (more details about the SOCP problem can be found in [15]).

Since a vertex is considered visited if the tour intersects with its covering region, a tour $T^*(S)$ can cover more vertices than the ones in $S$. Then a sequence $S$ that leads to a feasible tour of CETSP $T^*(S)$ may not contain all vertices in $V$. We will refer to a sequence not necessarily containing all vertices in $V$ as a partial sequence.

An algorithm can be created to find an optimal solution of the CETSP with vertices $V$ by constructing partial sequences with the vertices in $V$ and finding the respective shortest tour until all partial sequences that lead to feasible tours to CETSP have been checked. This is the idea behind `Cout`.

Pseudocode of the algorithm `Cout` is presented in Algorithm 6. The algorithm begins by selecting three vertices (including the depot) as the initial partial sequence $S_{root}$. An SOCP problem based on $S_{root}$ is solved to obtain the tour $T^*(S_{root})$. The algorithm then computes the set of vertices that are not covered by the tour. If all vertices are covered, the tour is optimal to the original problem and the algorithm terminates. If not all vertices are covered, the length of the partial tour $D(T^*(S_{root}))$ is a lower bound to the optimal tour. The initial partial sequence is added to the search tree as the root node.

Subsequently, in each iteration, a partial tour $S$ is selected to be explored. An uncovered vertex $v$ is selected as the branching vertex (more details in Section 5.4.1). The vertex $v$ is then inserted after the $i^{th}$ vertex in $S$ for $i = 1, 2, ..., |S|$ to create new partial sequences $S_i$ (the first vertex must be the depot). Before adding a new partial sequence $S_i$ to the search tree, an SOCP problem is solved to obtain $T^*(S_i)$ and the set of vertices not covered by $T^*(S_i)$ is computed. If all vertices are covered, $T^*(S_i)$ is also a feasible tour to the original problem and we can check if the tour length $D(T^*(S_i))$ is better than the current incumbent solution value which will be referred

to as $\hat{D}$. If not all vertices are covered by $T^*(S_i)$, the partial sequence is added to the search tree. Pruning is done by removing any partial sequence $S$ in the search tree with $D(T^*(S)) \geq \hat{D}$.

Since each node in the branch-and-bound search tree of `Cout` is associated with a partial sequence, we will also use node to refer to a partial sequence $S$. We say a node is explored if branching is done from the node or the node is pruned without branching.

**Algorithm 6:** `Cout`

Build an initial partial sequence $S_{root}$ with 3 vertices
Solve an SOCP problem based on $S_{root}$ for an optimal tour $T^*(S_{root})$ with length $D(T^*(S_{root}))$
Determine the vertices that are not covered by the tour
**if** all vertices are covered**:** Return $D(T^*(S_{root}))$
**else:** $\hat{D} := M$ where $M$ is larger than any tour length or a known feasible solution
Let $L$ be an empty list that stores unexplored partial sequences. Insert $S_{root}$ into $L$
**while** $L \neq \varnothing$**:**
    Select a partial sequence $S$ from $L$ to explore
    Select a uncovered vertex $v$ to be inserted into the partial sequence
    Generate $S_i$ for $i = 1, 2, ..., |S|$ by inserting vertex $v$ after the $i^{th}$ vertex in $S$
    **for each** partial sequence $S_i$**:**
        Solve an SOCP problem based on the sequence $S_i$ for an optimal tour $T^*(S_i)$ with length $D(T^*(S_i))$
        **if** $D(T^*(S_i)) \geq \hat{D}$**:** Continue
        Determine the vertices that are not covered by $T^*(S_i)$
        **if** all vertices are covered**:**
            **if** $\hat{D} > D(T^*(S_i))$**:** set $\hat{D} = D(T^*(S_i))$
        **else:** Insert $S_i$ into $L$
    Remove $S$ from $L$
Return $\hat{D}$

## 5.3 Search strategy

A search strategy is used to select a node to explore in each iteration of the B&B algorithm (Line 8 of Algorithm 6). Coutinho et al. [15] tests depth first search (DFS), breadth first search (BrFS) and best first search (BFS) with `Cout`. In particular, BFS selects the node with the smallest lower bound to be explored each iteration. Formally, BFS can be defined as selecting a node with the smallest value of some measure-of-best function $\mu$. For this problem, the function $\mu$ is the length of the optimal SOCP solution tour of a sequence and it can be written as

$$\mu(S) := D\left(T^*\left(S\right)\right). \tag{5.1}$$

BFS using a lower bound as a measure-of-best function has the advantage that it will not select a node with lower bound worse than the optimal tour length, which other search strategies tested cannot guarantee. Based on numerical experiments, Coutinho et al. [15] suggests that BFS outperforms both DFS and BrFS in terms of running time to prove optimality and chooses BFS as the search strategy for `Cout`.

Note that (5.1) is not a strong lower bound, and a partial sequence that is shorter than another partial sequence is likely to have a smaller lower bound as well. As a result, BFS has two disadvantages when used in `Cout`. First, BFS does not find feasible solutions early. Due to the weak lower bound, nodes with shorter partial sequences are often selected before the nodes with longer sequences. Since `Cout` constructs sequences by adding one vertex at a time, it can take a very long time for the algorithm to explore all the nodes with shorter sequences before exploring a sequence with enough vertices to result in a tour that is feasible for the original problem. If the algorithm terminates early, the search may not be able provide any feasible solution.

The benefit of finding feasible solutions early is not only so that when the algorithm terminates prematurely, there are at least feasible solutions that can be used. More importantly, the algorithm finds the child node tour and checks its the feasibility when the child node is generated (line 14 of Algorithm 6). If a good or optimal solution is found early, a child node may be pruned without getting checked and some computational cost can be saved. If solutions are not found, BFS cannot take advantage of this.

Second, BFS stores all unexplored nodes in memory. The number of nodes that has to be stored can increase exponentially as the problem size grows. There can be a large number of the nodes in the search tree with small lower bound, due to the way `Cout` constructs partial sequences as described in the first disadvantage. Since `Cout` only prunes unexplored nodes with lower bound, the number of unexplored nodes that has to be stored can be very large even for a medium sized problem, especially with the weak lower bound used. Additionally, because BFS generally does not find any feasible solutions early, even nodes with lower bound larger than the optimal tour length are stored for a long time.

Due to the complexity of the problem, we are not able to provide a better lower bound algorithm for the partial sequences. Therefore, we choose to address the disadvantages of BFS by using a different search strategy.

### 5.3.1 Contour based on covered vertices

We first consider CBFS-depth. The problems where CBFS-depth is effective share some similarities with CETSP: solutions are often deep in the search tree in which case BFS has to take a long time to reach any of them. However, there is no good lower bound algorithm for the nodes or good pruning strategy beyond lower bound in CETSP, which would make CBFS-depth even more effective compared with BFS. Our

numerical results suggest that CBFS-depth finds good feasible solutions regularly on a number of instances.

Moreover, unlike BFS, CBFS-depth does not explore nodes with the smallest lower bound successively. Since it is more likely for nodes with small lower bound to generate child nodes that cannot be pruned immediately and have to be stored for exploration later, with the same number of iterations, BFS can generate a larger number of unexplored nodes than CBFS-depth, which leads to BFS needing more space than CBFS-depth.

A better labeling function than $\kappa_{depth}$ should be one that guides the search to find good feasible solution faster while not exploring too many nodes with lower bound greater than the optimal solution (the nodes that will not be explored by BFS). Since the number of vertices covered by adding one vertex to the search tree can be very different depending on the position of the insertion, there can be significant difference in quality between nodes in the same level. Therefore, compared with depth in the search tree, the number of covered vertices by the tour $T^*(S)$ may a better indicator of how close a given node $S$ is to a feasible solution.

Therefore, we use the number of covered vertices as the contour labeling function. In particular, given a sequence $S$, the value of the labeling function $\kappa_{cv}(S)$ is the number of vertices that are covered by the tour $T^*(S)$ as written in (5.2)

$$\kappa_{cv}(S) = \text{The number of vertices covered by } T^*(S). \tag{5.2}$$

This search strategy will be referred to as CBFS-CV and the algorithm `Cout` using CBFS-CV will be referred to as `Cout-CBFS`.

Our experience suggests that CBFS-CV outperforms CBFS-depth in most instances primarily by finding feasible or optimal solutions faster than CBFS-depth, which results

in better pruning early in the search and less nodes with lower bound greater or equal to the optimal solution being explored. The space advantage over BFS is still valid as well for the same reason as CBFS-depth.

We will show in Section 5.5 that the benefit of CBFS-CV with $\kappa_{cv}$ can be demonstrated by numerical experiments.

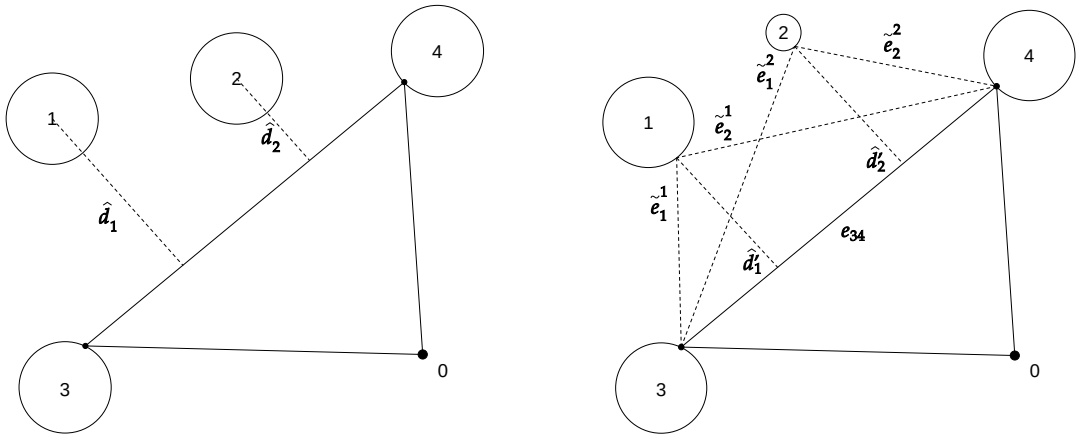
## 5.4 Further improvement

All methods discussed in this section can be applied to both 2D and 3D CETSP instances.

### 5.4.1 Branching vertex selection

The branching vertex selection used in `Cout` differs depending on the radii of the covering regions. When the radii are constant, for a given sequence $S$ and tour $T*(S)$, the algorithm computes the smallest distance from each vertex to all edges in the tour, represented by $\hat{d}_v$ for vertex $v$, and pick the vertex with the largest distance to be the node inserted into the tour. For example, in Figure 5.2a, the smallest distance from both uncovered vertices 1 and 2 are to the edge between covering region 3 and 4, represented as $\hat{d}_1$ and $\hat{d}_2$ respectively. Since $\hat{d}_1 > \hat{d}_2$, the algorithm will pick vertex 1 to be the next branching vertex.

On the other hand, when radius of covering regions are arbitrary, `Cout` uses a different branching variable selection method. For each vertex $v$, the algorithm finds the smallest value of $\tilde{d}_v = \tilde{e}_1^v + \tilde{e}_2^v - e_{ij}$ where $e_{ij}$ is the length of the edge from vertex $i$ to vertex $j$, and $\tilde{e}_1^v + \tilde{e}_2^v$ is the shortest sum of distances from a point on the covering region of $v$ to the end points of the edge from vertex $i$ to vertex $j$. Then the vertex with the largest $\tilde{d}$ value is the branching vertex. For example, in Figure 5.2b, since




(a) Constant radii (b) Arbitrary radii

Figure 5.2: Branching variable selection methods

$\tilde{d}_2 > \tilde{d}_1$, vertex 2 is the branching vertex.

We propose a simple change to the scheme that uses the value $\hat{d}'_v = \hat{d}_v - r_v$ which is the difference between the distance $\hat{d}_v$ and the radius of the corresponding covering region. This is the same scheme as the one in `Cout` when the instance has constant radii, and a slightly different one from `Cout` when the radii are arbitrary. We will show in Section 5.5 that this brings a substantial improvement to the performance of the B&B algorithms on instances with arbitrary radii.

### 5.4.2 Reduce the number of vertex coverage checks

The process of checking the feasibility of an optimal tour of a partial sequence (line 14 of Algorithm 6) involves checking whether all vertices not in the partial sequence is covered by the corresponding optimal tour. To correctly check the coverage of a vertex by a tour, it is necessary to check the covering region of the vertex against each edge in the tour to see if intersection occurs. However, since a child node is a partial sequence that differs from the partial sequence of its direct parent node by a single vertex inserted into the sequence, most of the turning points associated with vertices

far from the inserting position may not change from a parent node to a child node.

Let a child sequence be created by inserting some vertex after the $k^{th}$ vertex in the parent sequence. Then a vertex in the parent sequence is said to be $a$ steps away from the inserting position for some non-negative integer $a$, if it is covered by an edge starting from the $(k-a)^{th}$ vertex in the optimal parent sequence tour.

Given the observation that most turning points in the tour do not change from a parent node to a child node, we can reduce the number of coverage checks by the following method: if a vertex not in the sequence is covered by an edge of the optimal tour of the parent node more than $a$ steps from the inserting position in the current child node sequence, we assume that it is still covered.

Note that it is possible for such a vertex to not be covered by a tour when we assume it is, as the tour may change substantially from a parent node to a child node. Therefore, when the check indicates that all nodes are covered by the current tour, an additional check is set to confirm coverage of all vertices not in the sequence. The child node can be inserted back into the search tree, if it turns out that not all vertices are actually covered. This does not affect the ability of the B&B algorithm to prove optimality, since the child nodes inserted back into the tree now have correct coverage information.

On the other hand, if a vertex not in the sequence is covered by an edge of the optimal tour of the parent node no more than $a$ steps away from the inserting position, we can check its covering region against only the edges close to the inserting position. If it is still covered, we do not need to check it against other edges; but if it is not covered by the edges we checked, we can then check the covering region against all other edges in the current tour.

After some testing, we determined that $a$ can be set to 1, which resulted in noticeable reduction in the number of vertices to check while not causing significant increase

in the chance of a coverage error (which will lead to more number of nodes explored).

### 5.4.3 Find better feasible solutions

The quality of the feasible solutions found is another aspect of the algorithm that can be improved. Given a feasible solution to a CETSP, the tour $T$ must intersect with the covering regions of all vertices for at least one point. Therefore, we can obtain a set of points consisting of one intersecting point per covering region, which can be used to construct a conventional TSP. Then we have the following lemma.

**Lemma 5.1.** *If an optimal tour $T_C^*$ for the constructed TSP is shorter than $T$, then that tour is a better feasible solution to the CETSP. Furthermore, if we then solve an SOCP problem constructed with the optimal sequence $S_C^*$ of the TSP, the resulting tour will be at least as short as $T_C^*$.*

*Proof.* Since the TSP is constructed using an intersecting point of each covering region, any feasible solution to the TSP is a feasible solution to the CETSP. If $T_C^*$ is shorter than $T$, it must be a better feasible solution to the CETSP than $T$.

Since the tour $T_C^*$ is only a feasible solution for the CETSP with the sequence $S_C^*$, optimal solution to the SOCP problem, which is the optimal tour of the sequence $S_C^*$ for the CETSP, will be at least as good as the optimal solution to the TSP. ∎

Lemma 5.1 suggests that we can potentially find better feasible solutions to the CETSP by constructing and solving a TSP based on an existing feasible solution to the CETSP, and solve an SOCP problem to obtain another feasible solution to the CETSP. The process can be repeated with a new set of intersecting points based on the optimal tour for the SOCP problem, until no better tour for the TSP based on the intersecting points can be found.

For this improvement to work efficiently, we used Concorde [4] as it is considered to be the state-of-the-art for solving symmetric TSP [12]. Since Concorde takes integer distance value, we multiply all distances by 100000 and round to integer values. Since we are only interested in the sequences produced by Concorde, the loss of precision caused by rounding is not going to affect the CETSP tour length.

Note that the choice of the intersecting points can affect the symmetric TSP solution. For our algorithm, we used the first intersecting points with the tour for each covering region belonging to vertices that are not in the partial sequence. For the vertices that are in the partial sequence, the corresponding turning points are used.

### 5.4.4 Improve lower bound

A major issue that affects both `Cout` and `Cout-CBFS` is the space needed to store the unexplored nodes. On problems that require exploring a large number of partial sequences, the number of unexplored nodes can increase exponentially. Therefore, a probing method is used to further reduce the number of unexplored nodes that needs to be stored at any given moment during the search.

In particular, given a node $S$ selected for exploration, we first apply a limited BFS search with that node as the root. There can be two outcomes from the short search: either the search terminates with the entire subtree rooted at $S$ explored, or the search terminates with some nodes on the subtree rooted at $S$ still unexplored. If the first scenario occurs, the current node can be pruned. When the second scenario occurs, we first tried to simply take the still unexplored nodes generated and insert them back into the original search tree. However, it turned out that this also generated unexplored nodes too rapidly and was unable to address the space constraint issue. We then determined that the best approach is to keep some information (on vertex coverage and the SOCP solution) about each node generated during the search so that when

the same nodes are regenerated outside of the brief search, the stored information can be used to avoid repeated computation. We will refer to this method as a look-ahead search, or LAS for short.

## 5.5 Numerical experiments

All experiments are done on a computer with a 4.1GHz CPU and 12G of available memory and the implementation runs on a single thread. A time limit of 14400 seconds and a space limit of 8G memory usage are applied on each run of the algorithms.

The problem instances are obtained from Mennell [30] since the instances from Behdani and Smith [9] are already solved easily by `Cout` in Coutinho et al. [15]. The difficulty of each instance is associated with *overlap ratio*, which is defined as the ratio between the mean of all radii and the largest size of the rectangle that involves all covering regions. When overlap ratio is small, the mean radius is small and the CETSP approaches a TSP. Coutinho et al. [15] shows that the problems are more difficult to solve for `Cout` as overlap ratio becomes smaller.

The 62 instances provided in Mennell [30] includes: 28 instances (the instances with "d493", "dsj1000", "kroD100", "lin318", "pcb442", "rat195" and "rd400" in the names) are derived from 7 conventional instances from TSPLIB, denoted as *TSPLIB* instances; 14 instances with "team" and "bonus" in the names are from Gulczynski et al. [22], denoted as *Team* instances; 20 instances are generated by Mennell [30], denoted as *Geometric* instances.

All 20 *Geometric* instances are all constant radii instances, as the radii of covering regions in the same problem are the same. However, the overlap ratio varies from instance to instance. We say that these instances have constant radii and varied overlap ratios. There are 7 *Team* instances that also have constant radii and varied overlap

ratios. The other 7 *Team* instances are arbitrary radii instances, as the radii of covering regions in the same problem are different. Out of the 28 *TSPLIB* instances, 7 are arbitrary radii instances. The other 21 are constant radii instances divided into three sets that have the same overlap ratio within each set. The overlap ratios are 0.02, 0.1, and 0.3. We say that these instances have constant radii and constant overlap ratios.

Both the *Team* and the *TSPLIB* instances have 2D and 3D versions while the *Geometric* instances only have 2D version. There exist best known feasible solution values for all these problems from previous work [15, 43].

### 5.5.1 Implementation details

The original implementation of `Cout` is done in C++ and we obtained the code from the original authors. Note that, in addition to incorporating the changes mentioned in previous sections, we have improved the efficiency of the implementation of the algorithm `Cout` as well. In fact, the most significant improvement from the results reported in Coutinho et al. [15] is from the improvement of the implementation, as it yields running times that are often one half or one quarter of the original code.

Two aspects of the implementation improvement have the most substantial impact to the performance. First, the process of checking whether a vertex is covered by the optimal tour of a partial sequence is optimized. In addition to the improvement discussed in Section 5.4.2, we made additional changes to reduce the redundant computation. Second, since the SOCP problems for the child nodes of the same parent has a large number of constraints in common, we reduced the number of constraints that have to be created for each SOCP problem by reusing as many constraints from the previous SOCP problem as possible. Moreover, there are other improvement steps, including using more suitable data structures for node storage and retrieval and reducing communication overhead between functions, which also contribute to the improvement

in running time.

Table 5.1: Implementation improvement comparison

| | Cout-Orig | | Cout | |
|---|---|---|---|---|
| Instance | Running Time | SOCP Time | Running Time | SOCP Time |
| d493 | 27.964 | 12.066 | 12.516 | 10.162 |
| dsj1000 | 8839.783 | 2361.181 | 2123.764 | 1774.360 |
| kroD100 | 1.381 | 0.976 | 0.847 | 0.770 |
| lin318 | 6349.202 | 2393.808 | 2038.748 | 1814.157 |
| rat195 | 11.526 | 7.169 | 5.937 | 5.469 |

Let `Cout-Orig` represent the original implementation of `Cout` and let `Cout` represent the improved implementation of the original algorithm. Figure 5.1 shows the results of the implementation improvement on the constant radius instances from the TSPLIB Instances set with overlap ratio 0.1 that can be solved to optimality by both `Cout-Orig` and `Cout`. As can be seen, the implementation improvement have substantially reduced the overall running time (columns "Running Time") as well as the total SOCP solution time (columns "SOCP Time").

For the remainder of this section, if not specified otherwise, all variations of `Cout` experimented are using the improved implementation. We are going to test 3 variations of `Cout` to show the affect of the changes we have proposed in this chapter. It includes the following: `Cout`, `Cout` plus the changes discussed in Sections 5.4.1, 5.4.2, and 5.4.3 called `Cout+`, `Cout-CBFS` plus the changes in Sections 5.4.1, 5.4.2, and 5.4.3 called `Cout-CBFS+`, and `Cout-CBFS+` with the addition of LAS discussed in Section 5.4.4 called `Cout-CBFS-LAS+`. LAS is implemented with a time limit of 0.5 second.

### 5.5.2 Problems with feasible solutions

In this section, we will first present the results of applying the B&B algorithms on the 62 2D instances with best known feasible solution values. Then, we will present the results of applying the B&B algorithms on 42 3D instances (*Geometric* instances do not have 3D variants) with best known feasible solution values.

Table 5.2 presents the results of all algorithms on 62 2D instances. The column "Instance" contains the name of the instances, and the column "Known" contains the best known feasible solution for each instance in the literature. The column "UB" contains the best upper bound (incumbent solution) found during the search. If it is marked by -, it means no feasible solution better than the best known solution is found. If the number in column "UB" is marked by *, it means that the corresponding instance is solved to optimality; if the number in column "UB" is bold, then that solution is the best feasible solution found compared with all algorithms and the best known solution. The column "LB" contains the best lower bound when the algorithm terminates. The columns "Time" are the running times for each algorithm (the ones marked with * corresponds to instances that are solved to optimality) and the columns "Unexp" are the maximum number of unexplored nodes that each algorithm has to store.

On instances that can be solved optimally, both `Cout+` and `Cout-CBFS+` outperform `Cout` on most instances in terms of running time. The advantage is more substantial on problems that take more than a few seconds to solve, and an instance `chaoSingleDep` can be solved by `Cout+` and `Cout-CBFS+` but not by `Cout`. The new branching vertex selection scheme makes `Cout+` and `Cout-CBFS+` use substantially less time to prove optimality than `Cout` on instances with arbitrary radii. The instances `chaoSingleDep` and `kroD100rdmRad` are solved to optimality for the first time. `Cout-CBFS+` matches the performance with the two algorithms using BFS on most instances but the max-

imum number of unexplored nodes that Cout-CBFS+ has to store is substantially less than the maximum number stored by the two algorithms using BFS. In terms of Cout-CBFS-LAS+, although it is slower than other algorithms tested on most of the solved instances (i.e., chaoSingleDep is not solved by Cout-CBFS-LAS+), it also needs to store substantially fewer unexplored nodes than any other algorithms.

On instances that cannot be solved optimally, new best known feasible solutions are found for a number of instances by Cout-CBFS+ or Cout-CBFS-LAS+: bonus1000, bubbles4, team3_300, pcb442 (overlap ratio 0.1), rd400 (overlap ratio 0.1), d493rdmRad, lin318rdmRad and team2_200rdmRad. On the other hand, algorithms using CBFS-CV terminate with worse lower bounds than the BFS variations, as CBFS-CV does not selects the node with the minimum global lower bound to explore every iteration.

Table 5.3 presents the results of all algorithms on 42 3D instances. The column notations are the same as the ones in Table 5.2. On instances that can be solved optimally, Cout+ still runs faster than Cout on many instances, e.g. lin318 with overlap ratio 0.1. However, on instance team6_500, the running time for Cout+ and Cout-CBFS+ is noticeably worse than Cout, which may be attributed to coverage check reduction discussed in Chapter 5.4.2 not effective. On instance team2_200, Cout-CBFS+ takes more time to run than both Cout+ and Cout, which can happen if CBFS cannot find good feasible solutions early in the search and explore a number of nodes with lower bound greater than the optimal solution. However, on other instances that are solved optimally, Cout-CBFS+ matches the running time of Cout+ while stores fewer unexplored nodes. Cout-CBFS-LAS+ runs slower than Cout-CBFS+, but stores even fewer nodes, similar to the results on 2D instances.

There is only one instance with arbitrary radii that is solved optimally, and the new branching vertex selection scheme improves the running time substantially similar to the benefit shown in 2D instances.

On instances that cannot be solved optimally, new best known feasible solutions are found for a number of instances by `Cout-CBFS+` or `Cout-CBFS-LAS+`: `team1_100`, `pcb442` (overlap ratio 0.1), `rat195` (overlap ratio 0.1), `rd400` (overlap ratio 0.3), `dsj1000rdmRad`, `kroD100rdmRad`, `lin318rdmRad`, `pcb442rdmRad`, `team1_100rdmRad`, `team3_300rdmRad` and `team5_499rdmRad`.

Table 5.2: Results on 62 2D instances with known feasible solution

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| **Constant radii, varied overlap ratios** | | | | | | | | | | | | | | | | | |
| bonus1000 | 402.47 | - | 356.24 | 1315.95 | 203775 | - | 356.22 | 907.53 | 203169 | 377.79 | 352.11 | 2745.38 | 199248 | **377.66** | 355.42 | 14400.11 | 107815 |
| bubbles1 | 349.13 | 349.13* | 349.13 | 0.06 | 6 | 349.13* | 349.13 | 0.08 | 6 | 349.13* | 349.13 | 0.06 | 6 | 349.13* | 349.13 | 0.08 | 1 |
| bubbles2 | 428.28 | 428.28* | 428.28 | 0.13 | 11 | 428.28* | 428.28 | 0.12 | 11 | 428.28* | 428.28 | 0.13 | 7 | 428.28* | 428.28 | 0.11 | 1 |
| bubbles3 | 530.73 | 529.95* | 529.95 | 72.85 | 755 | 529.95* | 529.95 | 69.66 | 755 | 529.95* | 529.95 | 76.53 | 213 | 529.95* | 529.95 | 84.86 | 93 |
| bubbles4 | 825.33 | - | 693.92 | 5442.00 | 1330915 | - | 693.87 | 4118.88 | 1323365 | 807.69 | 678.41 | 14400.02 | 1242369 | **802.96** | 676.19 | 14400.03 | 271920 |
| bubbles5 | 1062.34 | - | 852.77 | 3016.59 | 1039115 | - | 852.72 | 2189.10 | 1034982 | - | 825.58 | 6245.84 | 1033571 | - | 830.58 | 14400.28 | 416668 |
| bubbles6 | 1263.68 | - | 992.84 | 2443.30 | 852616 | - | 992.79 | 1736.78 | 849274 | - | 960.25 | 4721.64 | 781313 | - | 970.98 | 14400.11 | 436838 |
| bubbles7 | 1639.33 | - | 1118.81 | 2006.14 | 652933 | - | 1118.73 | 1415.39 | 650087 | - | 1059.15 | 4565.80 | 639049 | - | 1089.16 | 14400.05 | 620791 |
| bubbles8 | 1972.99 | - | 1245.88 | 2057.09 | 600362 | - | 1245.69 | 1427.54 | 598761 | - | 1161.51 | 4279.41 | 552561 | - | 1194.89 | 10292.67 | 553000 |
| bubbles9 | 2330.31 | - | 1367.00 | 1987.34 | 520949 | - | 1366.92 | 1396.47 | 519740 | - | 1248.17 | 4058.03 | 436761 | - | 1291.95 | 7377.86 | 424284 |
| chaoSingleDep | 1022.88 | - | 1022.49 | 14400.01 | 304505 | 1022.88* | 1022.88 | 12839.94 | 303960 | 1022.88* | 1022.88 | 13384.27 | 256619 | - | 1009.36 | 14400.06 | 47157 |
| concentricCircles1 | 53.16 | 53.16* | 53.16 | 3.29 | 87 | 53.16* | 53.16 | 3.05 | 87 | 53.16* | 53.16 | 3.14 | 96 | 53.16* | 53.16 | 3.19 | 26 |
| concentricCircles2 | 153.13 | - | 152.09 | 14400.01 | 318702 | - | 152.29 | 14400.01 | 318702 | - | 150.81 | 14400.02 | 258426 | - | 148.69 | 14400.05 | 56129 |
| concentricCircles3 | 271.08 | - | 251.42 | 14400.00 | 2055751 | - | 251.52 | 14400.02 | 2115260 | - | 245.82 | 14400.02 | 1004596 | - | 243.29 | 14400.06 | 197843 |
| concentricCircles4 | 454.46 | - | 369.28 | 7330.04 | 3371765 | - | 369.27 | 6887.36 | 3369164 | - | 349.78 | 14400.05 | 2065545 | - | 346.74 | 14400.28 | 328018 |
| concentricCircles5 | 645.38 | - | 466.58 | 4927.82 | 2237200 | - | 466.62 | 4246.31 | 2223187 | - | 441.41 | 12562.47 | 2088163 | - | 439.61 | 14400.13 | 482283 |
| rotatingDiamonds1 | 32.39 | 32.39* | 32.39 | 0.05 | 5 | 32.39* | 32.39 | 0.05 | 5 | 32.39* | 32.39 | 0.03 | 5 | 32.39* | 32.39 | 0.06 | 1 |
| rotatingDiamonds2 | 140.48 | 140.48* | 140.48 | 331.62 | 9941 | 140.48* | 140.48 | 329.97 | 9979 | 140.48* | 140.48 | 338.19 | 8592 | 140.48* | 140.48 | 405.77 | 1994 |
| rotatingDiamonds3 | 380.88 | - | 353.85 | 6120.85 | 1405182 | - | 353.83 | 5426.24 | 1400553 | - | 347.34 | 6929.06 | 1356812 | - | 342.74 | 14400.05 | 224873 |
| rotatingDiamonds4 | 770.66 | - | 594.10 | 2303.83 | 951366 | - | 594.10 | 1951.97 | 951608 | - | 568.43 | 3867.34 | 851440 | - | 574.37 | 14400.44 | 467060 |
| rotatingDiamonds5 | 1510.75 | - | 1096.71 | 1586.34 | 465355 | - | 1096.54 | 1335.25 | 463865 | - | 1031.63 | 1615.11 | 412614 | - | 1047.41 | 11266.34 | 421404 |
| team1_100 | 307.34 | 307.34* | 307.34 | 4.36 | 47 | 307.34* | 307.34 | 4.31 | 47 | 307.34* | 307.34 | 4.22 | 28 | 307.34* | 307.34 | 4.20 | 22 |
| team2_200 | 246.68 | 246.68* | 246.68 | 0.21 | 3 | 246.68* | 246.68 | 0.25 | 3 | 246.68* | 246.68 | 0.19 | 3 | 246.68* | 246.68 | 0.22 | 1 |
| team3_300 | 465.80 | - | 453.07 | 8701.68 | 755329 | - | 453.03 | 7659.66 | 751100 | **461.89** | 451.15 | 14400.08 | 586906 | **461.89** | 443.95 | 14400.03 | 89883 |
| team4_400 | 680.21 | - | 505.22 | 2101.84 | 667652 | - | 505.21 | 1576.95 | 666558 | - | 486.02 | 2798.80 | 649460 | - | 495.26 | 14400.34 | 509682 |
| team5_499 | 702.82 | - | 523.60 | 2667.32 | 681211 | - | 523.58 | 2047.25 | 679609 | - | 503.08 | 3514.72 | 595536 | - | 510.19 | 14400.20 | 488475 |
| team6_500 | 225.22 | 225.22* | 225.22 | 0.33 | 3 | 225.22* | 225.22 | 0.32 | 3 | 225.22* | 225.22 | 0.30 | 3 | 225.22* | 225.22 | 0.31 | 1 |
| **Constant radii, constant overlap ratios** | | | | | | | | | | | | | | | | | |
| **overlap ratio: 0.02** | | | | | | | | | | | | | | | | | |

(continued on next page)

Table 5.2: Results on 62 2D instances with known feasible solution (continued)

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| d493 | 202.79 | - | 145.97 | 2281.72 | 672196 | - | 146.01 | 2020.00 | 686051 | - | 141.92 | 2223.64 | 577988 | - | 143.85 | 14400.12 | 563001 |
| dsj1000 | 935.74 | - | 554.18 | 1813.90 | 346984 | - | 554.15 | 1624.13 | 346362 | - | 522.50 | 2085.00 | 290840 | - | 533.05 | 6758.35 | 293136 |
| kroD100 | 159.04 | - | 146.50 | 14400.03 | 2960628 | - | 146.45 | 14400.00 | 2904668 | - | 141.33 | 14400.08 | 1156966 | - | 139.66 | 14400.30 | 200763 |
| lin318 | 2842.32 | - | 1998.34 | 2850.10 | 1042255 | - | 1998.27 | 2785.78 | 1040645 | - | 1899.03 | 4140.97 | 955241 | - | 1929.76 | 14400.01 | 630980 |
| pcb442 | 323.03 | - | 186.24 | 2836.97 | 871721 | - | 186.23 | 2264.58 | 870278 | - | 173.43 | 4464.92 | 673057 | - | 177.77 | 12525.34 | 707190 |
| rat195 | 158.79 | - | 109.04 | 3841.56 | 1667957 | - | 109.04 | 3477.34 | 1663434 | - | 104.09 | 7572.78 | 1482668 | - | 104.67 | 14400.16 | 611527 |
| rd400 | 1033.41 | - | 568.90 | 2986.97 | 956641 | - | 568.89 | 2354.85 | 955634 | - | 524.21 | 4464.77 | 763688 | - | 538.70 | 13560.61 | 820826 |
| **overlap ratio: 0.10** | | | | | | | | | | | | | | | | | |
| d493 | 101.73 | 100.72* | 100.72 | 12.52 | 988 | 100.72* | 100.72 | 12.57 | 988 | 100.72* | 100.72 | 11.69 | 170 | 100.72* | 100.72 | 15.44 | 123 |
| dsj1000 | 376.10 | 373.76* | 373.76 | 2123.76 | 74533 | 373.76* | 373.76 | 1965.24 | 74533 | 373.76* | 373.76 | 1925.67 | 8032 | 373.76* | 373.76 | 2326.51 | 2884 |
| kroD100 | 89.67 | 89.67* | 89.67 | 0.85 | 28 | 89.67* | 89.67 | 0.88 | 28 | 89.67* | 89.67 | 0.84 | 20 | 89.67* | 89.67 | 0.88 | 12 |
| lin318 | 1408.48 | 1394.63* | 1394.63 | 2038.75 | 129652 | 1394.63* | 1394.63 | 1844.84 | 129652 | 1394.63* | 1394.63 | 1940.69 | 20889 | 1394.63* | 1394.63 | 2782.77 | 6624 |
| pcb442 | 147.24 | - | 137.29 | 2341.25 | 454212 | - | 137.28 | 1819.64 | 452046 | **144.07** | 136.23 | 3245.20 | 445709 | 144.23 | 136.66 | 14400.39 | 181841 |
| rat195 | 68.08 | 67.99* | 67.99 | 5.94 | 362 | 67.99* | 67.99 | 5.84 | 362 | 67.99* | 67.99 | 6.55 | 89 | 67.99* | 67.99 | 7.74 | 56 |
| rd400 | 466.10 | - | 433.23 | 3103.74 | 586236 | - | 433.22 | 2491.76 | 584097 | **450.57** | 430.93 | 11196.91 | 561624 | 455.52 | 428.85 | 14400.40 | 163729 |
| **overlap ratio: 0.30** | | | | | | | | | | | | | | | | | |
| d493 | 69.76 | 69.76* | 69.76 | 0.26 | 1 | 69.76* | 69.76 | 0.28 | 1 | 69.76* | 69.76 | 0.28 | 1 | 69.76* | 69.76 | 0.28 | 1 |
| dsj1000 | 199.95 | 199.95* | 199.95 | 0.56 | 3 | 199.95* | 199.95 | 0.58 | 3 | 199.95* | 199.95 | 0.58 | 3 | 199.95* | 199.95 | 0.57 | 1 |
| kroD100 | 58.54 | 58.54* | 58.54 | 0.06 | 1 | 58.54* | 58.54 | 0.06 | 1 | 58.54* | 58.54 | 0.05 | 1 | 58.54* | 58.54 | 0.06 | 1 |
| lin318 | 765.96 | 765.96* | 765.96 | 0.18 | 2 | 765.96* | 765.96 | 0.18 | 2 | 765.96* | 765.96 | 0.19 | 2 | 765.96* | 765.96 | 0.18 | 1 |
| pcb442 | 83.54 | 83.54* | 83.54 | 0.23 | 1 | 83.54* | 83.54 | 0.24 | 1 | 83.54* | 83.54 | 0.23 | 1 | 83.54* | 83.54 | 0.23 | 1 |
| rat195 | 45.70 | 45.70* | 45.70 | 0.11 | 1 | 45.70* | 45.70 | 0.10 | 1 | 45.70* | 45.70 | 0.11 | 1 | 45.70* | 45.70 | 0.11 | 1 |
| rd400 | 224.84 | 224.84* | 224.84 | 0.22 | 3 | 224.84* | 224.84 | 0.23 | 3 | 224.84* | 224.84 | 0.23 | 3 | 224.84* | 224.84 | 0.24 | 1 |
| **Arbitrary radii** | | | | | | | | | | | | | | | | | |
| bonus1000rdmRad | 938.27 | - | 496.51 | 1363.44 | 275492 | - | 500.81 | 858.39 | 266014 | - | 461.30 | 1256.86 | 231815 | - | 477.77 | 4476.59 | 206194 |
| d493rdmRad | 135.02 | - | 125.05 | 2077.08 | 411955 | - | 129.56 | 2975.97 | 404738 | **134.23** | 128.72 | 14400.02 | 401074 | **134.23** | 127.77 | 14400.19 | 96578 |
| dsj1000rdmRad | 625.25 | - | 499.70 | 876.46 | 205481 | - | 532.40 | 693.73 | 203285 | - | 518.81 | 1056.77 | 202560 | - | 532.42 | 10436.77 | 184177 |
| kroD100rdmRad | 141.83 | - | 139.20 | 14400.02 | 715118 | 141.83* | 141.83 | 6338.82 | 83787 | 141.83* | 141.83 | 6120.88 | 70872 | 141.83* | 141.83 | 8313.14 | 15535 |
| lin318rdmRad | 2079.49 | - | 1805.37 | 2621.18 | 716930 | - | 1864.67 | 2253.45 | 709420 | **2066.02** | 1832.63 | 5520.84 | 703327 | 2070.63 | 1833.08 | 14400.36 | 229422 |
| pcb442rdmRad | 221.16 | - | 174.54 | 1650.51 | 558556 | - | 178.47 | 946.92 | 530495 | - | 171.69 | 1859.34 | 492301 | - | 176.81 | 14400.48 | 365522 |

(continued on next page)

Table 5.2: Results on 62 2D instances with known feasible solution (continued)

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| rat195rdmRad | 68.22 | 68.22* | 68.22 | 1.76 | 34 | 68.22* | 68.22 | 1.04 | 23 | 68.22* | 68.22 | 0.98 | 14 | 68.22* | 68.22 | 0.98 | 10 |
| rd400rdmRad | 1252.38 | - | 575.06 | 3942.87 | 958997 | - | 588.88 | 2772.94 | 987333 | - | 536.54 | 6738.34 | 893980 | - | 553.06 | 14400.48 | 951916 |
| team1_100rdmRad | 388.54 | 388.54* | 388.54 | 112.97 | 2680 | 388.54* | 388.54 | 44.43 | 1061 | 388.54* | 388.54 | 42.94 | 878 | 388.54* | 388.54 | 42.47 | 360 |
| team2_200rdmRad | 622.74 | - | 490.87 | 3338.85 | 1310359 | - | 513.47 | 2715.08 | 1261478 | **621.19** | 495.17 | 12562.73 | 1267737 | - | 494.33 | 14400.47 | 352445 |
| team3_300rdmRad | 378.09 | 378.09* | 378.09 | 191.32 | 3667 | 378.09* | 378.09 | 47.97 | 988 | 378.09* | 378.09 | 45.23 | 844 | 378.09* | 378.09 | 45.63 | 295 |
| team4_400rdmRad | 1006.71 | - | 551.87 | 3684.85 | 963167 | - | 571.78 | 2558.49 | 966018 | - | 529.69 | 4214.30 | 757191 | - | 547.05 | 14400.08 | 739075 |
| team5_499rdmRad | 446.19 | 446.19* | 446.19 | 6546.41 | 51720 | 446.19* | 446.19 | 1221.78 | 11228 | 446.19* | 446.19 | 1139.97 | 9076 | 446.19* | 446.19 | 1572.73 | 2786 |
| team6_500rdmRad | 621.99 | - | 483.63 | 1418.04 | 439814 | - | 500.95 | 882.52 | 427255 | - | 485.49 | 2193.78 | 420733 | - | 496.85 | 14400.03 | 374064 |

Table 5.3: Results on 42 3D instances with known feasible solution

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| **Constant radii, varied overlap ratios** | | | | | | | | | | | | | | | | | |
| bonus1000 | 941.35 | - | 464.71 | 1426.75 | 284088 | - | 464.68 | 1104.93 | 283244 | - | 416.90 | 1371.69 | 223019 | - | 429.70 | 3481.25 | 194021 |
| team1_100 | 820.73 | - | 708.29 | 6989.10 | 3336890 | - | 708.27 | 6212.88 | 3332300 | **816.29** | 677.03 | 14400.11 | 1556800 | - | 671.27 | 14400.33 | 282951 |
| team2_200 | 283.24 | 273.38* | 273.38 | 193.70 | 18729 | 273.38* | 273.38 | 177.43 | 18849 | 273.38* | 273.38 | 210.75 | 2500 | 273.38* | 273.38 | 318.34 | 1093 |
| team3_300 | 1484.41 | - | 771.77 | 3709.33 | 1378619 | - | 771.68 | 3095.16 | 1371321 | - | 704.15 | 4153.81 | 1058775 | - | 722.53 | 14400.09 | 762488 |
| team4_400 | 753.81 | - | 510.07 | 2852.53 | 854309 | - | 510.05 | 2291.01 | 852793 | - | 483.57 | 5701.18 | 705017 | - | 493.56 | 14400.30 | 705283 |
| team5_499 | 1924.53 | - | 714.33 | 4439.16 | 957412 | - | 714.32 | 2871.58 | 956861 | - | 603.65 | 9196.08 | 794176 | - | 627.92 | 13654.23 | 950626 |
| team6_500 | 236.96 | 230.92* | 230.92 | 0.35 | 29 | 230.92* | 230.92 | 1.00 | 29 | 230.92* | 230.92 | 1.06 | 16 | 230.92* | 230.92 | 1.06 | 2 |
| **Constant radii, constant overlap ratios** | | | | | | | | | | | | | | | | | |
| **overlap ratio: 0.02** | | | | | | | | | | | | | | | | | |
| d493 | 1353.14 | - | 474.93 | 5350.33 | 998896 | - | 474.92 | 3729.49 | 998279 | - | 424.64 | 14206.02 | 885531 | - | 439.09 | 14401.38 | 902556 |
| dsj1000 | 3147.87 | - | 753.30 | 3342.78 | 484561 | - | 753.25 | 2770.16 | 483792 | - | 580.97 | 8497.03 | 395579 | - | 589.41 | 9713.30 | 460953 |
| kroD100 | 202.02 | - | 152.19 | 10050.02 | 4063111 | - | 152.23 | 8777.73 | 4122139 | - | 143.07 | 14400.01 | 1836422 | - | 143.10 | 14400.33 | 457850 |
| lin318 | 3044.27 | - | 2010.22 | 4227.02 | 1288664 | - | 2010.09 | 5075.60 | 1284165 | - | 1883.51 | 6737.13 | 1053590 | - | 1912.36 | 14400.34 | 673976 |
| pcb442 | 404.49 | - | 187.35 | 3779.17 | 876138 | - | 187.34 | 3912.85 | 873018 | - | 170.21 | 8948.84 | 842552 | - | 176.25 | 13612.45 | 955347 |

(continued on next page)

Table 5.3: Results on 42 3D instances with known feasible solution (continued)

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| rat195 | 291.26 | - | 129.13 | 5564.59 | 1967434 | - | 129.13 | 4231.57 | 1966577 | - | 115.91 | 13900.16 | 2051876 | - | 117.93 | 14400.64 | 948587 |
| rd400 | 3218.20 | - | 879.05 | 4316.09 | 1073228 | - | 879.63 | 5094.65 | 1099461 | - | 704.19 | 14401.57 | 1036363 | - | 705.42 | 14400.03 | 1036443 |
| **overlap ratio: 0.10** | | | | | | | | | | | | | | | | | |
| d493 | 665.06 | - | 420.43 | 2566.57 | 680911 | - | 420.42 | 2092.57 | 679661 | - | 409.12 | 3258.50 | 594782 | - | 415.87 | 14400.06 | 568853 |
| dsj1000 | 1021.25 | - | 593.62 | 1448.00 | 298311 | - | 593.58 | 1171.75 | 297452 | - | 535.44 | 1462.67 | 232405 | - | 552.22 | 3815.59 | 204303 |
| kroD100 | 91.67 | 91.66* | 91.66 | 3.17 | 53 | 91.66* | 91.66 | 3.16 | 53 | 91.66* | 91.66 | 3.24 | 39 | 91.66* | 91.66 | 3.44 | 23 |
| lin318 | 1443.43 | 1398.50* | 1398.50 | 3657.72 | 355089 | 1398.50* | 1398.50 | 3140.94 | 355089 | 1398.50* | 1398.50 | 3076.22 | 28284 | 1398.50* | 1398.50 | 3906.84 | 7044 |
| pcb442 | 154.81 | - | 137.53 | 2030.41 | 467254 | - | 137.52 | 1498.71 | 464318 | 144.60 | 136.95 | 4148.57 | 454146 | **143.99** | 137.33 | 14400.05 | 167109 |
| rat195 | 112.40 | - | 89.07 | 3521.02 | 1277433 | - | 89.06 | 3246.07 | 1271169 | 99.47 | 87.58 | 14400.05 | 941532 | **99.46** | 87.09 | 14400.08 | 190802 |
| rd400 | 1552.72 | - | 756.72 | 3150.70 | 990697 | - | 756.66 | 3138.68 | 987085 | - | 679.36 | 4828.32 | 729710 | - | 708.18 | 12105.27 | 765429 |
| **overlap ratio: 0.30** | | | | | | | | | | | | | | | | | |
| d493 | 335.59 | 325.21* | 325.21 | 8.48 | 1366 | 325.21* | 325.21 | 7.36 | 1366 | 325.21* | 325.21 | 9.02 | 180 | 325.21* | 325.21 | 10.88 | 105 |
| dsj1000 | 270.40 | 267.75* | 267.75 | 4.10 | 326 | 267.75* | 267.75 | 4.32 | 310 | 267.75* | 267.75 | 5.62 | 183 | 267.75* | 267.75 | 4.91 | 31 |
| kroD100 | 58.93 | 58.93* | 58.93 | 0.06 | 1 | 58.93* | 58.93 | 0.06 | 1 | 58.93* | 58.93 | 0.05 | 1 | 58.93* | 58.93 | 0.06 | 1 |
| lin318 | 766.83 | 766.83* | 766.83 | 0.20 | 2 | 766.83* | 766.83 | 0.17 | 2 | 766.83* | 766.83 | 0.19 | 2 | 766.83* | 766.83 | 0.19 | 1 |
| pcb442 | 83.72 | 83.72* | 83.72 | 0.27 | 1 | 83.72* | 83.72 | 0.27 | 1 | 83.72* | 83.72 | 0.24 | 1 | 83.72* | 83.72 | 0.23 | 1 |
| rat195 | 47.89 | 47.89* | 47.89 | 0.11 | 1 | 47.89* | 47.89 | 0.12 | 1 | 47.89* | 47.89 | 0.11 | 1 | 47.89* | 47.89 | 0.09 | 1 |
| rd400 | 539.95 | - | 450.76 | 2112.50 | 468581 | - | 450.77 | 1480.89 | 484005 | **452.16** | 451.16 | 14400.03 | 307612 | 452.17 | 450.95 | 14400.08 | 61323 |
| **Arbitrary radii** | | | | | | | | | | | | | | | | | |
| bonus1000rdmRad | 2689.41 | - | 573.36 | 2262.53 | 342657 | - | 612.78 | 1458.82 | 339795 | - | 486.80 | 3809.02 | 279132 | - | 468.84 | 4358.30 | 263705 |
| d493rdmRad | 761.07 | - | 436.96 | 1928.30 | 575283 | - | 444.04 | 1469.96 | 571860 | - | 427.71 | 3254.33 | 549361 | - | 436.52 | 12577.25 | 548975 |
| dsj1000rdmRad | 2074.84 | - | 680.45 | 1262.06 | 266737 | - | 704.03 | 812.75 | 265904 | 1863.36 | 622.25 | 1708.36 | 219512 | **1854.62** | 649.56 | 3545.89 | 190900 |
| kroD100rdmRad | 171.57 | - | 141.50 | 8812.96 | 3208345 | - | 150.28 | 10757.80 | 3125805 | **170.45** | 144.83 | 14400.02 | 1367174 | 171.08 | 143.15 | 14400.16 | 256696 |
| lin318rdmRad | 2189.43 | - | 1798.96 | 2168.43 | 718183 | - | 1856.27 | 1645.60 | 716217 | **2112.19** | 1825.87 | 10128.30 | 708096 | 2115.60 | 1825.35 | 14400.31 | 223856 |
| pcb442rdmRad | 258.40 | - | 175.89 | 1720.89 | 588912 | - | 180.15 | 901.65 | 578218 | 247.35 | 171.18 | 6294.70 | 543704 | **243.36** | 176.05 | 14400.81 | 366541 |
| rat195rdmRad | 84.47 | 82.10* | 82.10 | 253.21 | 18308 | 82.10* | 82.10 | 76.00 | 2710 | 82.10* | 82.10 | 83.98 | 2201 | 82.10* | 82.10 | 76.89 | 342 |
| rd400rdmRad | 3592.60 | - | 886.58 | 5625.26 | 1070428 | - | 896.19 | 5355.85 | 1077255 | - | 679.03 | 14400.51 | 769343 | - | 689.45 | 14400.22 | 928931 |
| team1_100rdmRad | 907.59 | - | 749.99 | 7262.76 | 3575899 | - | 764.37 | 5637.32 | 3399981 | **904.47** | 725.96 | 14400.12 | 1841884 | - | 716.63 | 14400.06 | 329303 |
| team2_200rdmRad | 1055.95 | - | 533.51 | 5376.88 | 1993918 | - | 553.84 | 3631.39 | 1931010 | - | 507.36 | 7395.64 | 1545275 | - | 516.92 | 14400.19 | 803425 |

(continued on next page)

Table 5.3: Results on 42 3D instances with known feasible solution (continued)

| | | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | Known | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| team3_300rdmRad | 1053.38 | - | 673.09 | 1855.88 | 770647 | - | 695.58 | 1339.38 | 755865 | **1009.72** | 655.81 | 3482.02 | 744912 | 1016.51 | 671.48 | 14400.02 | 461812 |
| team4_400rdmRad | 1276.90 | - | 553.54 | 3663.95 | 918157 | - | 575.32 | 2634.09 | 918596 | - | 520.22 | 8643.83 | 916999 | - | 541.50 | 14400.78 | 990125 |
| team5_499rdmRad | 840.48 | - | 589.56 | 1091.24 | 419857 | - | 605.46 | 741.88 | 417135 | 808.25 | 578.62 | 2256.96 | 414475 | **798.70** | 597.54 | 13845.83 | 380562 |
| team6_500rdmRad | 1076.35 | - | 503.12 | 1772.04 | 566822 | - | 529.74 | 1177.96 | 551044 | - | 486.65 | 2967.41 | 467703 | - | 507.34 | 8415.55 | 452690 |

### 5.5.3 Problems without feasible solutions

Coutinho et al. [15] demonstrates the impact of the overlap ratio by changing the overlap ratio on the *TSPLIB* instances (the ones with constant radii). Since all covering regions have the same radius, to obtain an instance with a particular overlap ratio $\lambda$, the radius on the new instance is the radius of the instance with overlap ratio 0.02 multiplied by $\lambda/0.02$. In particular, Coutinho et al. [15] examined overlap ratio from 0.1 to 0.3 for 4 of the TSPLIB 2D instances.

In this section, we examine a total of 56 2D instances based on all 7 TSPLIB instances with overlap ratios $0.04, 0.06, 0.08, 0.12, 0.14, 0.16, 0.18$ and 0.2. Not only are we interested to see the relation between solvability and overlap ratio on all 7 TSPLIB instances, more importantly, we can observe the performance of CBFS-CV and BFS when there is no best known solution to provide initial pruning for the algorithm, since best known solutions are not discussed in previous works on these instances.

Table 5.4 presents the results of all algorithms on the 56 instances. The column notations are the same as the ones in Table 5.2 with the exception that there is no column with best known solutions.

There are 4 instances, `kroD100` (overlap ratio 0.04), `d493` (overlap ratio 0.08), `dsj1000` (overlap ratio 0.08) and `rd400` (overlap ratio 0.14), where `Cout-CBFS+` and `Cout-CBFS-LAS+` are able to prove optimality while `Cout` and `Cout+` cannot. Furthermore, on instances not solved to optimality by any algorithms, `Cout-CBFS+` is the only one that can provide feasible solutions to all of them when it terminates.

As indicated in Coutinho et al. [15], as the overlap ratio increases, the problems become easier to solve. However, note that for `pcb442`, which is not examined with different overlap ratios in Coutinho et al. [15], the running time to prove optimality increases when overlap ratio increases from 0.16 to 0.18.

Table 5.4: Results on 56 instances with constant radii with no known feasible solution

| | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| **Constant radii, constant overlap ratios** | | | | | | | | | | | | | | | | |
| **overlap ratio: 0.04** | | | | | | | | | | | | | | | | |
| d493 | - | 133.30 | 1691.86 | 533076 | - | 133.29 | 1394.10 | 531037 | 156.77 | 131.29 | 3074.83 | 490816 | **155.96** | 132.49 | 14400.35 | 356004 |
| dsj1000 | - | 508.09 | 1210.81 | 266176 | - | 508.07 | 980.28 | 265465 | 705.62 | 490.64 | 2015.38 | 239368 | **690.10** | 497.86 | 7326.91 | 218037 |
| kroD100 | - | 127.22 | 5296.53 | 2487785 | - | 127.23 | 5243.76 | 2497457 | 127.76* | 127.76 | 8182.13 | 58766 | 127.76* | 127.76 | 12866.79 | 15344 |
| lin318 | - | 1836.76 | 2374.71 | 909457 | - | 1836.70 | 2017.16 | 907275 | 2271.69 | 1788.11 | 8151.77 | 804251 | **2268.98** | 1792.81 | 14400.41 | 356839 |
| pcb442 | - | 172.40 | 1903.07 | 625307 | - | 172.39 | 1458.06 | 624251 | 278.70 | 162.69 | 3397.38 | 566048 | **245.52** | 167.70 | 14400.53 | 466314 |
| rat195 | - | 97.40 | 2964.63 | 1310988 | - | 97.40 | 2422.22 | 1308211 | **118.07** | 94.75 | 14400.19 | 1193669 | 120.87 | 94.75 | 14400.49 | 387916 |
| rd400 | - | 530.60 | 2166.68 | 739035 | - | 530.58 | 1678.87 | 737757 | **840.76** | 499.16 | 4980.11 | 657334 | 887.80 | 507.50 | 12495.78 | 639163 |
| **overlap ratio: 0.06** | | | | | | | | | | | | | | | | |
| d493 | - | 121.18 | 1295.59 | 424728 | - | 121.17 | 962.76 | 422671 | **127.23** | 121.22 | 3914.94 | 416429 | 127.66 | 121.16 | 14400.25 | 170358 |
| dsj1000 | - | 462.12 | 927.77 | 214072 | - | 462.10 | 654.31 | 213399 | 495.33 | 465.28 | 4076.11 | 206389 | **493.73** | 467.38 | 14400.27 | 113036 |
| kroD100 | 109.35* | 109.35 | 513.93 | 243954 | 109.35* | 109.35 | 500.29 | 243990 | 109.35* | 109.35 | 450.09 | 3373 | 109.35* | 109.35 | 632.33 | 1117 |
| lin318 | - | 1679.82 | 1907.35 | 740725 | - | 1679.75 | 1469.38 | 738141 | **1829.59** | 1673.77 | 12110.44 | 734242 | 1840.44 | 1662.58 | 14400.00 | 191641 |
| pcb442 | - | 159.59 | 1650.71 | 584395 | - | 159.58 | 1232.92 | 582917 | 223.51 | 154.84 | 1683.97 | 500682 | **194.05** | 158.19 | 14400.19 | 385629 |
| rat195 | - | 86.21 | 2669.71 | 1194684 | - | 86.21 | 2101.54 | 1189943 | **92.89** | 85.72 | 14400.03 | 848779 | 92.98 | 85.05 | 14400.23 | 144044 |
| rd400 | - | 496.01 | 1859.90 | 652711 | - | 495.99 | 1395.67 | 651795 | **659.27** | 476.12 | 5145.25 | 631544 | 666.50 | 484.41 | 14400.04 | 532965 |
| **overlap ratio: 0.08** | | | | | | | | | | | | | | | | |
| d493 | - | 110.20 | 1528.22 | 402345 | - | 110.19 | 908.11 | 400029 | 110.64* | 110.64 | 8546.50 | 52089 | 110.64* | 110.64 | 12378.57 | 22743 |
| dsj1000 | - | 415.73 | 852.05 | 200574 | - | 415.72 | 553.08 | 199952 | 420.69* | 420.69 | 8514.69 | 39975 | 420.69* | 420.69 | 12043.41 | 11901 |
| kroD100 | 96.90* | 96.90 | 8.97 | 4938 | 96.90* | 96.90 | 9.10 | 4938 | 96.90* | 96.90 | 8.75 | 258 | 96.90* | 96.90 | 19.76 | 387 |
| lin318 | - | 1535.86 | 1741.38 | 703042 | - | 1535.81 | 1326.30 | 699684 | **1568.14** | 1547.61 | 14400.03 | 407421 | **1568.14** | 1537.18 | 14400.29 | 76506 |
| pcb442 | - | 147.30 | 1440.67 | 519032 | - | 147.29 | 1017.45 | 515956 | 167.28 | 145.69 | 2456.55 | 484623 | **165.71** | 146.98 | 14400.20 | 278659 |
| rat195 | - | 75.92 | 2658.77 | 1151349 | - | 75.92 | 2034.74 | 1143496 | **76.72** | 76.24 | 14400.02 | 238720 | **76.72** | 75.89 | 14400.00 | 55275 |
| rd400 | - | 462.26 | 1690.21 | 618218 | - | 462.24 | 1246.74 | 616237 | **535.22** | 454.90 | 7104.89 | 605281 | 535.26 | 456.64 | 14400.09 | 272860 |
| **overlap ratio: 0.12** | | | | | | | | | | | | | | | | |
| d493 | 94.42* | 94.42 | 1.24 | 341 | 94.42* | 94.42 | 1.85 | 341 | 94.42* | 94.42 | 1.80 | 212 | 94.42* | 94.42 | 1.50 | 72 |
| dsj1000 | 336.34* | 336.34 | 321.31 | 72160 | 336.34* | 336.34 | 203.63 | 72142 | 336.34* | 336.34 | 204.44 | 7122 | 336.34* | 336.34 | 306.82 | 1818 |
| kroD100 | 85.26* | 85.26 | 0.46 | 201 | 85.26* | 85.26 | 0.49 | 201 | 85.26* | 85.26 | 0.52 | 87 | 85.26* | 85.26 | 0.51 | 1 |
| lin318 | 1249.01* | 1249.01 | 467.39 | 137741 | 1249.01* | 1249.01 | 357.01 | 90816 | 1249.01* | 1249.01 | 338.59 | 3189 | 1249.01* | 1249.01 | 434.61 | 1393 |

(continued on next page)

Table 5.4: Results on 56 instances with constant radii with no known feasible solution (continued)

| | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| pcb442 | - | 125.94 | 1211.99 | 440458 | - | 125.93 | 799.71 | 438566 | 135.02 | 125.63 | 2216.06 | 435527 | **132.34** | 125.92 | 14400.50 | 199123 |
| rat195 | 64.90* | 64.90 | 0.12 | 16 | 64.90* | 64.90 | 0.32 | 16 | 64.90* | 64.90 | 0.64 | 129 | 64.90* | 64.90 | 0.35 | 1 |
| rd400 | - | 396.44 | 1386.20 | 508988 | - | 396.42 | 930.67 | 505634 | **403.98** | 401.23 | 14400.13 | 352836 | **403.98** | 399.26 | 14400.31 | 72517 |
| **overlap ratio: 0.14** | | | | | | | | | | | | | | | | |
| d493 | 90.01* | 90.01 | 0.39 | 58 | 90.01* | 90.01 | 0.88 | 58 | 90.01* | 90.01 | 1.50 | 65 | 90.01* | 90.01 | 0.87 | 2 |
| dsj1000 | 309.20* | 309.20 | 36.73 | 6406 | 309.20* | 309.20 | 25.02 | 5150 | 309.20* | 309.20 | 45.17 | 2400 | 309.20* | 309.20 | 55.33 | 368 |
| kroD100 | 81.67* | 81.67 | 0.06 | 7 | 81.67* | 81.67 | 0.09 | 7 | 81.67* | 81.67 | 0.13 | 43 | 81.67* | 81.67 | 0.09 | 1 |
| lin318 | 1131.29* | 1131.29 | 106.34 | 26420 | 1131.29* | 1131.29 | 76.83 | 23896 | 1131.29* | 1131.29 | 75.02 | 1007 | 1131.29* | 1131.29 | 120.94 | 735 |
| pcb442 | - | 116.86 | 1192.28 | 434833 | - | 116.86 | 757.91 | 432891 | **118.28** | 117.39 | 6695.34 | 426840 | 118.47 | 117.21 | 14400.39 | 101638 |
| rat195 | 62.54* | 62.54 | 0.10 | 1 | 62.54* | 62.54 | 0.16 | 1 | 62.54* | 62.54 | 0.17 | 1 | 62.54* | 62.54 | 0.17 | 1 |
| rd400 | - | 365.65 | 1428.31 | 471820 | - | 365.65 | 913.41 | 469228 | 366.59* | 366.59 | 6358.88 | 27903 | 366.59* | 366.59 | 8821.00 | 9356 |
| **overlap ratio: 0.16** | | | | | | | | | | | | | | | | |
| d493 | 86.93* | 86.93 | 0.29 | 16 | 86.93* | 86.93 | 0.56 | 16 | 86.93* | 86.93 | 2.58 | 41 | 86.93* | 86.93 | 0.50 | 1 |
| dsj1000 | 285.90* | 285.90 | 5.94 | 1807 | 285.90* | 285.90 | 5.74 | 990 | 285.90* | 285.90 | 6.42 | 891 | 285.90* | 285.90 | 15.95 | 236 |
| kroD100 | 78.55* | 78.55 | 0.07 | 7 | 78.55* | 78.55 | 0.37 | 7 | 78.55* | 78.55 | 0.28 | 14 | 78.55* | 78.55 | 0.30 | 1 |
| lin318 | 1039.72* | 1039.72 | 17.61 | 2374 | 1039.72* | 1039.72 | 15.74 | 1809 | 1039.72* | 1039.72 | 16.58 | 282 | 1039.72* | 1039.72 | 28.53 | 102 |
| pcb442 | 108.82* | 108.82 | 1091.22 | 263109 | 108.82* | 108.82 | 830.18 | 262532 | 108.82* | 108.82 | 795.53 | 7426 | 108.82* | 108.82 | 1006.55 | 2126 |
| rat195 | 60.29* | 60.29 | 0.10 | 1 | 60.29* | 60.29 | 0.18 | 1 | 60.29* | 60.29 | 0.11 | 1 | 60.29* | 60.29 | 0.18 | 1 |
| rd400 | 335.11* | 335.11 | 513.50 | 35588 | 335.11* | 335.11 | 447.99 | 35588 | 335.11* | 335.11 | 468.58 | 3204 | 335.11* | 335.11 | 592.97 | 1558 |
| **overlap ratio: 0.18** | | | | | | | | | | | | | | | | |
| d493 | 84.38* | 84.38 | 0.27 | 7 | 84.38* | 84.38 | 0.97 | 7 | 84.38* | 84.38 | 3.09 | 7 | 84.38* | 84.38 | 0.95 | 1 |
| dsj1000 | 265.28* | 265.28 | 3.11 | 742 | 265.28* | 265.28 | 3.75 | 297 | 265.28* | 265.28 | 4.73 | 543 | 265.28* | 265.28 | 3.66 | 26 |
| kroD100 | 75.51* | 75.51 | 0.07 | 7 | 75.51* | 75.51 | 0.09 | 7 | 75.51* | 75.51 | 0.13 | 34 | 75.51* | 75.51 | 0.10 | 1 |
| lin318 | 978.27* | 978.27 | 0.19 | 16 | 978.27* | 978.27 | 0.32 | 16 | 978.27* | 978.27 | 1.38 | 119 | 978.27* | 978.27 | 0.34 | 1 |
| pcb442 | 102.32* | 102.32 | 1652.86 | 32330 | 102.32* | 102.32 | 1481.48 | 31996 | 102.32* | 102.32 | 1569.19 | 16653 | 102.32* | 102.32 | 1941.64 | 5072 |
| rat195 | 58.08* | 58.08 | 0.10 | 1 | 58.08* | 58.08 | 0.18 | 1 | 58.08* | 58.08 | 0.14 | 1 | 58.08* | 58.08 | 0.17 | 1 |
| rd400 | 306.62* | 306.62 | 6.49 | 2610 | 306.62* | 306.62 | 4.73 | 2610 | 306.62* | 306.62 | 6.84 | 392 | 306.62* | 306.62 | 12.19 | 196 |
| **overlap ratio: 0.20** | | | | | | | | | | | | | | | | |
| d493 | 81.86* | 81.86 | 0.27 | 7 | 81.86* | 81.86 | 1.65 | 7 | 81.86* | 81.86 | 2.03 | 4 | 81.86* | 81.86 | 1.89 | 1 |

(continued on next page)

Table 5.4: Results on 56 instances with constant radii with no known feasible solution (continued)

| | Cout | | | | Cout+ | | | | Cout-CBFS+ | | | | Cout-CBFS-LAS+ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp | UB | LB | Time | Unexp |
| dsj1000 | 251.68* | 251.68 | 0.85 | 106 | 251.68* | 251.68 | 1.22 | 106 | 251.68* | 251.68 | 1.64 | 381 | 251.68* | 251.68 | 1.29 | 7 |
| kroD100 | 72.54* | 72.54 | 0.06 | 4 | 72.54* | 72.54 | 0.10 | 4 | 72.54* | 72.54 | 0.11 | 27 | 72.54* | 72.54 | 0.09 | 1 |
| lin318 | 937.84* | 937.84 | 0.18 | 4 | 937.84* | 937.84 | 0.32 | 4 | 937.84* | 937.84 | 0.34 | 4 | 937.84* | 937.84 | 0.36 | 1 |
| pcb442 | 97.99* | 97.99 | 0.47 | 84 | 97.99* | 97.99 | 0.83 | 84 | 97.99* | 97.99 | 1.19 | 268 | 97.99* | 97.99 | 0.89 | 4 |
| rat195 | 55.91* | 55.91 | 0.11 | 1 | 55.91* | 55.91 | 0.21 | 1 | 55.91* | 55.91 | 0.20 | 1 | 55.91* | 55.91 | 0.25 | 1 |
| rd400 | 285.99* | 285.99 | 4.63 | 1969 | 285.99* | 285.99 | 3.90 | 1913 | 285.99* | 285.99 | 6.63 | 219 | 285.99* | 285.99 | 7.44 | 108 |

### 5.5.4 Discussion

There are several observations we can make based on the numerical experiments in this section.

First, the new branching vertex selection scheme simplifies the algorithm by having a single scheme for both constant radii and arbitrary radii instances. For the arbitrary radii instances, where the new scheme may select different vertices from the scheme in the original `Cout`, the new scheme outperforms the original one in terms of running time in tested instances.

Second, the two improvement methods discussed in Sections 5.4.2 and 5.4.3 help further improve the running time of the algorithm. Since BFS focuses on improving the lower bound and only finds solutions very late in the search, the reduction in vertex coverage check discussed in Section 5.4.2 is the main reason for the improvement when comparing `Cout+` with `Cout`.

On `Cout-CBFS+` and `Cout-CBFS-LAS+`, the feasible solution improvement method discussed in Section 5.4.3 can reduce the running time on instances where the best known solution value is not optimal. By improving the quality of feasible solutions found, it helps to reduce the number of nodes with lower bound no less than the optimal solution the CBFS-CV selects for exploration. However, since the number of nodes that must be explored (with lower bound less than the optimal solution value) is often very large, the reduction in vertex coverage check is still very important in reducing the running time of the algorithms.

Third, although CBFS-CV does not lead to substantial reduction in running time compared with BFS, the benefit of CBFS-CV is obvious. `Cout-CBFS+` stores fewer (sometimes significantly fewer) unexplored nodes than `Cout+`, while frequently finding good feasible solutions early in the search. However, if improving the global lower

bound is the priority, BFS is a better choice than CBFS-CV.

Finally, LAS can be very useful when space constraints are important, at the expense of longer running times.

## 5.6 Conclusion and future research

In this chapter, we presented an improved version of `Cout`, originally developed by Coutinho et al. [15], to solve the CETSP. A new variation of CBFS is proposed for `Cout` to address the disadvantages of BFS used in `Cout`. A branching vertex selection scheme that works for both the constant radii and arbitrary radii instances is presented. Additionally, we proposed methods to reduce computational workload and improve solution quality. Finally, the implementation of `Cout` by the original authors is improved.

The improved algorithm is tested on a set of 160 instances in total (104 instances with known feasible solutions and 56 instances without). Overall, it shows that the improved algorithm is comparable or slightly better on most instances that can be solved to optimality while using less space for node storage. On the instances where optimality cannot be proven due to time or space constraints, the improved algorithm tends to find better feasible solutions. On instances with arbitrary radii, the new branching vertex selection scheme outperforms `Cout` and a previously unsolved instance is solved to optimality for the first time. Additionally, since the improved algorithm can find good feasible solutions quickly, which reduces the number of nodes that has to be stored and maintained, when a good feasible solution is not provided at the beginning of the search, the improved algorithm solves more problems to optimality than `Cout`.

One future research direction is finding algorithms that can be used to compute good lower bounds to the problem, given some partial sequences. For the algorithms

in this chapter, a significant percentage of the iterations are processing partial sequences that do not lead to optimal or even good solutions. A better lower bound algorithm would reduce the number of iterations substantially which also reduces the number of nodes to store, removing an obstacle that prevents B&B algorithms from solving larger problems.

# Chapter 6

# Conclusion

In this dissertation, we study the CBFS strategy in B&B algorithms by applying variants of it to B&B algorithms for specific problems as well as by investigating some underlying conditions of the search tree that allows a CBFS variant, CBFS-depth, to explore fewer nodes than BFS to prove optimality in a B&B algorithm. We find that CBFS can be very effective, given the right contour labeling function for the right problem.

We first apply a known CBFS variant, CBFS-depth on a one machine scheduling problem with release and delivery times with the minimum makespan objective $(1|r_i, q_i, dpc|C_{max})$ and a more generalized version of the problem with delayed precedence constraints $(1|r_i, q_i, dpc|C_{max})$. In addition to the use of CBFS-depth as search strategy, we also propose a modified version of the heuristic `LTH`. In the new heuristic `MLTH`, we allow some jobs that are not yet released to be considered for scheduling as we build a feasible sequence for the problem. A rescheduling process is presented to make sure that schedules created by `MLTH` are valid in the branching process. The behavior of BFS with both FIFO and LIFO tie-breaking rules on the problem is discussed as well as the reason for choosing CBFS-depth. In particular, we show that CBFS-depth

can be better at balancing the diversity and intensity of the search. Computational results are reported showing that `LDepth`, which includes both `MLTH` and CBFS-depth outperforms `Bal` on both $1|r_i, q_i|C_{max}$ and $1|r_i, q_i, dpc|C_{max}$. We show that for the instances of $1|r_i, q_i|C_{max}$, `MLTH` contributes to the order of magnitude improvement over the state-of-the-art algorithm. On instances of $1|r_i, q_i, dpc|C_{max}$, the heuristic also improves the running time and the number of nodes explored from the original algorithm, but CBFS-depth helps when solving some instances that cannot be solved with BFS, with or without `MLTH`.

We then examine the performance of CBFS-depth in a non-problem-specific manner by studying the conditions of the search trees on which CBFS-depth explores fewer nodes than BFS to prove optimality. We first showed that, without sophisticated pruning rules beyond the lower bound, optimal solutions must come from nodes with $f_{LB} = Z^*$ in order for CBFS-depth to have an opportunity to explore fewer nodes than BFS. Based on the observation, we present a set of assumptions that aim to distinguish only three types of nodes in the search tree based on the relation between their lower bounds and the optimal solution value. In particular, the $E$-type, $L$-type and $G$-type nodes. A search tree model for B&B algorithms based on the distribution of the three types of nodes with different lower bounds is proposed so that we can use randomly generated search trees to study the performance of search strategies CBFS-depth and BFS in terms of the number of nodes explored to prove optimality. An estimation of the expected number of nodes explored by CBFS-depth based on the parameters of the search tree is proposed. Generated search trees and several problems are used to demonstrate the search tree conditions under which CBFS-depth is a better choice than BFS. In particular, the results show that one key to the success of CBFS-depth is the large number of nodes with $f_{LB} = Z^*$ on the levels where an optimal solution may exist. We then justify the observation with problem instances from $1|r_i, q_i, dpc|C_{max}$

discussed in the previous chapter as well as the MIPLIB library.

Finally, we present another scenario where a variant of CBFS can be a better search strategy than BFS by applying CBFS-CV to a B&B algorithm for CETSP. In particular, we show that even if CBFS does not explore fewer nodes than BFS, it can still be a better choice of search strategy given the right labeling function and problem. We improve a B&B algorithm proposed by Coutinho et al. [15] by using CBFS-CV and other methods that reduce computational workload and improve solution quality. We discuss the problem structure, specifically the weak lower bound and the lack of sophisticated pruning methods, that allow CBFS-CV to have two advantages over BFS: it requires less memory for storing unexplored nodes, and it finds good solutions substantially earlier than BFS in the search, which is confirmed than by our computational experiments. On problem instances that can be solved, CBFS-CV does not explore many more nodes than BFS and is mostly comparable with BFS in terms of running time with substantially fewer nodes to store. On problem instances that cannot be solved with the given time and space limit, CBFS-CV has the clear advantage of obtaining feasible solutions, sometimes better than the best known feasible solutions found by other heuristics, before termination while BFS terminates without finding any solution.

This dissertation presents both the examples of using the CBFS strategy to improve a B&B algorithm as well as the discussions on the reason behind the improvements. It aims to provide a different approach in an aspect of the B&B algorithm that is frequently overlooked, the search strategy. In particular, we demonstrate that, by using CBFS variants, we can guide the search away from large numbers of nodes that may not lead to good or optimal solutions. In the one machine problems, BFS is unable to distinguish nodes with the same lower bound; in CETSP, BFS has no capacity to move away from many nodes with small lower bounds, even though feasible solutions cannot be derived from them. Therefore, any class of problems that has large

numbers of nodes that are unlikely to generate good solutions while the commonly used search strategies such as BFS are unable to lead the search away from those nodes, are candidates for CBFS variants. As observed by Morrison et al. [34], many scheduling problems fit this description, which is why CBFS-depth has been used on a number of them [23, 38, 39, 32]. However, as indicated with the use of a different CBFS variant in Morrison et al. [33], as well as the use of CBFS-CV in CETSP, on many other optimization problems that fit the description, a CBFS variant that takes advantage of some problem-specific structure may result in better performance in terms of finding solutions early.

The author will provide all codes used in the paper along with data and results to interested readers and can be reached at `zwd.kid@gmail.com`.

# Bibliography


[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technical University of Berlin, 2007.

[2] T. Achterberg and R. Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of combinatorial optimization*, pages 449–481. Springer, 2013.

[3] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.

[4] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde. Available at: `http://www.math.uwaterloo.ca/tsp/concorde/index.html`, 2003.

[5] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study.* Princeton university press, 2006.

[6] K. R. Baker and Z. S. Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics Quarterly*, 21(1):171–176, 1974.

[7] E. Balas, J. K. Lenstra, and A. Vazacopoulos. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science*, 41(1):94–109, 1995.

[8] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012.

[9] B. Behdani and J. C. Smith. An integer-programming-based approach to the close-enough traveling salesman problem. *INFORMS Journal on Computing*, 26 (3):415–432, 2014.

[10] Nicolas Bourgeois, R. Catellier, Tom Denat, and Vangelis Th. Paschos. Average-case complexity of a branch-and-bound algorithm for maximum independent set, under the $\mathcal{G}(n, p)$ random model. *ArXiv*, abs/1505.04969, 2015.

[11] C. Briand, S. Ourari, and B. Bouzouia. An efficient ilp formulation for the single machine scheduling problem. *RAIRO-Operations Research*, 44(1):61–71, 2010.

[12] Leandro C. C. Install and run concorde with cplex. `https://www.leandro-coelho.com/install-and-run-concorde-with-cplex/`, 2019.

[13] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42–47, 1982.

[14] F. Carrabs, C. Cerrone, R. Cerulli, and M. Gaudioso. A novel discretization scheme for the close enough traveling salesman problem. *Computers & Operations Research*, 78:163–171, 2017.

[15] W. P. Coutinho, R. Q. do Nascimento, A. A. Pessoa, and A. Subramanian. A branch-and-bound algorithm for the close-enough traveling salesman problem. *INFORMS Journal on Computing*, 28(4):752–765, 2016.

[16] S. Dauzere-Peres and J-B Lasserre. A modified shifting bottleneck procedure for

job-shop scheduling. *The International Journal of Production Research*, 31(4): 923–932, 1993.

[17] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[18] E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[19] J. Dong, N. Yang, and M. Chen. Heuristic approaches for a tsp variant: The automatic meter reading shortest tour problem. In *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, pages 145–163. Springer, 2007.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, NY, USA, 1979.

[21] A. Gharbi and M. Labidi. Jackson's semi-preemptive scheduling on a single machine. *Computers & Operations Research*, 37(12):2082–2088, 2010.

[22] D. J. Gulczynski, J. W. Heath, and C. C. Price. The close enough traveling salesman problem: A discussion of several heuristics. In *Perspectives in Operations Research*, pages 271–283. Springer, 2006.

[23] 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*, 12(2):163–175, 2009.

[24] H. Kise and M. Uno. One-machine scheduling problems with earliest start and due time constraints. *Mem. Kyoto Tech. Univ. Sci. Tech*, 27:25–34, 1978.

[25] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[26] B. J. Lageweg, J. K. Lenstra, and A. R. Kan. Minimizing maximum lateness on one machine: computational experience and some applications. *Statistica Neerlandica*, 30(1):25–41, 1976.

[27] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrics*, 1960.

[28] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11 (2):173, 1999.

[29] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations research*, 23(3):475–482, 1975.

[30] W. Mennell. *Heuristics for solving three routing problems: Close-enough traveling salesman problem, close-enough vehicle routing problem, sequence-dependent team orienteering problem.* PhD thesis, 2009.

[31] W. Mennell, B. Golden, and E. Wasil. A steiner-zone heuristic for solving the close-enough traveling salesman problem. In *2th INFORMS computing society conference: operations research, computing, and homeland defense*, 2011.

[32] D. R. Morrison, E. C. Sewell, and S. H. Jacobson. An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. *European Journal of Operational Research*, 236(2):403–409, 2014.

[33] D. R. Morrison, E. C. Sewell, and S. H. Jacobson. Solving the pricing problem in a branch-and-price algorithm for graph coloring using zero-suppressed binary decision diagrams. *INFORMS Journal on Computing*, 28(1):67–82, 2016.

[34] D. R. Morrison, J. J. Sauppe, W. Zhang, S. H. Jacobson, and E. C. Sewell. Cyclic best first search: Using contours to guide branch-and-bound algorithms. *Naval Research Logistics (NRL)*, 64(1):64–82, 2017.

[35] Y. Pan and L. Shi. Branch-and-bound algorithms for solving hard instances of the one-machine sequencing problem. *European Journal of Operational Research*, 168(3):1030–1039, 2006.

[36] C. N. Potts. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):1436–1441, 1980.

[37] R. Sadykov and A. Lazarev. Experimental comparison of branch-and-bound algorithms for the $1|r_j|l_{max}$ problem. In *Proceedings of the seventh international workshop MAPSP*, volume 5, pages 239–41, 2005.

[38] E. C. Sewell and S. H. Jacobson. A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing*, 24(3):433–442, 2012.

[39] E. C. Sewell, J. J. Sauppe, D. R. Morrison, S. H. Jacobson, and G. Kao. A BB&R algorithm for minimizing total tardiness on a single machine with sequence dependent setup times. *Journal of Global Optimization*, 54(4):791–812, Dec 2012.

[40] K. Sourirajan and R. Uzsoy. Hybrid decomposition heuristics for solving large-scale scheduling problems in semiconductor wafer fabrication. *Journal of Scheduling*, 10(1):41–65, 2007.

[41] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[42] N. R. Vempaty, V. Kumar, and R. E. Korf. Depth-first versus best-first search. In *AAAI*, 1991.

[43] X. Wang, B. Golden, and E. Wasil. A steiner zone variable neighborhood search heuristic for the close-enough traveling salesman problem. *Computers & Operations Research*, 101:200–219, 2019.

[44] Z. Yang, M. Q. Xiao, Y. W. Ge, D. L. Feng, L. Zhang, H. F. Song, and X. L. Tang. A double-loop hybrid algorithm for the traveling salesman problem with arbitrary neighbourhoods. *European Journal of Operational Research*, 265(1):65–80, 2018.

[45] B. Yuan, M. Orlowska, and S. Sadiq. On the optimal robot routing problem in wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1252–1261, 2007.

[46] W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241–292, 1995.

[47] W. Zhang, J. J. Sauppe, and S. H. Jacobson. An improved branch-and-bound algorithm for the one-machine problem and delayed precedence constraints variation. Technical report, 2018.

[48] W. Zhang, J. J. Sauppe, and S. H. Jacobson. A branch-and-bound algorithm for the close-enough traveling salesman problem. Technical report, 2020.

# Appendix A

# Additional Results: MLTH

The following tables are additional results on MLTH:

Table A.1: Ratio of MLTH used in all iterations on $1|r_i, q_i|C_{max}$ instances

| No. Jobs | $k = 15$ | $k = 20$ | $k = 25$ |
|---|---|---|---|
| $n = 50$ | 0.4760 | 0.8635 | 0.8179 |
| $n = 100$ | 0.5890 | 0.8630 | 0.8605 |
| $n = 200$ | 0.7397 | 0.8726 | 0.8675 |
| $n = 500$ | 0.8646 | 0.8658 | 0.8792 |
| $n = 1000$ | 0.8558 | 0.9204 | 0.8668 |

Table A.2: Ratio of MLTH used in all iterations on $1|r_i, q_i, dpc|C_{max}$ instances

| No. Jobs | $k = 10, p = 0.01$ | $k = 10, p = 0.02$ | $k = 15, p = 0.01$ | $k = 15, p = 0.02$ |
|---|---|---|---|---|
| $n = 50$ | 0.4590 | 0.7268 | 0.7839 | 0.6763 |
| $n = 100$ | 0.7536 | 0.6017 | 0.7086 | 0.5813 |

Table A.3: Percentage difference from optimal solutions on $1|r_i, q_i|C_{max}$ instances

| | $k = 15$ | | $k = 20$ | | $k = 25$ | |
|---|---|---|---|---|---|---|
| No. Jobs | LTH | MLTH | LTH | MLTH | LTH | MLTH |
| $n = 50$ | 0.93% | 0.67% | 0.93% | 0.21% | 0.70% | 0.15% |
| $n = 100$ | 0.54% | 0.30% | 0.45% | 0.10% | 0.35% | 0.06% |
| $n = 200$ | 0.28% | 0.11% | 0.22% | 0.04% | 0.17% | 0.03% |
| $n = 500$ | 0.12% | 0.03% | 0.08% | 0.02% | 0.07% | 0.01% |
| $n = 1000$ | 0.05% | 0.01% | 0.04% | 0.01% | 0.03% | 0.01% |

Table A.4: Percentage difference from optimal solutions on $1|r_i, q_i, dpc|C_{max}$ instances

| | $k = 10, p = 0.01$ | | $k = 10, p = 0.02$ | | $k = 15, p = 0.01$ | | $k = 15, p = 0.02$ | |
|---|---|---|---|---|---|---|---|---|
| No. Jobs | LTH | MLTH | LTH | MLTH | LTH | MLTH | LTH | MLTH |
| $n = 50$ | 2.73% | 3.65% | 2.03% | 1.17% | 1.40% | 0.67% | 0.85% | 0.48% |
| $n = 100$ | 0.83% | 0.46% | 0.33% | 0.26% | 0.37% | 0.21% | 0.14% | 0.11% |

# Appendix B

# Compute $\hat{\tau}_{i,j,l,e,g}$

Let the probabilities $\hat{P}_L^{i,j}$, $\hat{P}_E^{i,j}$ and $\hat{P}_G^{i,j}$ be the estimated probabilities that an $L$-type, an $E$-type and a $G$-type node is explored in depth level $i$ cycle $j$. They can be expressed using $\hat{\tau}$

$$\begin{aligned}
\hat{P}_L^{i,j} &= 1 - \hat{\tau}_{i,j,0,*,*} \\
\hat{P}_E^{i,j} &= \hat{\tau}_{i,j,0,*,*} - \hat{\tau}_{i,j,0,0,g_{max}^{i,j}} \\
\hat{P}_G^{i,j} &= \hat{\tau}_{i,j,0,0,g_{max}^{i,j}},
\end{aligned}$$

where $g_{max}^{i,j}$ is the maximum number of unprocessed nodes in depth level $i$ cycle $j$, and $\hat{\tau}_{i,j,0,*,*}$ is the estimated probability that $N_L^{i,j} = 0$.

First, we discuss the situation when $j = 1$. Since there are no nodes in each depth level before the first cycle, the probability of level $i > 1$ with 2 $L$-type, $E$-type and

$G$-type nodes can be written as

$$\hat{\tau}_{i,1,2,0,0} = \hat{P}_L^{i-1,1} q_{i-1}$$
$$\hat{\tau}_{i,1,0,2,0} = \hat{P}_E^{i-1,1}(1 - r_{i-1}) + \hat{P}_L^{i-1,1}(1 - q_{i-1} - r_{i-1})$$
$$\hat{\tau}_{i,1,0,0,2} = \hat{P}_G^{i-1,1} + (1 - \hat{P}_G^{i-1,1}) r_{i-1}.$$

Second, we discuss the situation when $i > 2$ and $2 \leq j \leq 2^{i-2}$. For all depth levels $i > 2$, if $j \leq 2^{i-2}$, the previous depth level $i-1$ has at least one unprocessed node in cycle $j$ which leads to two new unprocessed nodes generated in depth level $i$. There should be exactly $j+1$ unprocessed nodes in depth level $i$ cycle $j$ before a node from the depth level is selected and thus $\tau_{i,j,l,e,g} = 0$ for any $l + e + g \neq j + 1$.

We distinguish several different cases to calculate $\hat{\tau}_{i,j,l,e,g}$, based on the values of $l$, $e$ and $g$. Since we know the number of nodes in depth level $i$ cycle $j$ is $j+1$, only two of $l$, $g$ and $e$ are needed to determine the combination of nodes. In particular, we use the value of $l$ and $g$ to determine the cases, and the value of $e$ is implied.

**Case 1:** $l = 0$ and $g \leq j - 2$. Given $l = 0$, $N_L^{i,j-1}$ can be 0 or 1 in cycle $j-1$ and the new unprocessed nodes generated in the $j^{th}$ cycle in depth level $i$ are not of $L$-type. On the other hand, given $g \leq j-2$, $N_G^{i,j-1}$ can be $g-2$ or $g$. If $N_G^{i,j-1} = g-2$, the new unprocessed nodes generated in depth level $i$ cycle $j$ have to be $G$-type nodes. If $N_G^{i,j-1} = g$, the new nodes have to be $E$-type nodes.

Therefore, we have the following expression for $\hat{\tau}_{i,j,0,e,g}$

$$\begin{aligned}\hat{\tau}_{i,j,0,e,g} = &(\hat{\tau}_{i,j-1,0,e-1,g} + \hat{\tau}_{i,j-1,1,e-2,g}) \\ &\cdot \left(\hat{P}_E^{i-1,j}(1 - r_{i-1}) + \hat{P}_L^{i-1,j}(1 - q_{i-1} - r_{i-1})\right) \\ &+ \hat{\tau}_{i,j-1,0,e+1,l-2} \cdot \left(\hat{P}_G^{i-1,j} + (1 - \hat{P}_G^{i-1,j}) r_{i-1}\right).\end{aligned} \tag{B.1}$$

**Case 2:** $l = 0$ and $g = j - 1$. Given $g = j - 1$, $N_G^{i,j-1}$ can be $j - 3$, $j - 1$ and $j$. In particular, there are $j$ unprocessed nodes depth level $i$ cycle $j-1$, and if all unprocessed nodes are $G$-type nodes, one $G$-type node is explored. This case is otherwise the same as Case 1. Therefore, we can write $\hat{\tau}_{i,j,0,2,j-1}$ as

$$\begin{aligned}\hat{\tau}_{i,j,0,2,j-1} = &(\hat{\tau}_{i,j-1,0,1,j-1} + \hat{\tau}_{i,j-1,1,0,j-1} + \hat{\tau}_{i,j-1,0,0,j}) \\ &\cdot \left(\hat{P}_E^{i-1,j}(1 - r_{i-1}) + \hat{P}_L^{i-1,j}(1 - q_{i-1} - r_{i-1})\right) \\ &+ \hat{\tau}_{i,j-1,0,3,j-3} \cdot \left(\hat{P}_G^{i-1,j} + (1 - \hat{P}_G^{i-1,j})r_{i-1}\right).\end{aligned} \tag{B.2}$$

**Case 3:** $l = 0$ and $g = j$. Given $g = j$, $N_G^{i,j-1}$ can only be $j - 2$. Therefore, similar to the scenario in Case 1 when $N_G^{i,j-1} = g - 2$, we can write $\hat{\tau}_{i,j,0,1,j}$ as

$$\hat{\tau}_{i,j,0,1,j} = \hat{\tau}_{i,j-1,0,2,j-2} \cdot \left(\hat{P}_G^{i-1,j} + (1 - \hat{P}_G^{i-1,j})r_{i-1}\right). \tag{B.3}$$

**Case 4:** $l = 0$ and $g = j + 1$. Given $g = j + 1$, $N_G^{i,j-1}$ can be $j - 1$ and $j$. If $N_G^{i,j-1} = j - 1$, the new unprocessed nodes generated in depth level $i$ cycle $j$ must be $G$-type nodes. If $N_G^{i,j-1} = j$, a $G$-type node is explored in depth level $i$ cycle $j - 1$ and the new nodes should also be $G$-type nodes. We can write $\hat{\tau}_{i,j,0,0,j+1}$ as

$$\hat{\tau}_{i,j,0,0,j+1} = (\hat{\tau}_{i,j-1,0,1,j-1} + \hat{\tau}_{i,j-1,1,0,j-1} + \hat{\tau}_{i,j-1,0,0,j}) \cdot \left(\hat{P}_G^{i-1,j} + (1 - \hat{P}_G^{i-1,j})r_{i-1}\right). \tag{B.4}$$

**Case 5:** $l = 1$ and $g \leq j$. Given $l = 1$, $N_L^{i,j-1}$ can only be 2. The new nodes generated in depth level $i$ cycle $j$ are $G$-type nodes if $N_G^{i,j-1} = g - 2$ and $E$-type nodes

if $N_G^{i,j-1} = g$. We can write $\hat{\tau}_{i,j,1,e,g}$ as

$$\begin{aligned}\hat{\tau}_{i,j,1,e,g} =& \hat{\tau}_{i,j-1,2,e-2,g} \\ & \cdot \left(\hat{P}_E^{i-1,j}(1-r_{i-1}) + \hat{P}_L^{i-1,j}(1-q_{i-1}-r_{i-1})\right) \\ & + \hat{\tau}_{i,j-1,2,e,g-2} \cdot \left(\hat{P}_G^{i-1,j} + (1-\hat{P}_G^{i-1,j})r_{i-1}\right).\end{aligned} \tag{B.5}$$

**Case 6:** $l = 2$ and $g \leq j-1$. Given $l = 2$, $N_L^{i,j-1}$ can be 0, 1 and 2. If $l = 0$ or 1, the new nodes generated in depth level $i$ cycle $j$ have to be $L$-type nodes. Otherwise, the generated nodes are $G$-type nodes if $N_G^{i,j-1} = g-2$ and $E$-type nodes if $N_G^{i,j-1} = g$. We can write $\hat{\tau}_{i,j,2,e,g}$ as

$$\begin{aligned}\hat{\tau}_{i,j,2,e,g} =& \hat{\tau}_{i,j-1,3,e-2,g} \\ & \cdot \left(\hat{P}_E^{i-1,j}(1-r_{i-1}) + \hat{P}_L^{i-1,j}(1-q_{i-1}-r_{i-1})\right) \\ & + (\hat{\tau}_{i,j-1,0,e+1,g} + \hat{\tau}_{i,j-1,1,j-g-1,g}) \cdot \hat{P}_L^{i-1,j} q_{i-1} \\ & + \hat{\tau}_{i,j-1,3,e,g-2} \cdot \left(\hat{P}_G^{i-1,j} + (1-\hat{P}_G^{i-1,j})r_{i-1}\right).\end{aligned} \tag{B.6}$$

**Case 7:** $l \geq 2$ and $g \leq j+1-l$. Since $l \geq 2$, $N_L^{i,j-1}$ can be $l-1$ and $l+1$. Since $g \leq j+1-l$, $N_G^{i,j-1}$ can be $g-2$ or $g$. We can write $\hat{\tau}_{i,j,l,e,g}$ as

$$\begin{aligned}\hat{\tau}_{i,j,l,e,g} =& \hat{\tau}_{i,j-1,l+1,e-2,g} \\ & \cdot \left(\hat{P}_E^{i-1,j}(1-r_{i-1}) + \hat{P}_L^{i-1,j}(1-q_{i-1}-r_{i-1})\right) \\ & + \hat{\tau}_{i,j-1,l-1,e,g} \cdot \hat{P}_L^{i-1,j} q_{i-1} \\ & + \hat{\tau}_{i,j-1,l+1,e,g-2} \cdot \left(\hat{P}_G^{i-1,j} + (1-\hat{P}_G^{i-1,j})r_{i-1}\right).\end{aligned} \tag{B.7}$$

Third, if $2^{i-2} < j \leq 2^{i-1}$, there are $2^{i-1} - j + 1$ nodes in depth level $i$ cycle $j$. The probability that an $E$-type node is selected in depth level $i$ cycle $j$ is the probability that all $L$-type nodes in depth level $i$ at cycle $2^{i-2}$ (the last cycle with new nodes generated

for depth level $i$) has been explored before the $j^{th}$ cycle and not all remaining nodes are $G$-type nodes. This probability can then be expressed as

$$\hat{P}(N_L^{i,2^{i-2}} \leq j - 2^{i-2}, N_G^{i,2^{i-2}} \leq 2^{i-1} - j) = \sum_{m=0}^{2^{i-1}-j} \sum_{n=0}^{j-2^{i-2}} \hat{\tau}_{i,2^{i-2},n,2^{i-1}-j+1-m-n,m}. \quad \text{(B.8)}$$

With all the values of $i$ and $j$ discussed, the values of $\hat{\tau}_{i,j,l,e,g}$ can be obtained recursively.

# Appendix C

# Estimated Probability of E-type Nodes Exploration in Depth Level d

We choose to show the comparison for $q = 0.9, r = 0.3$ and $q = 0.5, r = 0.7$ as they are representative of the results for other pairs of $q$ and $r$ we tested in Section 4.4.1.

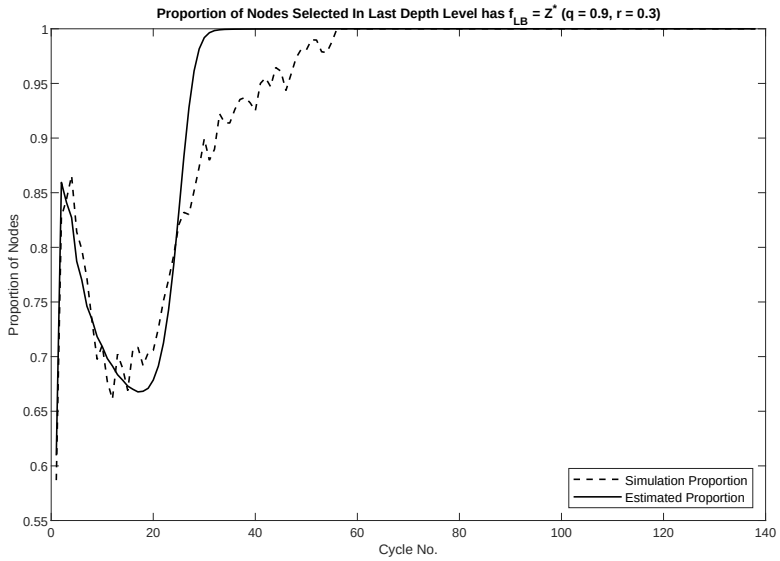



Figure C.1: Comparison of the estimated probability of $E$-type node exploration in depth level $d$ with simulation ($q = 0.9$, $r = 0.3$)

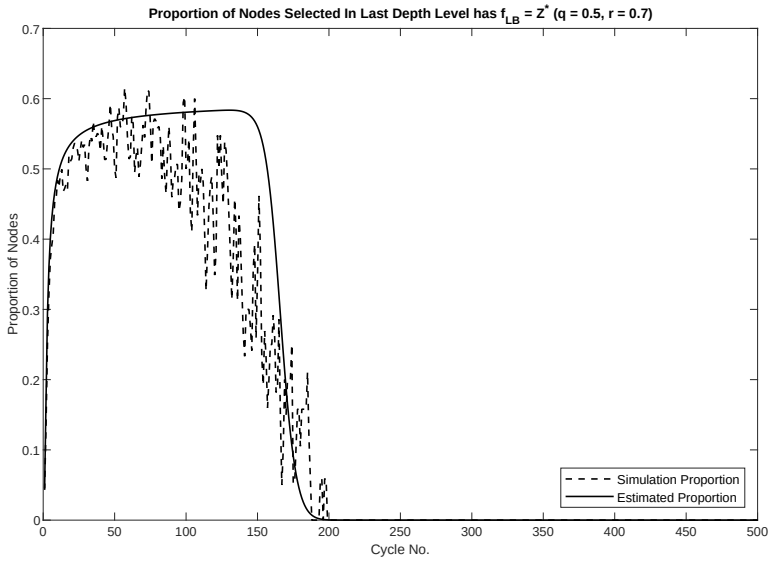



Figure C.2: Comparison of the estimated probability of $E$-type node exploration in depth level $d$ with simulation ($q = 0.5$, $r = 0.7$)

Figures C.1 and C.2 indicate that the estimated probability is close to the proportions obtained in simulation results. Note that the estimation overestimates the probability of $E$-type node exploration between cycles 20 and 40 in Figure C.1 and between cycles 100 and 200 in Figure C.2, which can explain the estimated expectation being smaller shown in Figures 4.1 and 4.3.