Optimization of Large-Scale Single Machine and Parallel Machine Scheduling

Large-Scale Single Machine and Parallel Machine Scheduling
in the Steel Industry
with Sequence-Dependent Changeover Costs

by

Che Lee, B.Ch.E

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Applied Science

McMaster University

MASTER OF Applied Science (2022) McMaster University
(Chemical Engineering) Hamilton, Ontario, Canada

TITLE: Large-Scale Single Machine and Parallel Machine Scheduling in the Steel Inudstry with Sequence-Dependent Changeover Costs

AUTHOR: Che Lee, B.Ch.E
(University of Minnesota Twin Cities, United States)

SUPERVISOR: Dr. Christopher L.E. Swartz

NUMBER OF PAGES: xii, 116

ABSTRACT

Hundreds of steel products need to be scheduled on a single or parallel machine in a steel plant every week. A good feasible schedule may save the company millions of dollars compared to a bad one. Single and parallel machine scheduling are also encountered often in many other industries, making it a crucial research topic for both the process system engineering and operations research communities.

Single or parallel machine scheduling can be a challenging combinatorial optimization problem when a large number of jobs are to be scheduled. Each job has unique job characteristics, resulting in different setup times/costs depending on the processing sequence. They also have specific release dates to follow and due dates to meet.

This work presents both an exact method using mixed-integer quadratic programming, and an approximate method with metaheuristics to solve real-world large-scale single/parallel machine scheduling problems faced in a steel plant. More than 1000 or 350 jobs are to be scheduled within a one-hour time limit in the single or parallel machine problem, respectively. The objective of the single machine scheduling is to minimize a combined total changeover, total earliness, and total tardiness cost, whereas the objective of the parallel machine scheduling is to minimize an objective function comprising the gaps between jobs before a critical time in a schedule, the total changeover cost, and the total tardiness cost. The exact method is developed to benchmark computation time for a small-scale single machine problem, but is not practical for solving the actual large-scale problem. A metaheuristic algorithm centered on variable neighborhood descent is developed to address the large-scale single machine scheduling with a sliding-window decomposition strategy. The algorithm is extended and modified to solve the large-scale parallel machine problem. Statistical tests, including Student's t-test and ANOVA, are conducted to determine efficient solution strategies and good parameters to be used in the metaheuristics.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to Dr. Christopher Swartz for his teaching and continued support over the last two years. I would not be where I am today, and this thesis would not have been possible without his encouragement and guidance.

Additionally, this research is in collaboration with ArcelorMittal Dofasco. I would like to extend my sincere thanks to Dr. Tyler Homer for his invaluable insights on the problems encountered and solution strategies developed in this work. My sincere thanks also goes to Vit Vaculik and Doug Bell who presented me the research problem and provided me with data to investigate it. This thesis would not have been possible without their valuable feedback.

I would also like to thank my colleagues in the McMaster Advanced Control Consortium for going through the research adventure together, and am grateful for the financial assitance recieved from the McMaster Department of Chemical Engineering.

Lastly, this thesis is dedicated to my parents, Pin Lee and Chi-Ling Chen, and my brother, Keifer Lee. I would not be where I am today and pursue the goals I have without their love and unconditional support.

Che Lee

August 2022

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Research Objectives

Steel manufacturing plants may need to sequence and process hundreds of steel products every week. Generating a feasible schedule that can minimize earliness, tardiness, and changeover costs of these products can save the company tremendous costs. A large portion of steel products after casting and hot rolling processes typically needs to be further refined by pickling, cold rolling, galvanizing, or even more processes. We consider a process of this type involving a single or parallel machine that processes hundreds of products a week, where single and parallel machine scheduling are also encountered often in many other industries. Scheduling these large-scale products on these machines becomes a challenging optimization problem because many types of sequence-dependent changeover costs are involved and each product has its own due date to meet.

The complex sequence-dependent cost relationship comes from the fact that each product has its unique product characteristics, and processing products with dissimilar product characteristics consecutively may require a long setup time and tear machine parts more often, resulting in a high changeover cost. Different product characteristics also weigh differently in a changeover cost. For example, products are categorized into a small number

of "product families" based on their special operation requirement on the machine, which can lead to a high changeover cost if two products come from different product families are processed consecutively.

Besides the above-mentioned changeover costs, the due date of a product is another important product characteristic that affects the total cost of a schedule. Neither processing a product too early nor too late is desired. If a product gets processed early, the inventory might be too full to accommodate it; if a product gets processed late, the customer service will be compromised. Thus, processing these products close to their due dates without being late is crucial, and the cost for earliness or tardiness of a product is high. Furthermore, a product cannot be processed before its arrival date. Doing so will lead to an infeasible schedule, so the optimization method needs to generate not only a low-cost but a feasible schedule within an hour for practical industrial implementation.

The large-scale single or parallel machine scheduling problem studied in this research can be approached by optimization methods like mathematical optimization or metaheuristics. Mathematical optimization is an exact method that models the optimization problem with specific mathematical formulation. This model can be solved by a modern solver such as Gurobi to optimality, which provides the exact information on lower and upper bounds of a solution during the solving process. However, an exact method might not solve our large-scale scheduling problems in a reasonable time. For the actual industrial problem size that involves hundreds of products to be scheduled, mathematical optimization is not a practical method to solve the actual problem.

On the other hand, metaheuristics are approximate methods that can find a sufficiently good, if not optimal, solution for a complex optimization problem faster than mathematical programming in general (Bianchi *et al.* [2009]). They are distinguished from heuristics by their generalizability, where a metaheuristic can be applied to solve different types of the optimization problems while a heuristic method may be specifically designed to solve one type of optimization problem (Talbi [2009]). Metaheuristics are also highly scalable, once the method is programmed, to increased problem sizes through techniques such as parallel

computing.

The goal of this research is to develop efficient metaheuristic algorithms to generate a feasible and high-quality schedule for the large-scale single/parallel machine scheduling problem within a one-hour limit for pratical implementation in industry. An exact method using mixed-integer quadratic programming is also developed for the single machine problem for comparison with metaheuristics, and statistical analysis such as t-test and ANOVA are conducted to analyze the peformance of metaheuristics.

1.2 Main Contributions

The main contributions of our research are as follows:

1. **Exact methods for benchmarking other solution approaches.** A rigorous mathematical model that can solve the smaller scale of our single machine scheduling problem to optimality is constructed. Although it is not practical for industrial applications due to its slowness for solving a large-scale problem, the exact method can benchmark the optimality against the performance of other methods such as a metaheuristic for solving a small-scale problem.

2. **Metaheuristics for efficient single machine scheduling.** Metaheuristic algorithms centered on variable neighborhood descent Hansen *et al.* [2010] are applied and developed to address our large-scale single machine scheduling problem. In particular, a sliding-window decomposition strategy is developed within the metaheuristics and combined with multiprocessing techniques for solving the large-scale problem efficiently.

3. **Metaheuristics for efficient parallel machine scheduling.** The algorithms we develop to solve the single machine problem are modified and extended to solve our large-scale parallel machine scheduling problem. Unique features designed for addressing parallel machines are developed in the metaheuristics to search for a good

solution systematically and efficiently.

4. **Statistical methods for analyzing algorithm performance.** The statistical methods centered on t-test and ANOVA are conducted to explore good parameters for our metaheuristic algorithm and compare the effectiveness of different solution strategies within the algorithm.

1.3 Thesis overview

Chapter 2 – Literature Review

An overview of research on single machine and parallel machine scheduling is provided. First, planning and scheduling in the steel industry are reviewed and the research scope is narrowed down to production scheduling. Classification on single and parallel machine scheduling within production scheduling then is provided. Last, research on this topic from both the process system engineering community focusing on exact methods and the operations research community focusing on metaheuristics is reviewed.

Chapter 3 – Single Machine Scheduling

This section describes our research to solve the real large-scale single machine scheduling problem faced in a steel plant. The details of the problem setups are provided. A mixed-integer quadratic programming model is developed to solve a small scale of the problem to optimality. A metaheuristic algorithm is developed along with decomposition and multiprocessing techniques to solve the large-scale problem efficiently. Statistical analysis such as ANOVA is provided to determine good parameters for the algorithm.

Chapter 4 – Parallel Machine Scheduling

This section extends the methodologies developed in the previous chapter to address another real large-scale parallel machine scheduling problem faced in the steel plant. Extra features for parallel machines are added to the metaheuristic algorithm to search for a good and feasible solution systematically and efficiently. Results of the algorithm performance and schedule quality are provided along with the statistical analysis such as t-test to determine a better solution strategy within the algorithm.

Chapter 5 – Conclusions and Recommendations

Conclusions on efficient performance of the developed metaheuristics from this research for solving the large-scale single and parallel machine scheduling problems are given. Multiple research directions for improving the solution strategies further along with the mindset of the continuous update of the solution strategies with the modern computing power is provided.

Chapter 2

Literature Review

The intent of this chapter is to provide a literature review relevant to the single machine and parallel machine scheduling problems we are addressing in this research. We start by reviewing planning and scheduling in steel manufacturing and classifying our single and parallel machine scheduling problems from a broader context of production scheduling. We then extend the scope from the steel industry to general industry, and review the important research and their methodologies for solving a single and/or parallel machine scheduling problem. In particular, we focus on the contributions to this research area from both the process systems engineering (PSE) community and operations research (OR) community.

2.1 Scheduling Problems in Steel Industry

In this section, we describe the importance and the challenges of scheduling in steel manufacturing, and classify the single and parallel machine problems we are focusing on from other types of production scheduling problems. For steel manufacturing, the reader is directed to the two papers, one for reviewing the planning and scheduling systems for integrated steel production by Tang *et al.* [2001], and the other for describing the integrated steelmaking processes and demonstrating a powerful solution for scheduling such complex integrated

production by Okano *et al.* [2004].

A modern steel plant contains a highly integrated production system connecting the upstream to downstream steel manufacturing processes (Tang *et al.* [2001]). The primary steelmaking, or the more upstream processes in steel production, involve the processes such as continuous casting and hot strip mill, while the finishing lines, or the more downstream processes in steel manufacturing, includes the processes such as cold mill, annealing, and galvanizing (Okano *et al.* [2004]). Planning and scheduling of such integrated steel production is very challenging because different processing constraints and objectives in different stages of the manufacturing process have to be met, and often these objectives may conflict with one another. However, the reduction of production cost, energy, and environmental pollution can be significant by proper planning and scheduling in the steel industry as in other industries such as electronics, but the amount of research devoted to planning and scheduling in the steel industry is relatively less compared to those industries (Tang *et al.* [2001]).

2.1.1 Solution Methodologies for Planning and Scheduling in Steel Industry

According to Tang *et al.* [2001], some of the main strategies for addressing steel production planning and scheduling are as follows:

1. Mathematical optimization: a mathematical programming formulation that models the constraints and objectives of the problem can be constructed. The mathematical model then is passed to a solver such as Cplex or Gurobi to find an optimal solution.

2. Intelligent Search: a metaheuristic algorithm such as a genetic algorithm, simulated annealing, or tabu search is applied to find a good and feasible solution in a reasonable amount of time for practical implementation in industry.

3. Human-machine coordination: an experienced human scheduler interacts with a computer system back and forth to iteratively improve the quality of a schedule.

4. Multi-agents methods: multiple models and algorithms, each of them is called an agent, work cooperatively to search for good and feasible solutions where the best solution out of them can be determined.

Each method above has its own advantages and disadvantages. For example, mathematical optimization such as mixed-integer linear programming (MILP) is able to find an optimal solution to a scheduling problem, but it may take too long to find the optimal solution to a large-scale problem for real applications in industry. Compared to mathematical optimization, intelligent search is able to search for a good and feasible solution to a large-scale optimization problem in a short amount of time and is suitable to be implemented in industry, but the solution cannot be guaranteed to be optimal and the quality of the solution is uncertain. For many industrial applications, finding a good and feasible solution instead of an optimal solution within a reasonable time is sufficient (Tang *et al.* [2001]).

In this research, we focus on the first two methods and especially the intelligent search with metaheuristics for generating practical solutions to two real-world large-scale scheduling problems faced by a steel company. In particular, we focus on the research on production scheduling instead of production planning where the difference between the two will be discussed in the following section.

2.1.2 Production Planning vs. Production Scheduling

Production planning and scheduling are interconnected in the context of supply chain management, as illustrated in Figure 2.1 reproduced from the paper for classification of chemical production scheduling by Maravelias [2012]. Specifically, the inputs to a production scheduling problem and the types of the production scheduling problem are determined by production planning decisions (Harjunkoski *et al.* [2014]). For example, depending on the planning system used in a manufacturing company, the release dates of the raw materials in a scheduling problem can be determined by procurement planning, and the due dates of the products in the scheduling problem can be determined by demand planning Harjunkoski *et al.* [2014]. In other words, we can think of production planning as a mas-

ter problem to production scheduling, where the upper level decisions made in production planning such as the types, quantities, and timing (e.g. release dates and due dates) of the products to be made are inputs to production scheduling for making lower-level decisions such as assignment and sequencing of the products, i.e., assigning these products to which machines with what processing orders and timing of executing these processing orders.

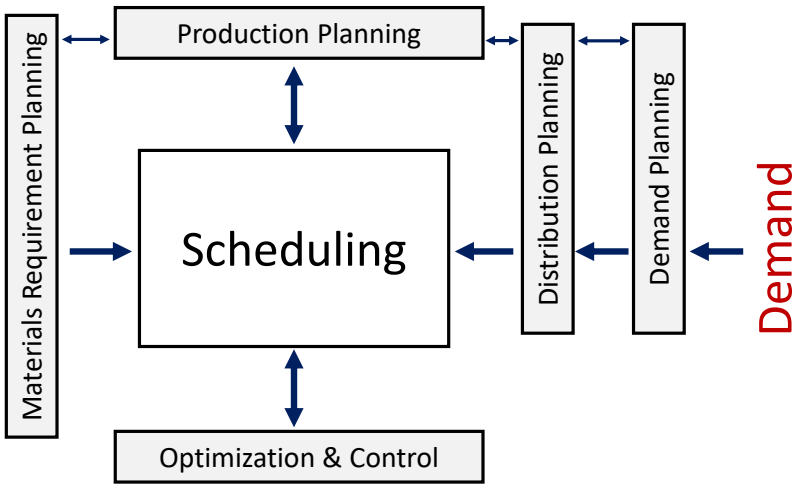

Figure 2.1: Planning and scheduling in supply chain management (Maravelias [2012]).

In this research, we narrow down the scope to production scheduling without worrying about how the upper-level decisions have been made in production planning. In particular, we focus on solving the two specific types of the production scheduling problems, single machine and parallel machine scheduling, which can be classified in both the PSE and OR communities and encountered in not just steel industry but also many other manufacturing industries. We will briefly describe the different types of the production scheduling problems and highlight our interests of the specific types of the problems in the next section.

2.1.3 Classification of Production Scheduling Problems

A scheduling problem can be described by a three-field notation $\alpha/\beta/\gamma$, where α stands for the machine environment such as a single machine or parallel machine, β describes the details of the processing characteristics such as whether the setup time and cost are involved, and γ defines the objective function of the scheduling problem (Graham *et al.* [1979]). There are more machine environments than single or parallel machine that could be encountered in production scheduling. We introduce some of the key machine environments in this section using the context of discrete manufacturing.

In discrete manufacturing, "job" is the term used to describe an object that cannot be split to be scheduled, "machine" stands for the resource for a job to be assigned to, and the number of "steps" refers to the number of stages or operations a job needs to go through to be made (Maravelias [2012]). Using this terminology, some of the main machine environments encountered in a scheduling problem are as follows:

1. Single machine: only one machine is considered in the scheduling problem. Every job considered in this problem has to be processed on the single machine.

2. Parallel machine: each job has to be processed on one of the M machines. So far, only one step is involved for each job in either the single or parallel machine problem.

3. Flow shop: each job has to go through the same series of M machines to be made, so there are M steps with the same specific order of the machines for each job to follow.

4. Job shop: there are M machines in total in this problem. Each job needs to follow a predetermined route of m machines ($m \leq M$) to be made, where a job may re-visit a machine more than once in its route.

5. Open shop: M machines in total are involved this problem. Each job needs to go through m machines ($m \leq M$) to be made, but the route of m machines for each job is not fixed and can be decided by a scheduler.

More complex machine environments could exist in a scheduling problem and the reader is directed to the paper for classifying production scheduling problems by Maravelias [2012] for more details. In this research, the machine environments we focus on are the single machine and parallel machine that are encountered often in steel industry and in many other industries. We will review how the PSE and OR communities address these types of problems with different processing characteristics (β) and objective functions (γ) in the following sections, and delve into the specific setup of the single machine and parallel machine problems we are solving in this research in Chapter 3 and 4.

2.2 Exact Methods for Production Scheduling

In the PSE community, exact methods, or mathematical optimization, are studied extensively to solve different types of production scheduling problems with different mathematical models. A review on the classification of the scheduling problems and optimization models for short-term batch scheduling can be found in Méndez *et al.* [2006], and a review on the industrial applications of production scheduling models and methods is given in Harjunkoski *et al.* [2014]. Based on these papers, we classify a general single or parallel machine problem following the classification used in the PSE community in this section. In the rest of this thesis, we use "SMSP" to refer to a single machine scheduling problem and "PMSP" to refer to a parallel machine scheduling problem. After giving examples of different mathematical models developed to solve different scheduling problems, we point out the unique challenges faced in our specific large-scale SMSP and PMSP being solved in this research, where we use LS-SMSP and LS-PMSP to refer to our specific large-scale single and parallel machine problems in the rest of this thesis.

2.2.1 Classification of SMSP and PMSP in the PSE Literature

Following the classification of scheduling problems defined by Méndez *et al.* [2006], a SMSP can be classified as a single-stage single-unit sequencing problem. It is a batch process where

batching (size and number of batches (jobs)) and assignment (which unit a batch (job) is assigned to) decisions have been predetermined prior to sequencing. The only decisions left to be made in a SMSP is to find the right sequence of the batches to minimize the objective. It is worth mentioning that a "batch" in the PSE literature basically refers to a "job" in the OR literature, and for the scheduling problems we consider in this research, a batch or a job cannot be mixed or split in a SMSP or PMSP.

A PMSP can be classified as a batch process with single stage that contains parallel units. Similar to a SMSP, the batching decisions have been made prior to a PMSP. However, assignment decisions need to be determined in addition to sequencing decisions in a PMSP, that is, each batch needs to be decided to be assigned to which unit with what order (timing of processing the batch on the unit).

Different mathematical models have been developed to solve scheduling problems with different $\alpha/\beta/\gamma$ in the PSE literature. Since a single machine or parallel machine is the basic unit forming a more complex production environment, a mathematical formulation developed for solving a more complex machine environment, for example, a multi-stage process, can be modified to solve a SMSP or PMSP given that the β and γ of the problems are similar.

However, it does not mean that solving our LS-SMSP and LS-PMSP with mathematical optimization is easy, since the processing characteristics (β) and objective function (γ) of our problems are complex to model due to the considerations of setup time/cost, earliness/tardiness, and arrival (release) date of a job. Also, the number of jobs to be scheduled in our problems is large ($\geq$ 1000 jobs for LS-SMSP and $\geq$ 350 jobs for LS-PMSP). We will describe some of the key mathematical formulations that could model a general SMSP/PMSP and discuss their applicability to our LS-SMSP/LS-PMSP in the following section. The specific problem setups of our LS-SMSP and LS-PMSP will be discussed in detail in Chapters 3 and 4.

2.2.2 Mathematical Models for SMSP and PMSP

Following the PSE literature, one could try modeling a SMSP using mixed-integer linear programming (MILP) with time-slot based formulation (Pinto and Grossmann [1995]) or general-precedence based formulation (Mendez *et al.* [2001]; Cóccola *et al.* [2014]). Due to the fact that more than 1000 jobs or 350 jobs are to be sequenced in LS-SMSP or LS-PMSP, one may apply MIP-based decomposition methods (Kopanos *et al.* [2010]) or combine MILP with heuristics (Roslöf *et al.* [2002]) in an attempt to solve such a large-scale problem. However, building these MILP models is not trivial, since one has to tailor a model to make sure it includes the sequence-dependent features between jobs, earliness or tardiness of each job, and the job-availability hard constraint. Moreover, even with the help of the decomposition or heuristics, MIP-based methods may still be too slow to solve LS-SMSP because more than 1000 jobs are to be sequenced, and the computation efforts of the MIP would typically increase with the number of jobs being modeled.

To deal with the challenging scheduling problems faced in steel industry, researchers have tailored their optimization methods to approach specific scheduling problems in steel manufacturing. A decomposition and aggregation strategy utilizing MILP and LP that first partitions the jobs into groups of similar job properties, sequences the jobs in each group respectively, and aggregates these groups with a fast improvement strategy in the end can be found in Harjunkoski and Grossmann [2001]. In their problem, about 100 jobs are considered to be scheduled across 4 serial stages, where the first stage consists of 2 parallel machines while the rest of the stages consists of a single machine.

Another novel algorithm using MILP with decomposition methods to address scheduling of the batch annealing process, which can be considered as a multipurpose batch plant with re-entrant flows in the parallel units, is proposed by Moon and Hrymak [1999]. A maximum of 24 jobs and 29 shared units are considered in their work, where each job has to go through a series of stages, and the objective is to minimize the makespan while meeting complex constraints. From the above examples, one may notice that different optimization strategies have been customized to approach specific scheduling problems in the steel industry, but

the structure of these problems is quite different from our LS-SMSP/LS-PMSP. Although no multiple stages and units are involved in LS-SMSP/LS-PMSP, it needs to sequence more than 1000 or 350 jobs with complex sequence-dependent relationships within a reasonable time. This imposes a different aspect of difficulty than a multi-stage multi-machine problem with a small number of jobs.

In this work, we applied and modified a unit-specific general precedence mathematical formulation proposed by Kopanos *et al.* [2009] to model our LS-SMSP and compare the exact method with the intelligent search. The details of the mathematical model, problem statements of LS-SMSP, and comparison between the exact method and intelligent search can be found in Chapter 3.

2.3 Metaheuristics for Production Scheduling

In this section, we start by giving the definitions of metaheuristics and comparing it with heuristics to distinguish the two terms. Then, we review papers for solving different types of SMSPs and PMSPs using metaheuristics in the OR community, highlight the difference between their problems and our LS-SMSP/LS-PMSP, and point out the need of our research.

2.3.1 Metaheuristics vs. Heuristics

The reader is directed to a book for classification, design, and implementation of a variety of metaheuristics by Talbi [2009] for this section. The intelligent search methods we focus on applying to solve our challenging LS-SMSP and LS-PMSP are metaheuristics, which are extensively studied in the OR community to address a variety of scheduling problems. Metaheuristics represent a family of approximate methods and are algorithms that can generate good feasible solutions to challenging optimization problems faced in science and engineering in a reasonable time (Talbi [2009]). Unlike exact methods, the solutions generated by metaheuristics are not guaranteed to be optimal, and unlike approximation algorithms, metaheuristics cannot prove how close their solutions are to the optimal one

(Talbi [2009]). However, obtaining an optimal solution to a challenging optimization problem faced in industry is often unrealistic, so metaheuristics that can generate acceptable solutions in a reasonable time are gaining significant interest (Talbi [2009]).

Compared to heuristics, metaheuristics are upper-level methodologies that can be used as a template to address different kinds of optimization problems and help to design underlying heuristics to solve a specific problem (Talbi [2009]). In other words, metaheuristics can be viewed as a more generalizable intelligent search method than heuristics where the latter is usually designed for solving a specific problem and hard to transfer to solve another problem directly.

2.3.2 Metaheuristics for Solving SMSPs and PMSPs

SMSPs and PMSPs with different variations in the problem setup (β and γ) have been addressed by a variety of metaheuristics in the OR literature, where a comprehensive survey on scheduling problems with setup times/costs can be found in Allahverdi [2015]. We are particularly interested in the processing characteristics (β) – setup times/costs involved in a SMSP/PMSP since these features are practical for industrial applications and encountered in our LS-SMSP/LS-PMSP. We review some key literature applying metaheuristics to address SMSPs/PMSPs with setup times or costs in this section.

For single machine scheduling, Kirlik and Oguz [2012] proposed a general variable neighborhood search algorithm to solve a SMSP with sequence-dependent setup times where the objective is to minimize the total weighted tardiness. The maximum number of jobs to be considered in their work was 85. Sioud *et al.* [2012] proposed a hybrid genetic algorithm to solve a SMSP with sequence-dependent setup times where the objective is to minimize the total tardiness. We abbreviate their SMSP as SMSP-MTT since their goal is to minimize the total tardiness for later reference. The number of jobs to be considered in the authors' work was also up to 85. Wang and Tang [2010] proposed a hybrid metaheuristic that combines scatter search and variable neighborhood search to solve a prize-collecting SMSP with sequence-dependent setup times. The problem is motivated from practical production

scheduling in the steel industry, where there are too many jobs to be processed within a shift on a machine so a scheduler has to select a subset of jobs to be processed and maximize the productivity of the machine. The maximum number of jobs considered in their work is 180.

For parallel machine scheduling, Avalos *et al.* [2014] proposed an efficient multi-start algorithm to solve the unrelated (processing time of a job can vary on different machines) PMSP with sequence and machine-dependent setup times. They applied a constructive heuristic to form an initial solution and utilized variable neighborhood descent algorithms that consist of inter and intra-machine movements to improve an initial solution. In addition, they proposed an acceptance criterion to guide the search and explore a neighborhood efficiently. The maximum number of jobs and machines to be considered in their work are 250 and 30, respectively. Lin *et al.* [2013] applied an ant colony optimization (ACO) algorithm to solve the unrelated PMSP where the objective is to minimize total weighted tardiness. They designed a machine reselection step and incorporated a local search technique in their ACO algorithm and found that it improved the algorithm performance significantly. The maximum number of jobs and machines to be considered in the authors' work are 100 and 10, respectively.

Many other metaheuristics have been applied to solve SMSPs/PMSPs with different variations in the problem setup, but to the best of our knowledge, no literature has addressed SMSPs with the problem size as large as ours ($\geq$ 1000 jobs for LS-SMSP), and with the sequence-dependent setup costs and times, tardiness, earliness, and availability hard constraint. Similarly, the unique problem setup and objective function encountered in our LS-PMSP have not been addressed in the literature. Since the SMSP-MTT is NP-hard, our more complex LS-SMSP and LS-PMSP are also NP-hard. In addition, a one-hour time limit is imposed to solve LS-SMSP by our industrial partner, so the algorithm developed has to return the best solution it can find within this time frame for practical implementation in the steel plant. We thus will focus on applying metaheuristics instead of exact methods to approach LS-SMSP, where the details of the problem statements, methodologies, and computational results are given in Chapter 3. We also extended the methods we developed for

LS-SMSP to address LS-PMSP, where the unique problem setup, metaheuristic algorithms, and computational performance can be found in Chapter 4.

Chapter 3

Single Machine Scheduling

In a steel manufacturing plant, oftentimes hundreds of jobs are required to be processed on a single machine within a week. A good processing sequence of these jobs can save the company millions of dollars compared to a bad one. However, sequencing these jobs is a challenging combinatorial optimization problem that we are addressing in this work because (1) the problem size is large (>1000 jobs), (2) many types of sequence-dependent setup costs/times between jobs are involved, (3) earliness and tardiness of jobs need to be considered, and (4) an availability hard constraint that forces jobs not to be processed before their arrival dates exists. We will use LS-SMSP as the abbreviation to refer to our large-scale single machine scheduling problem in this thesis.

In this chapter, we propose an algorithm centered around variable neighborhood descent (VND) with a sliding-window decomposition strategy, which we abbreviate as swVND to solve LS-SMSP within a one-hour limit. VND is one of the basic variations of variable neighborhood search (VNS) introduced by Mladenović and Hansen [1997] where the idea is to systematically change the neighborhoods during the local search to find a good local optimum. Different from VNS that involves randomness by perturbing the current solution before a local search, VND is deterministic and conducts a local search without any random perturbations. We chose VND over VNS for solving LS-SMSP because the problem size is so large (>1000 jobs) that even finding a local optimum within the one-hour time limit

is computationally expensive. Given such limited time, we prefer putting all the time to search for a local optimum with respect to multiple neighborhoods than splitting time to perturb the solution early in the hope of landing at a better local optimum later.

The rest of the chapter is organized as follows. Section 3.1 gives the detailed problem statement on LS-SMSP. Section 3.2 explains the algorithm components supporting swVND, and Section 3.3 demonstrates the implementation of swVND. Computational results and statistical analysis of the proposed algorithm on the real data provided by our industrial partner are given in Section 3.4. Finally, Section 3.5 concludes this chapter.

3.1 Problem Statement

In this section, we start by describing the details of the challenges faced by LS-SMSP in Section 3.1.1, and present a mathematical formulation to model our problem in Section 3.1.2. We then describe the solution representation of LS-SMSP in metaheuristics and how to evaluate and determine the feasibility of a solution in Sections 3.1.3, 3.1.4, and 3.1.5.

3.1.1 Problem Details

Setup Costs and Times

As mentioned in the beginning of this chapter, one of the key characteristic of LS-SMSP is that many types of sequence-dependent setup costs/times between jobs are involved. The complex sequence-dependent cost relationship comes from the fact that each job has its unique job attributes, and processing jobs with dissimilar attributes consecutively may require a long setup time and tear machine parts more often, resulting in a high setup cost. Some job attributes are continuous, such as the dimensions of a product being processed; others are discrete, such as either activating a specific tool (marked as 1) or not (marked as 0) when processing this job. Different attributes weigh differently in a changeover cost.

In addition, jobs are categorized into a small number of "job families" based on their special operation treatment on the single machine. Two jobs belong to different job families processed consecutively will lead to a huge setup cost and time since a long cleanup or maintenance will be required for such a transition. The penalty for the job family setup cost is the highest among all types of setup costs.

In this paper, we can assume that the setup costs and times between jobs have been pre-computed before solving the optimization problem. They are saved in two two-dimensional matricies, one for setup costs and the other for setup times. In each 2D matrix, the rows and columns are ordered by job identifier number (ID), i.e., $setup\_cost\_matrix[0, 1]$ represents the setup cost between job 0 and job 1 if job 1 is processed right after job 0.

Time-Related Costs

Besides the above-mentioned setup costs, due date is another job attribute that can lead to time-related costs like earliness or tardiness of a job based on when the job gets processed. If a job gets processed earlier than its due date (earliness), the inventory might be too full to accommodate it; if it gets processed later than its due date (lateness), the customer service will be compromised. Thus, processing these jobs close to their due dates without being late is crucial, and the cost for earliness or tardiness of a job is high.

Arrival Dates

Another important job attribute is the arrival date of a job which imposes the availability hard constraint to the scheduling problem: a job cannot be processed before its arrival date. The impact of the availability constraint to the problem will be discussed in detail in Section 3.1.5.

For simplicity and confidentiality reasons, only the following job attributes will be discussed in this paper: job family, job size, processing time, tool activation, due date, and arrival date. All of these job attributes are deterministic and their values are specified in the input

data before solving the scheduling problem.

3.1.2 Mathematical Formulation

In this section, we model our single-machine scheduling problem following the unit-specific general precedence (USGP) mathematical formulation proposed by Kopanos *et al.* [2009] with modifications to tailor their general formulation that is applicable to parallel machines to fit our unique single-machine problem. The resulting mathematical formulation is in a form of mixed-integer quadratic programming (MIQP). We will start by introducing the nomenclature used to model our problem, describe the assumptions made in this model, and explain each constraint in the model.

Nomenclature:

Subscripts

i, i': job ID

f: job family ID

Sets

I: set of jobs

I_f: set of jobs that belong to job family f ($I_f \subseteq I$)

Parameters

n: number of jobs to be scheduled

$sc_{i,i'}$: setup cost between job i and job i'

$st_{i,i'}$: setup time between job i and job i'

pt_i: processing time of job i

d_i: due date of job i

ad_i: arrival date of job i

M: a very large number

Binary variables

$X_{i,i'}$: general precedence relationship, i.e., whether job i' is processed after job i ($X_{i,i'} = 1$) or before job i ($X_{i,i'} = 0$)

$Seq_{i,i'}$: immediate precedence relationship, i.e., whether job i' is processed right after job i ($Seq_{i,i'} = 1$) or not ($Seq_{i,i'} = 0$)

δ_{T_i}: whether job i is processed later than its due date ($\delta_{T_i} = 1$) or not ($\delta_{T_i} = 0$)

Non-negative continuous variables

C_i: completion time of job i

E_i: earliness of job i

T_i: tardiness of job i

F_i: whether job i is the first job to be processed in the sequence ($F_i = 1$) or not ($F_i = 0$)

$Pos_{i,i'}$: position difference between job i and job i'. Auxiliary variables used to determine $Seq_{i,i'}$

Model Assumptions

1. A no-wait policy that forces a job to be processed right after the completion time of its immediate predecessor plus the setup time between these two consecutive jobs is enforced in our scheduling problem. In other words, a solution generated by either MIQP or metaheuristics cannot contain any gaps in the schedule except for the setup times between any two consecutive jobs.

2. Continuing the concept of the first assumption, the first job in the sequence must be processed at time 0 of the scheduling time horizon to avoid any gaps other than the setup times in the schedule.

3. An upper-level algorithm will preprocess the data such that the input data in the scheduling problem does not contain jobs that arrive so late that a solver cannot find a feasible solution following the no-wait policy discussed above. Those very late jobs relative to the current scheduling time horizon will be considered in a separate solution

batch and solved with an appropriate new starting time in their new scheduling time horizon.

4. All the parameters in the model are deterministic and not stochastic.
5. A job can be considered as a batch that cannot be split.
6. A job will not be interrupted once it starts being processed.

Model Constraints

Arrival-Time Constraints. A job can only be processed after its arrival time, so its completion time must be greater than or equal to its arrival time plus its processing time.

$$C_i \geq ad_i + pt_i \quad \forall i \in I \tag{3.1}$$

Earliness and Tardiness Constraints. A job is either processed earlier or later than its due date if its completion time is not equal to its due date. Since E_i and T_i are non-negative and the objective of this problem involves minimizing the earliness and tardiness costs, E_i or T_i of a job will be kept at 0 if it is processed later or earlier than its due date, respectively.

$$E_i \geq d_i - C_i \quad \forall i \in I \tag{3.2}$$

$$T_i \geq C_i - d_i \quad \forall i \in I \tag{3.3}$$

In addition, a base penalty is added in the objective function whenever a job is processed late, so we can use δ_{T_i} to indicate if job i is processed late and multiply the base penalty with δ_{T_i} in the objective function to activate the base penalty when $\delta_{T_i} = 1$. Constraints 3.4 and 3.5 are used to assign the right values to δ_{T_i}. If job i is processed late, $\delta_{T_i} = 1$ so its tardiness is positive and constraint 3.5 becomes redundant. Similarly, if job i is processed early, $\delta_{T_i} = 0$ so constraint 3.4 becomes redundant and constraint 3.5 forces its tardiness to be zero.

$$T_i \geq -M(1 - \delta_{T_i}) \quad \forall i \in I \tag{3.4}$$

$$T_i \leq M\delta_{T_i} \quad \forall i \in I \tag{3.5}$$

Sequencing Constraints. The sequencing constraints between jobs i and i' can be expressed in constraint 3.6. The big-M constraint is activated when the global sequencing binary variable $X_{i,i'}$ equals to 1, meaning job i' is processed after but not necessarily immediately after job i. The constraint is redundant if job i' is processed before job i ($X_{i,i'} = 0$). Furthermore, if job i' is processed immediately after job i, i.e., the immediate sequencing binary variable $Seq_{i,i'}$ equals to 1, then the sequence-dependent setup time between jobs i and i' ($st_{i,i'}$) is considered in the sequencing constraint. It is otherwise excluded because $Seq_{i,i'} = 0$.

$$C_{i'} \geq C_i + st_{i,i'}Seq_{i,i'} + pt_{i'} - M(1 - X_{i,i'}) \quad \forall i, i' \in I, \quad i \neq i' \tag{3.6}$$

Zero-wait Constraints. Constraint 3.7 ensures that job i' starts to be processed right after job i is done plus the setup time between them without any additional delay, if the immediate sequencing binary variable $Seq_{i,i'}$ equals to 1. It is redundant when $Seq_{i,i'} = 0$.

$$C_{i'} \leq C_i + st_{i,i'}Seq_{i,i'} + pt_{i'} + M(1 - Seq_{i,i'}) \quad \forall i, i' \in I, \quad i \neq i' \tag{3.7}$$

For processing the first job in the sequence right at the beginning of the scheduling time horizon without any delay (assumption 2 in Section 3.1.2), constraints 3.8 and 3.9 are implemented. Constraint 3.8 utilizes the fact that $\sum_{i \neq i'} Seq_{i,i'} = 0$ only if job i' is the first job in the sequence, since no other jobs are sequenced before job i' in this case. Thus, $F_{i'}$ equals to 1 only if job i' is the first job in the sequence, and the big-M constraint 3.9 that forces the first job to be processed right at the beginning of the scheduling time horizon is activated only if F_i equals to 1.

$$F_{i'} = 1 - \sum_{i \neq i'} Seq_{i,i'} \quad \forall i' \in I \tag{3.8}$$

$$C_{i'} \leq pt_{i'} + M(1 - F_{i'}) \quad \forall i \in I \tag{3.9}$$

Global Precedence Constraints. Constraint 3.10 ensures that if job i' is processed after job i ($X_{i,i'} = 1$), then job i' is not processed before job i ($X_{i',i} = 0$) and vice versa.

$$X_{i,i'} + X_{i',i} = 1 \quad \forall i, i' \in I, \quad i < i' \tag{3.10}$$

Immediate Precedence Constraints. Kopanos *et al.* [2009] observe that $\sum_{i'' \neq [i,i']}(X_{i,i''} - X_{i',i''}) = 0$ only if jobs i and i' are sequenced consecutively. Following this fact, constraint 3.11, constraint 3.12, and auxiliary variables $Pos_{i,i'}$ are developed to determine $Seq_{i,i'}$. $Pos_{i,i'}$ equals to 0 only if $X_{i,i'} = 1$ and $\sum_{i'' \neq [i,i']}(X_{i,i''} - X_{i',i''}) = 0$, i.e., job i' is not only processed after job i but immediately processed after job i. Constraint 3.12 then forces $Seq_{i,i'}$ equal to 1 when $Pos_{i,i'} = 0$, meaning that job i' is immediately processed after job i.

$$Pos_{i,i'} = \sum_{i'' \neq [i,i']} (X_{i,i''} - X_{i',i''}) + M(1 - X_{i,i'}) \quad \forall i, i' \in I, \quad i \neq i' \tag{3.11}$$

$$Pos_{i,i'} + Seq_{i,i'} \geq 1 \quad \forall i, i' \in I, \quad i \neq i' \tag{3.12}$$

In addition, constraints 3.13 and 3.14 ensure $Seq_{i,i'}$ equals to 0 when job i' is not immediately processed after job i. Constraint 3.13 enforces that at most one of the jobs other than job i can be immediately processed after job i. It does not use the equal sign since no job is processed after the last job in the sequence, i.e., $\sum_{i' \neq i} Seq_{i,i'} = 0$ for the last job in the schedule. Lastly, constraint 3.14 ensures that given n jobs to be sequenced on a single machine, in total there are $n-1$ connections between the consecutive jobs in this sequence.

$$\sum_{i' \neq i} Seq_{i,i'} \leq 1 \quad \forall i \in I \tag{3.13}$$

$$\sum_{i \neq i'} \sum_{i' \neq i} Seq_{i,i'} = n - 1 \tag{3.14}$$

Objective Function

The objective function in our problem involves minimizing a combination of sequence-dependent setup costs, earliness costs, and tardiness costs with predetermined coefficients for each term in the equation. The quadratic terms in earliness and tardiness costs intend to penalize the very early or very late jobs more. Although the penalty coefficients are confidential and the exact numbers can not be reported, they are just constants tailored for a specific application and do not affect the general solution strategies – the exact method

and the metaheuristics that we are presenting in this thesis.

$$min \sum_{i \neq i'} \sum_{i' \neq i} Seq_{i,i'} sc_{i,i'} + \sum_{i \in I_1} (\alpha_1 E_i + \alpha_2 E_i^2) + \sum_{i \in I_{2,3,4}} (\alpha_3 E_i + \alpha_4 E_i^2) + \sum_{i \in I} (\beta_1 \delta_{T_i} + \beta_2 T_i + \beta_3 T_i^2) \tag{3.15}$$

3.1.3 Solution Representation in Metaheuristics

In our metaheuristics algorithm, a solution (x) to the LS-SMSP problem is represented as a permutation vector of n jobs ($x = \{x_1, ..., x_k, ..., x_n\}$) where n represents the number of jobs to be sequenced and x_k is the job to be processed in the kth order. This representation can be easily coded in many programming languages and manipulated by metaheuristics. The convention for specifying the first position in the vector will stick to 1 instead of 0 in this chapter.

3.1.4 Evaluating a Solution in Metaheuristics

Given a candidate solution x represented as a permutation vector of n jobs, one can calculate the objective function value of x, $f(x)$, explicitly. This evaluation of a sequence x will be performed many times in metaheuristics, so we give it a special attention in this section. We will outline one of the possible ways to program such a task below.

An artificial input dataset can be illustrated in Table 3.1. In each row in Table 3.1, the value in the first column represents the job ID number while the rest of columns in the row represents the job attribute values for that job. The parameters used in the MIQP model such as the $n \times n$ setup cost matrix and setup time matrix are constructed based on this data and can be assume to be pre-computed before solving the scheduling problem.

These parameters are also utilized to evaluate a solution x in metaheuristics. For the total setup cost of a sequence ($\sum_{i \neq i'} \sum_{i' \neq i} Seq_{i,i'} sc_{i,i'}$ in Equation 3.15), one can sequence through the vector x and sum over the setup costs between consecutive jobs by utilizing the setup cost matrix in programming. For the total tardiness and earliness of a sequence

Table 3.1: Example of an artificial input data.

Job ID	Job Family	Processing Time	Due Date	Arrival Date	Size	Tool Activation
1	3	100	1000	0	20	1
2	3	150	1200	150	15	1
3	1	90	1100	200	22	0
4	2	200	2000	0	10	0
5	1	250	500	0	25	1
...	...	...	...	...	...	...
1169	2	100	700	0	20	0

($\sum_{i\in I_1}(\alpha_1 E_i + \alpha_2 E_i^2) + \sum_{i\in I_{2,3,4}}(\alpha_3 E_i + \alpha_4 E_i^2) + \sum_{i\in I}(\beta_1 \delta_{T_i} + \beta_2 T_i + \beta_3 T_i^2)$ in Equation 3.15), first we can calculate the start time and completion time of each job in x. The first job in the sequence always start at time 0, and its completion time is simply its processing time. We then can sequence through the jobs in the vector x one by one and compute a job's start time by adding the setup time between the job and its immediate predecessor to the completion time of the latter. The completion time of a job is simply the start time of the job plus the processing time of the job.

Once we compute all the start and completion times of all the jobs in x, the earliness or tardiness of a job is simply its due date minus its completion time or its completion time minus its due date, depending on which value is larger. By summing the total setup costs and total earliness and tardiness of a sequence, we obtain the objective function value.

3.1.5 Feasibility of a solution

Although a solution x is represented as a permutation vector, not every permutation of n jobs is feasible. Following the calculation of start times in the previous section, if any start time of a job in the sequence is smaller than its arrival date, i.e., $s_i < ad_i$, then such sequence is infeasible no matter how good its objective function value is.

As assumption 3 in section 3.1.2 describes, the input data will not include any job that has

a very late arrival date that makes no feasible permutation exist. Those very late-arrival jobs will be considered in the future week's scheduling. How the upper-level algorithm sorts out very late jobs is not the focus of this work, but in general, the input data does not include any job whose arrival date is later than 50 percent of the total processing times of the current batch of jobs, so we can make sure that most of the permutations are feasible. The algorithm we present in the next section will also show how it is likely to return a feasible solution given that no arrival date of a job is unreasonably high in a dataset.

3.2 Algorithm Components in the Proposed Solution

Metaheuristics usually start with an initial solution to begin the improvement process to approach a local or global optimum. A good initial solution may help metaheuristics find a good solution in a shorter time. Section 3.2.1 shows an efficient way to obtain a good initial solution to LS-SMSP. A good initial solution thus can be further improved by the VND metaheuristic explained in section 3.2.2.

3.2.1 Greedy Constructive Heuristic to Obtain an Initial Solution

In our work, we apply a fast heuristic to construct a good and feasible initial solution for metaheuristics to further improve on. The constructive heuristic we apply follows the main concepts in the constructive step of the solution strategy introduced by Kopanos *et al.* [2010] with some modifications. In their paper, the authors use mixed-integer programming to carry out this task, while we use computer programming to perform the similar task instead for the sake of the solution speed. In addition, we tailor this method to deal with the feasibility issue caused by the arrival times of the jobs in our problem, so the revised algorithm generates both a good and feasible initial solution. We will call this modified constructive heuristic "greedy constructive heuristic with feasibility guaranteed (GCF)" throughout this work.

Two vectors, one named "*wait_list*" and the other named "*current_sequence*" are used

in GCF to construct an initial solution. Initially, *wait_list* contains randomly permutated n jobs and *current_sequence* is empty. GCF then removes jobs from the "*wait_list*" one at a time, and inserts them to a good position in "*current_sequence*" while keeping the relative positions of remaining jobs in "*current_sequence*" the same. The good insertion position for the job is determined by iteratively inserting it to a feasible position in the "*current_sequence*", computing the corresponding sequence value and updating the best sequence and best value obtained so far, and proceeding with the position that leads to the best sequence value out of all possible insertions.

GCF is greedy in a sense that when a job is removed from the *wait_list* and being inserted into the *current_sequence*, it keeps the relative positions of the previous jobs in the *current_sequence* the same without going through other possible permutations in the *current_sequence* which may lead to a better sequence value at this stage. This saves a lot of computation time and constructs a good initial solution efficiently. It leaves the rest of the improvements by searching for more good possible permutations to metaheuristics, which will be covered in the next section.

In addition, GCF guarantees that the output sequence will be feasible. Every time a job is removed from the *wait_list* and inserted to a position in the *current_sequence*, the algorithm first checks the feasibility of the resulting sequence following the calculation of the start and completion times of each job and the comparison of the start time with arrival date introduced in Section 3.1.4. Therefore, the job can only be inserted to a feasible position in *current_sequence*, and if none of the positions in *current_sequence* is feasible for the job at this point, it is simply added back to the end of the *wait_list*. This way, when the previously infeasible job is removed from the *wait_list* again, the *current_sequence* that has been constructed is much longer, and there will likely be feasible positions for the job to be inserted into.

GCF terminates when the *wait_list* is empty, meaning that all of the jobs have been removed from the *wait_list* and the *current_sequence* contains n jobs at this point. Since GCF is fast to run and a different initial sequence passed to GCF may lead to a different quality

of the output, we can simply pass the output of GCF to GCF to run it again, and repeat this process until the new output is no longer improved by GCF. We call this second phase of the algorithm iterative GCF (iGCF), since we simply use a while loop to wrap GCF to further improve the initial solutions. The pseudo code for Algorithm GCF_BestInsertion that determines the best insertion position for a job to be inserted into the *current_sequence* is shown in Algorithm 1, and for Algorithm GCF which iteratively removes jobs from the *wait_list* and applies Algorithm GCF_BestInsertion to insert them to the *current_sequence* until the *wait_list* is empty is shown in Algorithm 2. Finally, the pseudo code for Algorithm iGCF that iteratively applies Algorithm GCF until it cannot improve a solution further is shown in Algorithm 3.

Algorithm 1: GCF_BestInsertion

```
1  Function GCF_BestInsertion(job, current_seq):
2      best_val ← M;                                  // M is a very large value
3      best_seq ← [ ];
4      for k ← 1 to Length(current_seq)+1 do
5          insert job to kth position in current_seq;
6          if Feasible(current_seq) then
7              val ← Evaluate(current_seq);
8              if val < best_val then
9                  best_val ← val;
10                 best_seq ← Copy(current_seq);
11             end
12         end
13         remove job from current_seq;
14     end
15     if best_seq == [ ] then                  // if no feasible position exists
16         return current_seq
17     end
18     return best_seq
```

Algorithm 2: GCF

```
1  Function GCF(wait_list):
2      current_seq ← [ ];
3      while wait_list != [ ] do
4          job ← Remove(wait_list);   // Remove the 1st element out of wait_list
5          current_seq_old ← Copy(current_seq);
6          current_seq ← GCF_BestInsertion(job, current_seq);
7          if current_seq == current_seq_old then
8              insert job to the end of wait_list;
9          end
10     end
11     return current_seq
```

Algorithm 3: iGCF

```
1  Function iGCF(wait_list):
2      current_seq ← GCF(wait_list);
3      current_val ← Evaluate(current_seq);
4      new_seq ← GCF(current_seq);
5      new_val ← Evaluate(new_seq);
6      while new_val < current_val do
7          current_seq ← new_seq;
8          current_val ← new_val;
9          new_seq ← GCF(current_seq);
10         new_val ← Evaluate(new_seq);
11     end
12     return current_seq
```

An example to illustrate the iGCF algorithm is shown in Figure 3.1. Given a randomly permuted sequence ordered as job IDs [012] as input to iGCF, at the beginning of the algorithm *current_sequence* is empty and *wait_list* contains [012]. First, we remove job

0 from the *wait_list* and insert it to *current_sequence*. Assuming job 0 is available to be processed at time 0, *current_sequence* now has job 0 while *wait_list* contains [12]. Next, we remove the first job in the updated *wait_list* which is job 1 out of *wait_list*, and try to insert it before or after job 0 in *current_sequence*. Unfortunately, job 1 has an arrival date that is later than the processing time of job 0, so neither [01] nor [10] in *current_sequence* would be a feasible schedule. In this case, we simply place job 1 at the end of *wait_list*, and consider other jobs that might be available to be processed earlier in the schedule. Now, *current_sequence* contains job 0 while *wait_list* contains [21].

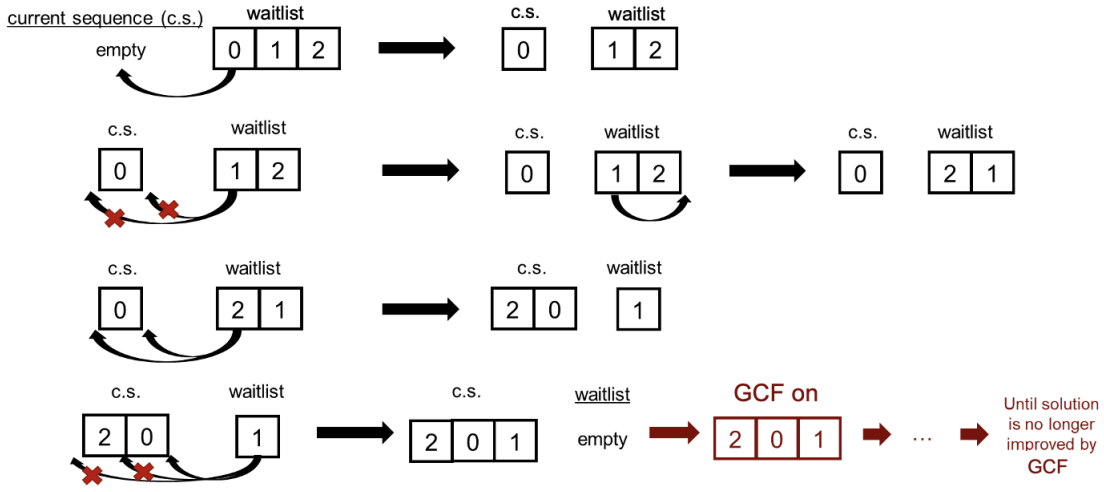

Figure 3.1: Illustration of iGCF algorithm.

Next, we remove the first job in the updated *wait_list* which is job 2 out of *wait_list* and try to insert it to all possible positions in *current_sequence*. Assuming job 2 is available to be processed at time 0, we now have two possible orders in *current_sequence*: either [20] or [02]. Assuming [20] gives us a lower objective value than [02], the algorithm proceeds with [20] as *current_sequence* and [1] as *wait_list*. We now remove the last job in *wait_list* and insert it to *current_sequence* while keeping the relative order of jobs in *current_sequence*, job 2 before job 0, the same. Although job 1 is not available to be processed in the first two positions in the *current_sequence* because of its arrival date, it now has the third option which is to be processed at the end of *current_sequence* where it is available by the time job 2 and 0 are both processed. At the end of GCF, a feasible and good initial solution is

constructed, but further improvement by GCF can still be possibly made. We then put the resulting *current_sequence* that contains [201] as input to GCF and see if the output of the second-time GCF is better than the previous output, [201]. We repeat this process until the solution is no longer improved by GCF, in which case we return the best solution we have obtained so far as the iGCF output. At the end of iGCF, a feasible and good initial solution for further improvement by other metaheuristics is obtained.

3.2.2 Variable Neighborhood Descent

After an initial solution is obtained by iGCF, we apply the metaheuristic variable neighborhood descent to further improve the solution. As stated in the Introduction section, VND is a deterministic version of VNS where the idea is to systematically change the neighborhoods during the local search to find a local optimum. The details of VND, VNS, and other variants can be found in Hansen *et al.* [2010] and Hansen *et al.* [2017], and the application of VNS to a single-machine scheduling problem that has a smaller problem size and simpler model compared to our problem is well-presented in Kirlik and Oguz [2012]. We will rephrase the core concepts relating to VND from the above-mentioned papers in this section for readers who are not familiar with VND.

The general form for a combinatorial optimization problem can be formulated as $min\{f(x)|x \in X \subseteq S\}$, where S represents a finite set of the solution space, X represents the set of all of the feasible solutions, x represents a feasible solution, and $f(x)$ represents the objective function value of x. A solution $x^* \in X$ is a global optimum to this problem if $f(x^*) \leq f(x),\ \forall x \in X$.

In VND, a finite set of neighborhood structures N_k $(k = 1, ..., k_{max})$ is pre-selected, where $N_k(x)$ denotes the set of solutions in the kth neighborhood of x. A solution $x' \in X$ is defined as a local minimum with respect to N_k if no solution x in $N_k(x')$ is better than x', i.e., $f(x) \geq f(x')\ ,\forall\ x \in N_k(x') \subseteq X$. VND moves toward such x' by applying a local search heuristic with systematic changes of the neighborhoods.

A local search heuristic generally starts with an initial solution x, searches for a descent

direction within a predefined neighborhood structure $N_k(x)$, and follows that descent direction to move toward a local minimum. It repeats this process until no descent direction can be found so a local minimum is reached. A common way to choose a descent direction in a local search is to follow the steepest direction in a neighborhood $N_k(x)$, also referred as best improvement and its algorithm is given in Algorithm 4.

Algorithm 4: Best improvement local search (Hansen *et al.* [2010]).

```
1 Function Best_Improvement(x, k):
2     repeat
3         x' ← x;
4         x ← argmin_{y∈N_k(x')} f(y);
5     until f(x) ≥ f(x');
6     return x
```

More than a best-improvement local search, VND involves a procedure to change the neighborhood structures systematically to guide the search process and find a better local minimum. A generic approach that sequentially changes the neighborhood structures is given in Algorithm 5. Function *NeighborhoodChange* compares the new solution x' with the current solution x in the kth neighborhood structure. If the solution is improved (line 2), the current solution is updated (line 3) and k is reset to 1 for starting with the first neighborhood structure for finding the next improvement on the updated solution (line 4). Otherwise, the next neighborhood structure being searched is updated to $(k + 1)$ in the hope of finding an improved solution in the new neighborhood structure that could not be found in the previous neighborhood structure (line 6).

Algorithm 6 shows the VND algorithm. Starting with an initial solution x, VND finds the best neighbor (steepest descent) in $N_k(x)$ (line 4) and resets the neighborhood to the initial neighborhood if the solution improves, or proceeds to the next neighborhood if no improvement is made (line 5). It differs from a typical local search heuristic in a way that it systematically explores many neighborhood structures ($k_{max} \geq 1$) where most local search heuristics involve only one neighborhood structure ($k_{max} = 1$). Since different neighborhood

Algorithm 5: Sequential changes in neighborhood structures (Hansen *et al.* [2010]).

```
1 Function Neighborhood_Change(x, x', k):
2     if f(x') < f(x) then
3         x ← x';
4         k ← 1;
5     else
6         k ← k + 1;
7     end
8     return x, k
```

structures are visited in VND, its local minimum with respect to these neighborhoods should be closer or equal to the global minimum than a local minimum with respect to only one neighborhood.

Algorithm 6: Variable neighborhood descent (Hansen *et al.* [2010]).

```
1 Function VND(x, k_max):
2     k ← 1;
3     repeat
4         x' ← argmin_{y∈N_k(x)} f(y);
5         x, k ← Neighborhood_Change(x, x', k);
6     until k = k_max;
7     return x
```

Neighborhood Structures

Kirlik and Oguz [2012] have showed that the choices of the neighborhood structures and the order of the neighborhood structures affect the quality of a converged solution by VND to their single-machine scheduling problem. In our work, the two simple but effective neighborhood structures we apply to solve the LS-SMSP problem are: (1) insertion, and

(2) swap.

The insertion neighborhood is composed of all the feasible solutions from all the possible insertion moves that can be made on the current solution, where an insertion move removes the job at position k_1 and inserts it to the position k_2 in the sequence as shown in Algorithm 7.

Algorithm 7: An insertion move (adapted from Kirlik and Oguz [2012]).

```
1 Function InsertionMove(x, k1, k2):
2     x^new ← Copy(x);
3     Remove job at position k1 in x^new;
4     Insert the removed job to position k2 in x^new;
5     return x^new
```

The best and feasible solution (best neighbor) in the current insertion neighborhood can be found by iterating through all the possible and feasible insertion moves that can be made on the current solution (Algorithm 8), which is essentially line 4 in function VND (Algorithm 6). Line 7 in Algorithm 8 shows that we are able to sort out the infeasible solutions found by insertion moves based on the feasibility rule discussed in Section 3.1.5. Note that the algorithms presented here are simply one of the many possible ways to find the best neighbor in an insertion neighborhood. They may be modified to complete the same task more computation or memory efficiently in a programming language.

Algorithm 8: The best and feasible insertion move (adapted from Kirlik and Oguz [2012]).

```
1  Function BestFeasible_InsertionMove(x):
2      x^best ← x;
3      best ← Evaluate(x);
4      for k1 ← 1 to n do
5          for k2 ← 1 to n do
6              x^new ← InsertionMove(x, k1, k2);
7              if Feasible(x^new) then
8                  temp ← Evaluate(x^new);
9                  if temp < best then
10                     x^best ← x^new;
11                     best ← temp;
12                 end
13             end
14         end
15     end
16     return x^best
```

Similarly, the swap neighborhood contains all the feasible solutions from all the possible swap moves applied to the current solution, where a swap move swaps the jobs at positions k_1 and k_2 as shown in Algorithm 9.

Algorithm 9: A swap move (adapted from Kirlik and Oguz [2012]).

```
1  Function SwapMove(x, k1, k2):
2      x^new ← Copy(x);
3      swap the jobs at positions k1 and k2 in x^new;
4      return x^new
```

Following the same strategy used in Function BestFeasible_InsertionMove, Algorithm 10 shows one way to find the best and feasible neighbor in the swap neighborhood.

Algorithm 10: Best and feasible swap move (adapted from Kirlik and Oguz [2012]).

```
1  Function BestFeasible_SwapMove(x):
2      x^best ← x;
3      best ← Evaluate(x);
4      for k1 ← 1 to n do
5          for k2 ← 1 to n do
6              x^new ← SwapMove(x, k1, k2);
7              if Feasible(x^new) then
8                  temp ← Evaluate(x^new);
9                  if temp < best then
10                     x^best ← x^new;
11                     best ← temp;
12                 end
13             end
14         end
15     end
16     return x^best
```

The time complexity for finding a best neighbor in either the insertion (Algorithm 8) or swap (Algorithm 10) neighborhood is $O(n^2)$, so finding the next best neighbor repeatedly when n is large is computationally expensive. Since the goal is to search for the best possible solution within one hour, methods that can speed up VND to find a better solution for the large-scale scheduling problem in a shorter time are valuable. Two methods we applied to speed up VND are "sliding-window" decomposition and multiprocessing, which will be introduced in the next section.

3.3 Methods to Speed Up VND

3.3.1 Sliding-Window VND

The main idea of sliding-window VND (swVND) is that unlike VND which spends much time to find a best neighbor from the complete sequence with n jobs, swVND finds a good neighbor from the smaller search space at the beginning and systematically increases the search space to find better moves when no better solution can be found in the previous smaller search space. It reduces the search space by imposing an imaginary window with a prespecified size on the search space, where we only conduct VND in this window, i.e., the jobs in this window can only be reinserted or swapped with other jobs in this window while no move is done on any jobs outside of this window. Once the solution can no longer be improved by applying VND to this window, we slide the window with a prespecified number of positions to the right (toward the end of the sequence), and re-apply VND in the slid window that contains a new range of positions in the sequence. The process repeats until the slid window hits the end of the sequence and the VND converges within this last window, which we call it one pass of sliding window. Then, we move the window to the beginning of the sequence and repeat the improvement process again, until no improvement can be made within a pass of swVND. At this point, the swVND algorithm terminates and we say that the solution has converged w.r.t. swVND.

To explain the above concepts more concretely, we will use Figure 3.2 and Algorithm 11 - 13

to go through an example of swVND application. The two parameters used in swVND are window size (win_size) that controls the size of the window and slide size ($slide\_size$) that controls the number of positions we slide the window toward the end of the sequence when no improvement can be made by VND in the previous window. In the Figure 3.2 example, the 300-job sequence is improved by swVND with the two parameters set as $win\_size = 100$ and $slide\_size = 80$. Algorithm 11 first computes all the start and end positions of the sliding window (line 2 in Algorithm 13). In this case, the four window start and end positions using the notation [start, end] are: [1, 100], [81, 180], [161, 260], [241, 300], so Algorithm 11 will return $win\_start\_list = [1, 81, 161, 241]$ and $win\_end\_list = [100, 180, 260, 300]$ given the inputs x, $win\_size = 100$, and $slide\_size = 80$. Then, the improvement process starts (Algorithm 12 and line 3 in Algorithm 13).

In the first pass of swVND, we start by applying VND to the window that contains the jobs from position 1 to position 100 (line 5 in Algorithm 12). This means that the jobs in the current window can be reinserted or swapped in the window depending on which neighborhoods are used in VND, but the jobs out of the current window do not get to move at all. In addition, the quality of a move made in the window is based on the evaluation of the full-length sequence corresponding to such move, so a best move found in the current window is the move that improves the value of the full-length sequence the most, not partial sequence. Following the VND concepts discussed in Section 3.2.2 and putting it into the context of swVND, an improvement made to the solution is based on the best move found in the current window, where the best move found in the current window [1, 100] is a relatively good move in the full-length sequence [1, 300]. Therefore, we can think of swVND as a improvement method to proceed with a good move instead of a best move w.r.t. the full-length search space. Also, the order of the jobs in the current window will keep being improved following each best move until no more improvement can be made within that window. At that point, the solution has converged w.r.t. VND applied to that window, or we simply say that VND has converged in that window.

Once the solution is no longer improved by VND in window [1, 100], we slide the window toward right by 80 positions and apply VND to the new window [81, 180]. We can see that

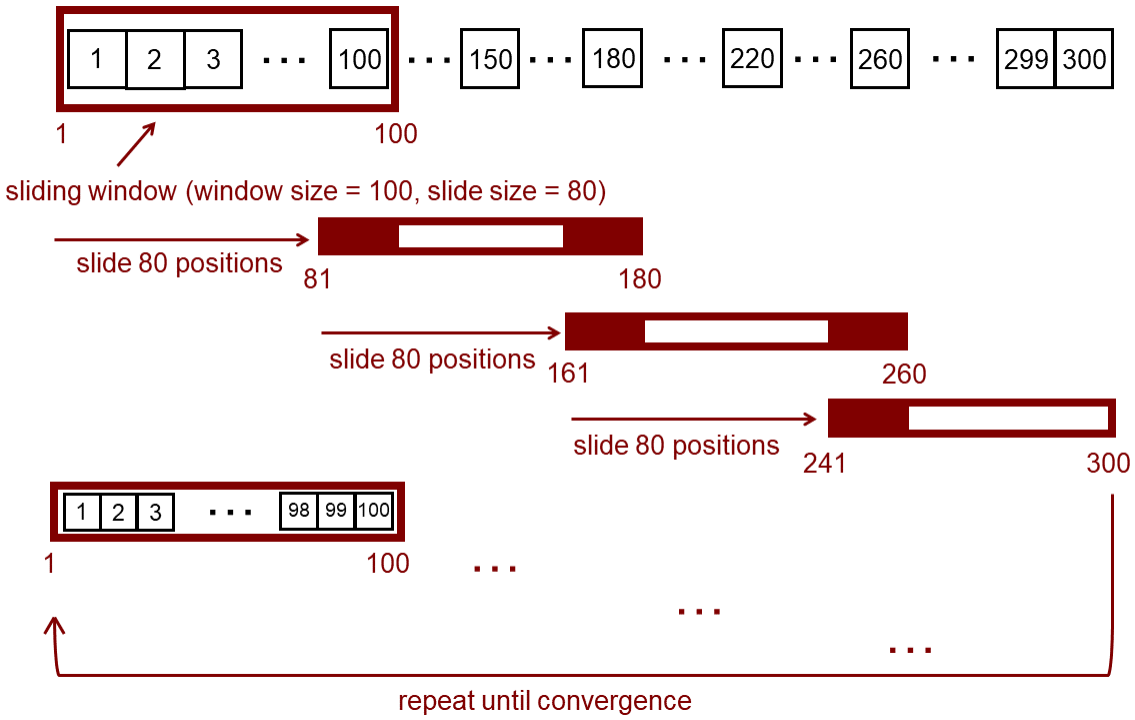

Figure 3.2: Illustration of the sliding-window decomposition method.

position 81 to 100 (the color-filled area in the rectangle in Figure 3.2) are reconsidered in the new window, where the rationale is that jobs at the end of the previous window (position 81 to 100) may be better off being moved to other positions in the new window (position 101 to 180), so the slide size is set to 80 instead of 100 which would be the same length of the window size. Next, we repeatedly slide the window toward right and apply VND to the new window when VND has converged in the previous window, until the window reaches the end position of the solution (window [241, 300] in this case) and VND has converged in this last window. Up to this point, we have done one pass of swVND which is line 3 of Algorithm 13 or Algorithm 12. However, the whole improvement process has not yet been done since we can move the window to the start of the sequence again (the long backward arrow in Figure 3.2 and the line 4 while loop in Algorithm 13) and repeat a new pass of swVND until the full sequence can no longer be improved by any pass of swVND. At this point, Algorithm 13 terminates and we say that the solution is converged w.r.t. swVND with the parameters $win\_size = 100$ and $slide\_size = 80$.

Algorithm 11: Get lists of window sizes and slide sizes.

```
1  Function GetWinList(x, win_size, slide_size):
2      n ← Length(x);
3      win_start_list ← [ ];
4      win_end_list ← [ ];
5      start ← 1;
6      end ← start + win_size − 1;
7      win_start_list.append(start);
8      win_end_list.append(end);
9      while end < n do
10         start = start + slide_size;
11         end = start + win_size − 1;
12         if end > n then
13             end = n;
14         end
15         win_start_list.append(start);
16         win_end_list.append(end);
17     end
18     return win_start_list, win_end_list
```

Algorithm 12: One pass of swVND.

```
1  Function swVND_1pass(x, win_start_list, win_end_list):
2      for i ← 1 to Length(x) do
3          start = win_start_list(i);
4          end = win_end_list(i);
5          x′ ← apply VND on jobs located from "start" to "end" positions in x;
6          x ← x′;
7      end
8      return x'
```

Algorithm 13: swVND.

```
1 Function swVND(x, win_size, slide_size):
2     win_start_list, win_end_list = GetWinList(x, win_size, slide_size);
3     x' = swVND_1pass(x, win_start_list, win_end_list);
4     while Evaluate(x') < Evaluate(x) do
5         x ← x';
6         x' ← swVND_1pass(x, win_start_list, win_end_list);
7     end
8     return x'
```

There are a couple of reasons why swVND may be well-suited to solve our LS-SMSP problem. First, swVND speeds up the improvement process by taking a good move instead of the best move that may otherwise take too long to find if we apply VND directly to a large-size sequence, so it can reach a better solution within a limited time. Take the Figure 3.2 case study for example. Given 300 jobs to be sequenced, instead of spending $\sim 300^2$ computations for finding one best move in a large search space by direct VND on the complete sequence, we can spend $\sim 100^2$ computations for finding one best move in a smaller search space in swVND which is a good move with respect to the complete sequence. The computation difference may not seem significant in the Figure 3.2 example, but the average actual problem size for LS-SMSP is more than 1000 jobs. In that case, spending $\sim 1000^2$ computations for finding a best move among 1000 jobs is much more expensive than spending $\sim 100^2$ computations for finding a best move among 100 jobs. This is the key that makes swVND improve the solution fast.

In addition, the initial solution obtained by iGCF may have jobs located at relatively suitable region already. We may simply move jobs nearby their original locations to improve the solution efficiently without the need to move these jobs very far away from there original locations to achieve this. Therefore, spending time to find a best move in the full search space in the early improvement phase might be time-consuming and inefficient. Instead, a sliding-window that starts by searching a smaller serach space and then can systematically

increase the search space to find a better move might be beneficial for finding a good solution in a limited time.

The way that swVND increases the search space for finding a better move when no improvement can be made in the previous smaller search space is by enlarging the parameters win_size and $slide\_size$ in swVND when the solution is converged w.r.t. swVND with previous smaller parameters. For example, given 1169 jobs to be sequenced which is one of the actual case studies provided by our industrial partner, we can systematically increase swVND parameters by setting up two parameter lists, i.e., $window\_list = [400, 800, 1169]$ and $slide\_list = [200, 400, 0]$. We first apply swVND with $win\_size = 400$ and $slide\_size = 200$ to improve the initial solution obtained by iGCF. Once the solution converges w.r.t. swVND with this setting, we increase the search space by doubling the parameters and apply swVND with $win\_size = 800$ and $slide\_size = 400$ to the previous converged solution for further improvement. Once the solution converges again, since doubling the window size here $(800 * 2 = 1600 > 1169)$ would basically contain all 1169 jobs, we set the last window size to be 1169. In this case, $slide\_size$ should be equal to 0 because there is no need to slide the window if the window contains the full sequence.

Also, when we reach the point where we apply swVND with $win\_size = n, slide\_size = 0$ to a solution, it means we now apply VND directly to n jobs. When the solution is converged w.r.t. swVND with $win\_size = n, slide\_size = 0$, it is also a converged solution w.r.t. VND on n jobs. Based on our experiments , we found that a series of swVND that starts from a small window size and ends with a full-length window size, which we call sequential swVND, improves the solution more efficiently than applying VND directly to the full length sequence, which we call direct VND in this chapter. Since our LS-SMSP problem imposes a time limit to the solving process and the goal is to find a solution as good as possible by any methods within this time limit, the improvement efficiency is the key aspect when we consider which method we should apply to improve the solution. Sequential swVND turns out to be a good method to solve LS-SMSP, where the details of the experimental results and discussion can be seen in Section 3.4.3.

3.3.2 Multiprocessing

The computation work for finding a best move in a window in swVND or for finding a best move in the full-length sequence in VND ($O(n^2)$) can be completed faster by distributing the work to multiple cores in a computer and processing the distributed work simultaneously, also known as multiprocessing in computing. In our algorithm, we apply multiprocessing to find a best neighbor when win_size or n is large. More sophisticated strategies to parallelize VNS-related algorithms can be found in Davidović and Crainic [2013].

A simple example to show the idea of multiprocessing is described as follows. Given $n = 800$ and a 8-core computer for finding a best neighbor in a neighborhood, instead of doing $\sim 800 \times 800$ computations to search for all possible neighbors on a single core, the total number of the computations can be divided into the eight different cores where each core only processes $\sim 100 \times 800$ computations simultaneously. For example, the first core computes all the possible insertions from the first 100 out of 800 jobs, and the second core computes all the possible insertions from the next 100 out of 800 jobs, and so on. When all the computations are done, each core outputs each best solution found during the computing, and the best solution out of the eight solutions simply is the best neighbor in the neighborhood. This multiprocessing technique can be implemented in swVND and VND, and the performance can be scaled with a computer with different number of CPUs.

However, multiprocessing can be less efficient than single-core processing when the number of computations is small due to the overhead in multiprocessing. The exact amount of overhead time may vary from different hardware, programming language, and operating system that a user uses. Based on the performance of our computer, we activate multiprocessing when $win\_size \geq 100$ in swVND or $n \geq 100$ in VND.

Finally, after introducing swVND and multiprocessing where the two methods make the improvements done by VND on a solution more efficient, we illustrate the complete algorithm which we call swVND-SMS for solving LS-SMSP in Figure 3.3. We can see that from this flowchart, if the number of jobs to be sequenced (n) is less than 400, we simply apply VND instead of swVND to improve the initial solution obtained by iGCF since the

problem size in this case is not large. However, a typical problem faced by our industrial partner contains more than 1000 jobs. In this typical case, we apply sequential swVND for efficient improvements followed by direct VND, if time allows, for final refinements to solve the problem instead. In swVND, we only use insertion neighborhood and not swap or other neighborhoods for the sake of speed improvement, but it does not mean that if one would like to implement our algorithm, swap or other customized neighborhoods in swVND cannot be included. A similar concept is applied to VND. We only use insertion and swap neighborhoods in direct VND, but one can add more neighborhoods to improve the solution further given a longer computation time limit. Last, the typical time limit required for the on-site implementation is 1 hour for LS-SMSP, so when it comes to the actual implementation, we can set a timer in swVND-SMS and return the best solution we have found so far when the time limit is reached even if the solution has not converged by that time.

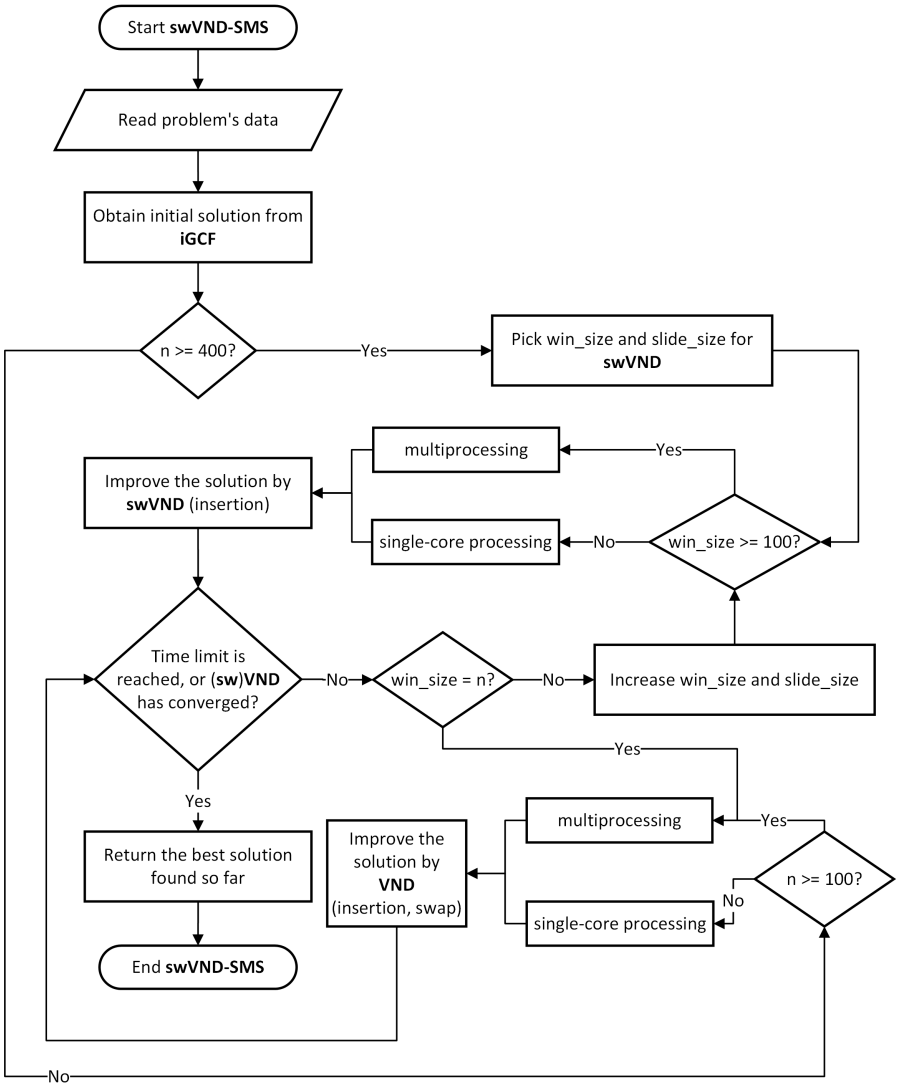

Figure 3.3: Flowchart of the swVND algorithm.

3.4 Computational Results and Discussion

3.4.1 MIQP vs. Metaheuristics

In this section, we compare the performance of MIQP with the performance of our swVND-SMS algorithm (Figure 3.3) on some test data. We will use "MIQP" to refer to the same

mathematical formulation introduced in Section 3.1.2 which we use to solve the test problems. Each test dataset is limited to only 10 or 30 jobs which is much less than the typical actual dataset that contains more than 1000 jobs, because we found that MIQP cannot solve our LS-SMSP problem with just 30 jobs in a reasonable time already, let alone letting it solve the problem with more than 30 or even 1000 jobs. In addition, each test dataset is generated by randomly selecting 10 or 30 jobs from the actual dataset provided by the industrial partner which contains 1169 jobs, following the same ratio of the number of available jobs at time 0 to the number of late-arrival jobs in the actual dataset. In order to avoid the cases where some randomly selected jobs have very late arrival dates so that no feasible solution can be found by the solver (discussed in Section 3.1.2), we revise the arrival dates of late-arrival jobs in the test data so that no jobs will arrive later than 50% of the total job processing time. In this study, 10 samples are generated for the 10-job case and 5 samples are generated for the 30-job case. Following the flowchart illustrated in Figure 3.3, for solving the test problems that involve only a small number of jobs, our swVND-SMS algorithm applies VND right after iGCF without swVND since the improvement process can be achieved by direct VND fast enough and the scale of the problem is small enough that does not require decomposition.

The experiments in this subsection were conducted on Windows 10 Pro 64-bit with an Intel Core i7-10700 CPU (8-core) and 16GB RAM. The MIQP model was programmed in Gurobi Python API and solved by Gurobi 9.5. The swVND-SMS algorithm was coded in Python with NumPy and Numba packages where the latter acts as a Python just-in-time compiler that compiles computational heavy functions coded with NumPy in machine code and makes our metaheuristic algorithm run faster. The details of Numba can be found in Lam *et al.* [2015]. The computational results from the two methods are summarized in Table 3.2 and 3.3.

The %Diff values in Table 3.2 and 3.3 are computed by the following equation:

$$\%Diff = \frac{(swVND\text{–}SMS\ Objective)-(MIQP\ Objective)}{MIQP\ Objective}$$

A positive %Diff means that the swVND-SMS algorithm converges to a worse solution than

Table 3.2: Comparing MIQP with swVND-SMS for LS-SMSP with 10 jobs.

Sample	MIQP			swVND-SMS		
	Obj. Val.	Run Time (sec)	%Gap	Obj. Val.	Run Time (sec)	%Diff
1	29,329.3	8.2	0	29,329.3	0.003	0
2	13,330.9	53.7	0	14,128.6	0.003	5.98
3	26,929.4	10.8	0	37,126.4	0.002	37.9
4	25,289.2	5.6	0	25,289.2	0.003	0
5	30,579.0	4.6	0	30,977.8	0.002	1.3
6	32,480.7	6.1	0	32,480.7	0.001	0
7	31,359.9	6.0	0	31,760.4	0.002	1.28
8	25,110.0	43.9	0	25,110.0	0.003	0
9	27,208.9	8.9	0	31,208.0	0.004	14.7
10	31,806.1	6.1	0	32,606.9	0.002	2.52
Average	-	-	-	-	-	6.36

Table 3.3: Comparing MIQP with swVND-SMS for LS-SMSP with 30 jobs.

Sample	MIQP			swVND-SMS		
	Obj. Val.	Run Time (sec)	%Gap	Obj. Val.	Run Time (sec)	%Diff
1	86,562.0	7200	90	33,358.0	0.02	-61.5
2	97,315.0	7200	85	41,105.3	0.02	-57.8
3	149,969.4	7200	95	65,954.5	0.02	-56.0
4	68,579.6	7200	88	39,771.3	0.04	-42.0
5	95,034.7	7200	83	60,424.9	0.04	-36.4
Average	-	-	-	-	-	-50.7

the MIQP solution, and a negative %Diff means that swVND-SMS algorithm converges to a better solution than the MIQP solution. According to Table 3.2, MIQP is able to find the optimal solution to the LS-SMSP problem with 10 jobs within a minute since the %Gap values are 0 for all of the 10 samples. This also means that %Diff directly reflects how far a swVND-SMS solution is away from the optimal solution in the 10-job case study. The

worst case for the swVND-SMS algorithm is sample 4, where the solution from swVND-SMS prematurely converges at a value 37.9% worse than the optimal value. However, swVND-SMS is able to find the optimal solutions from 4 out of the 10 samples, and on average a solution obtained from swVND-SMS is just 6.36% worse than the optimal solution with more than 1000 times faster solution speed than MIQP.

It is worth noting that we only apply insertion and swap neighborhoods in swVND-SMS since the algorithm is already hard to converge with respect to just the 2 neighborhoods within the time limit when solving the actual large-scale problem. If we were to apply more neighborhoods in VND or introduce a stochastic search step before the local search step in VND, i.e., applying VNS instead of VND, the revised metaheuristics may find a even better solution for the 10-job case study given a long enough solution time. Here, we would just like to show that with 2 simple but effective neighborhoods used in VND, we are able to find a good solution to to the test problems efficiently.

From Table 3.3, we can see that MIQP is not able to solve or find a good solution to our LS-SMSP problem with just 30 jobs within a reasonable amount of time. We capped the time limit in MIQP to 2 hours and compare the solutions obtained from 2-hr MIQP with the ones from swVND-SMS. The results show that the solutions found by swVND-SMS that takes less than 0.05 seconds to converge are 50.7% better than the solutions obtained by 2-hr MIQP on average. In addition, swVND-SMS solutions are all better than the 2-hr MIQP solutions for all 5 samples. This again shows the effectiveness of our algorithm for solving the LS-SMSP problem. Given the actual LS-SMSP problem size is more than 1000 jobs which is much greater than the test problems that only contain 10 or 30 jobs, we have shown that MIQP is not an effective way to address our large-scale problem compared to the metaheuristics method.

3.4.2 Performance of iGCF

For the remaining subsections in Section 3.4, all the experiments were run on the same computer described in Section 3.4.1 but on a different operating system - WSL2 with Linux

Ubuntu 20.04.3 where the algorithms are still coded in Python utilizing Numpy and Numba packages. We switched to Linux operating system for running experiments associated with swVND-SMS since some parts of the algorithm utilize the multiprocessing technique and we found that Python multiprocessing ran faster on Linux OS than on Windows OS.

Three actual datasets with different numbers of jobs to be scheduled and different distributions of job characteristics depending on the daily operation were extracted from the database system in the steel plant and provided by our industrial partner for running the rest of the experiments in Section 3.4. Specifically, dataset 1 (d1) contains 1169 jobs which is the typical problem size they face in the plant, dataset 2 (d2) contains 787 jobs which is on the lower-end of the problem size, and data 3 (d3) contains 1414 job which is on the high-end of the problem size.

In order to determine the performance of iGCF algorithm where the algorithm output depends on an input that is a random permutation of a full-length sequence given a dataset, replication is required in the experiment to draw the statistical results. Thus, 10 replicates for each of the 3 datasets were conducted, where each replicate started with a randomly permuted sequence followed by the iGCF algorithm. The statistical results including mean and standard deviation (std) of iGCF performance for the 3 different datasets are reported in Table 3.4, and the improvement process made by each iteration in the iGCF algorithm for each replicate is plotted in Figure 3.4.

Table 3.4: iGCF performance on 3 actual datasets.

Data #	n	Randomly Permuted Sequence (mean ± std)	After iGCF (mean ± std)	Time (sec) (mean ± std)
d1	1169	13,222,630 ± 291,881	146,292 ± 11,977	60 ± 20
d2	787	7,738,739 ± 210,234	94,824 ± 8,468	17 ± 5
d3	1414	13,268,300 ± 214,357	133,134 ± 8,888	93 ± 22

Given dataset 1 for example, a randomly permuted sequence has an average objective value on the order of 10^7 as shown in Table 3.4. After iGCF, the objective is improved to an order

of 10^5 which means the algorithm is able to improve a random solution by 2 magnitude in just a minute on average. The efficient improvement made by iGCF is consistent across all of the 3 datasets, where the time it takes for the algorithm to terminate scales with the number of jobs to be sequenced given a dataset.

In Figure 3.4, the three columns of plots starting from left to the right are plots for d1, d2, and d3. Here, we use "r" to represent the replicate number for the specific dataset. For example, "d1-r1" represents the replicate 1 in dataset 1. A blue dot in each sub-plot indicates the objective value of the solution constructed by GCF in each iteration in iGCF. The algorithm terminates when a GCF step results in a poorer objective function value (orange dots in the plots). The final solution is then taken from the previous iterate which yielded a lower objective function value. We can see that for any cases, the first solution returned by GCF can be further improved by GCF again. In many cases, the iterative GCF helps to improve the solution a couple of more times until a worse solution compared to the previous ones is found (orange dot), which shows that iGCF is an efficient way to construct a good initial solution for metaheuristics to refine the solution further.

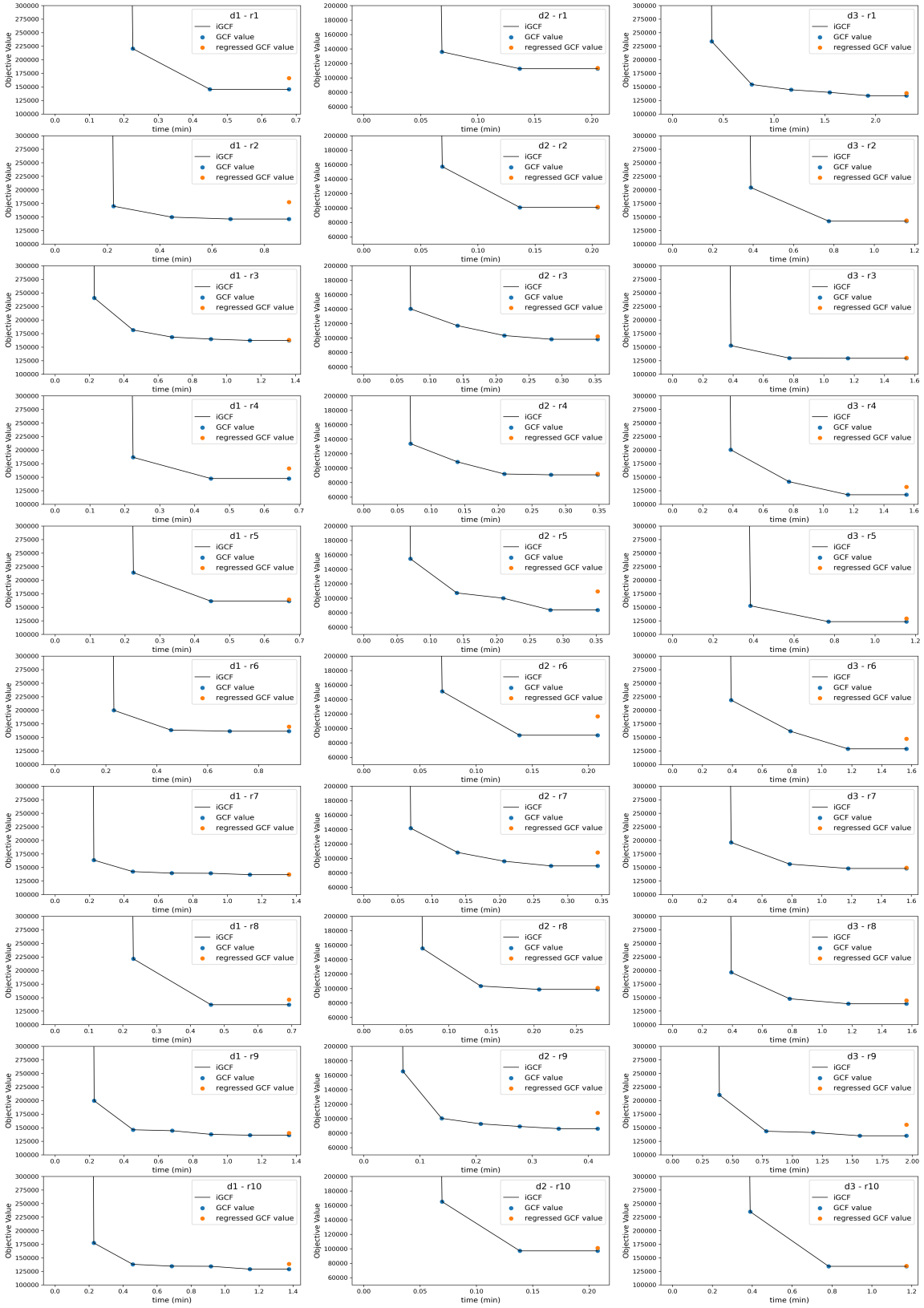

Figure 3.4: iGCF algorithm's performance on 10 replicates for each of the 3 datasets.

3.4.3 Sequential Sliding-Window VND vs. direct VND

In this subsection, we compare sequential sliding-window VND with direct VND by designing and conducting statistical experiments. We first introduce the design of experiments, and then we present the statistical analysis on the experimental results.

Design of Experiments

To compare swVND with VND, we first analyze swVND with its parameters. As described in Section 3.3, two parameters *win_size* and *slide_size* are used in the swVND algorithm, and a user can specify a series of window and slide sizes, which we call them *window_list* and *slide_list* to execute sequential swVND. Therefore, *window_list* and *slide_list* can be thought of as the two parameters to sequential swVND. Since there are infinite combinations of these two parameters and it would be impossible to test all of them, we applied a design of experiment concept where we define a low and high value for each parameter, and conduct experiments for testing the combination of the low and high parameters that are summarized in Table 3.5.

Test #	Parameter 1 (*window_list*)	Parameter 2 (*slide_list*)
t1	-1 (low)	-1 (low)
t2	-1 (low)	+1 (high)
t3	+1 (high)	-1 (low)
t4	+1 (high)	+1 (high)

Table 3.5: Design of experiment for swVND.

According to Table 3.5, the four tests, t1 to t4, assign the sequential swVND method with different designed parameters where "-1" represents the low value while "+1" represents the high value. Given the number of jobs to be sequenced in a dataset, the low value of *window_list* means we start with $win\_size = 50$ as an input to swVND, and double that

amount each time the current swVND converges until the doubled size exceeds the total number of jobs n, in which case we assign n as the final *win_size* in *window_list*. For the high value of *window_list*, we start with *win_size* = 400 directly and double it until the *win_size* is larger than n, the same as the logic applied to the low value case.

For the second parameter *slide_list*, the low value of *slide_list* is set to *slide_size* = $\frac{current\ win\_size}{2}$ unless the current *win_size* is equal to n, in which case there is no need to slide the window that contains full search space further so we simply assign 0 as the final *slide_size* in *slide_list*. A concrete example of the design for the low and high parameter values is given for dataset 1, shown in Table 3.6. The same strategy for setting up the parameters is applied to dataset 2 and 3.

Test #	*window_list*	*slide_list*
t0	[n]	[0]
t1	[50, 100, 200, 400, 800, n]	[25, 50, 100, 200, 400, 0]
t2	[50, 100, 200, 400, 800, n]	[50, 100, 200, 400, 800, 0]
t3	[400, 800, n]	[200, 400, 0]
t4	[400, 800, n]	[400, 800, 0]

Table 3.6: Design of experiment for dataset 1 where n is equal to 1169.

From Table 3.6, we can see that an extra test case, t0, is included in the experimental design. t0 is simply the test case for direct VND, where we can view it as a special case of sequential swVND in a way that its starting *win_size* is directly equal to n and *slide_size* is equal to 0. Following such an experimental design and conducting appropriate statistical experiments, we are able to compare the parameter effects within the sequential swVND method and compare the performance of sequential swVND with direct VND at the same time.

Specifically, we outline the detailed experiments we conducted in Figure 3.5. Given dataset 1 for example, 10 different randomly permuted sequences were first generated followed by iGCF to acquire the 10 different initial solutions as described in Section 3.4.2. Each initial

solution then can be further improved by 5 different methods: t0 to t4 that we just discussed. In the first stage of the improvement process by t0 to t4, only the insertion neighborhood (I) was applied in each method. Once a test method was converged, the second-stage improvement process by VND with insertion and swap neighborhoods (I+S) was applied to further refine a solution. The same experimental design was applied to dataset 2 and 3 where each dataset had 10 replicates for the statistical experiments.

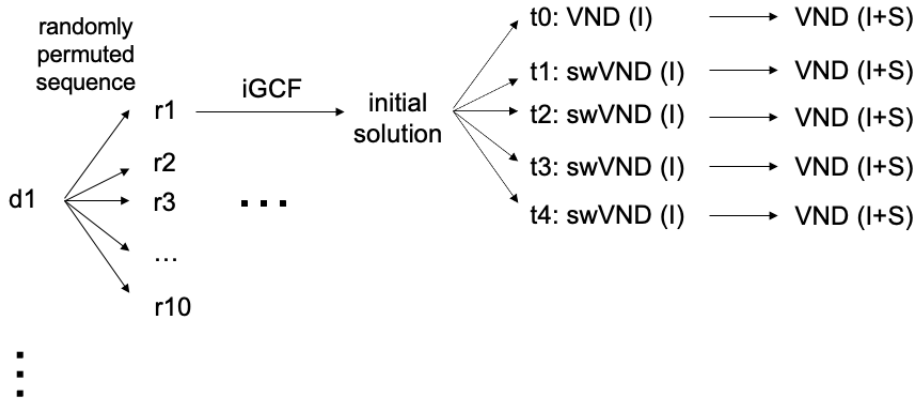

Figure 3.5: Design of experiment for comparing sequential swVND with different parameters and with direct VND.

What we were interested in comparing here was the following:

1. The performance of VND versus the performance of sequential swVND in the first stage of the improvement process.

2. How the parameters *window_list* and *slide_list* in sequential swVND affected its performance, and whether there were any interactions involved between the two parameters.

3. Whether the different methods applied in the first stage of the improvement process affected the overall outcome which includes the second stage of the improvement process.

4. Whether the above results found from one dataset held for another.

We will answer these questions in the remaining subsections by showing the results and statistical analysis that we conducted.

Background of Statistical Tests Applied in this Work

In this section, we give a brief introduction to the statistical tests that we applied to analyze our experimental results and the most relevant statistics knowledge for the readers who are not familiar with this area. This section is based on the book for statistics for engineers and scientists by Walpole *et al.* [2012], and paper for reviewing Student's t-test, analysis of variance, and covariance by Mishra *et al.* [2019]. We will rephrase the most relevant concepts from these literature sources below.

- **Two-sample *t*-test**: A two-sample t-test compares whether the two population means are significantly different. The null hypothesis states that the two means are the same ($H_0 : \mu_1 - \mu_2 = 0$), while the alternative hypothesis states that the two means are different ($H_1 : \mu_1 - \mu_2 \neq 0$). A t-distribution based on the null hypothesis can be drawn and shown in Figure 3.6. A t-statistic can be computed based on the average and standard deviation of each sample, using a formula based on on the relationship between the two samples, such as whether the two population variances are equal and whether the observations are paired. Based on this t-statistic, the probability of obtaining a value of t as large as or larger than the computed t (the shaded areas in the t-distribution shown in Figure 3.6), also known as the p-value, can be calculated. The smaller the p-value, the less likely the null hypothesis is true and the larger the difference between the two means. A common way to judge if the mean difference is significant is to see if the p-value is less than 0.05. If this is the case, we are 95% confident that the difference between the two means are statistically significant.

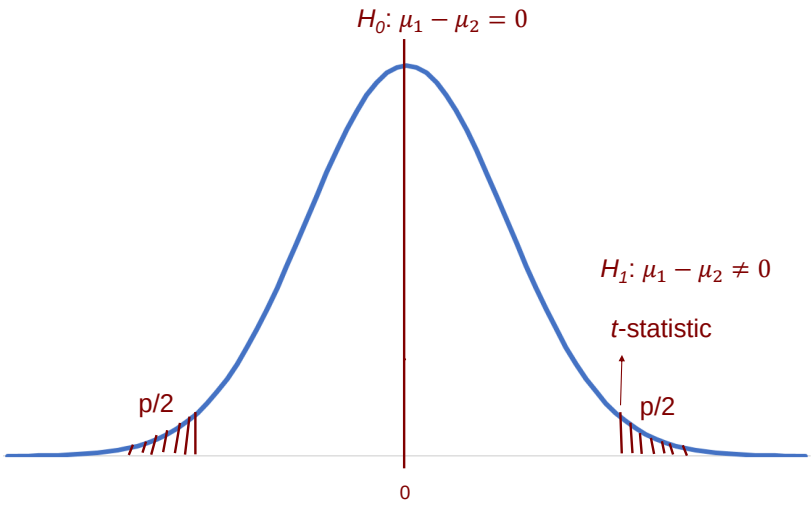

Figure 3.6: Illustration of p-value for two-sample t-test.

- **Analysis of Variance (ANOVA)**: ANOVA compares the means of three or more groups. The statistical concept used in ANOVA is similar to the one applied in a two-sample t-test where the latter is limited to comparing the two means. Instead of calculating a t-statistic and obtaining the p-value from the t-distribution as a two-sample t-test does, ANOVA calculates a f-statistic based on the averages and standard deviations of the groups and obtains the p-value from the F-distribution under a specific null hypothesis. Depending on the number of factors (independent variables) involved in an experiment, ANOVA can be categorized into one-way ANOVA and two-way ANOVA.

  - **One-Way ANOVA**: The one-way ANOVA involves only one factor with k ($k \geq 3$) different groups. If the p-value of a one-way ANOVA test is significantly small, we are confident that out of all the group means, at least one pair of the means are significantly different from each other. However, to determine which pairs of the groups have significantly different means, a post-hoc test, also called multiple comparisons, needs to be conducted (Walpole *et al.* [2012]). A post-hoc test is similar to conducting multiple pair-wise t-test with adjustments

to reduce the errors from multiple comparisons, making the computed p-values more conservative.

- **Two-Way ANOVA**: The two-way ANOVA involves two factors (independent variables) and the goal is to see whether there are interactions between these factors affecting the outcomes of the dependent variable. We can determine whether the interaction between the two factors are significant, and whether a factor affects the outcomes of the dependent variable significantly, by observing the computed p-values for them.

After providing the background to these statistical tests, we now proceed by showing the results of statistical analysis on our experiments where the nuances within each statistical test will be discussed further in the following sections.

Results: One-Way ANOVA with Repeated Measures

The improvement made by t0 to t4 in the first-stage process and VND (I+S) in the second-stage process for each replicate in each dataset can be seen in Figure 3.7. Each column of the plots in Figure 3.7 from left to right represent 10 replicate plots for d1, d2, and d3. Take "d1-r1" (first replicate in dataset 1) subplot in this figure for example. All of the test cases start with the same initial solution constructed by iGCF, so the plot is basically the continuation of the "d1-r1" plot in Figure 3.4. The black solid line indicates the improvement made by t0 (direct VND) in the first-stage improvement process, while other solid lines with specific colors indicate the improvement made by t1 to t4 (sequential swVND with specific parameters) in the first-stage improvement process. A circle on a colored solid line represents a converged value with respect to the current win_size and $slide\_size$ in the sequential swVND method. The dashed lines after the solid lines regardless of their colors represent the improvement made by the second-stage common improvement method - VND with insertion and swap neighborhoods.

We can see that the comparison of the performance made by different test methods in a replicate of a dataset does not necessarily hold for another replicate from the same dataset.

For example, the "d1-r1" subplot in Figure 3.7 shows that t1 (blue line) improves the initial solution more efficiently than t0 (black line) in this case since the objective value of the blue line goes down much quicker than the black line and the blue line converges at a value lower than the black line in the end. However, in the "d1-r3" subplot we can see the comparison is different in this case: the black line (t0) is below the blue line (t1) by the end of the convergence of the blue line, so t0 actually improves the initial solution more efficiently for this replicate. The reason for this comparison difference is that a different randomly permuted sequence passed as an input into the our overall algorithm, swVND-SMS, would lead to a different output. In some cases one method or parameter setting used in swVND-SMS may outperform another, but what we want to generalize is whether in most of the cases one method or a specific parameter setting outperforms other methods, and that is why running experiments with multiple replicates and conducting statistical analysis on the results are required in our study.

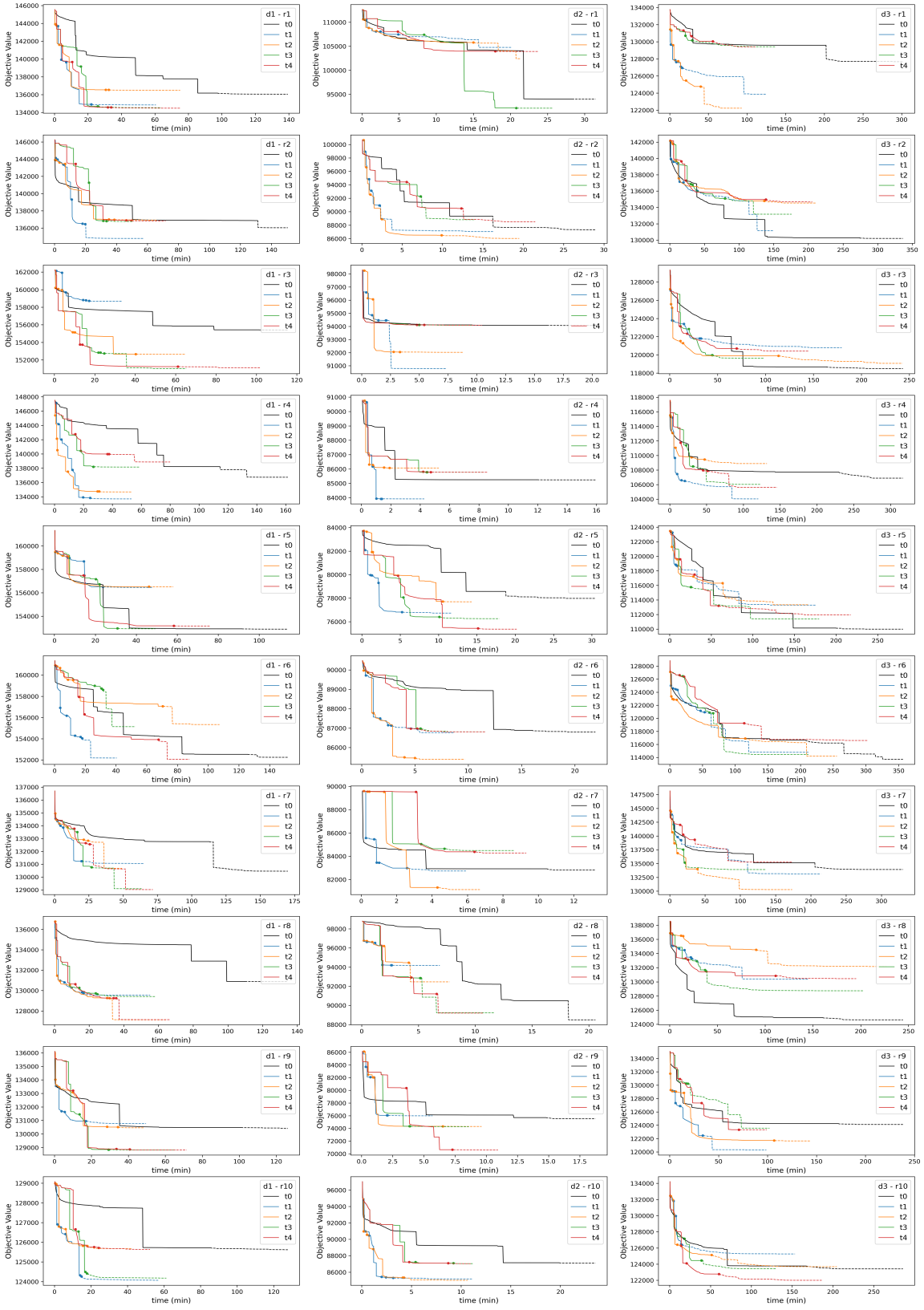

Figure 3.7: Figures of improvements made by sequential swVND and direct VND for each of 10 replicates per dataset.

In particular, the four metrics we used to compare the performance between t0 to t4 are: rate of improvement in the first-stage process (Rate I), rate of improvement including the first and second stage processes (Rate IS), convergence value of the first-stage improvement process (Convergence I), and the convergence value of the second-stage improvement process (Convergence IS). Rate I and Rate IS are calculated as follows:

$$Rate\ I = \frac{(Objective\ of\ initial\ solution\ constructed\ by\ iGCF) - (Objective\ of\ stage1\ converged\ solution)}{(timestamp\ of\ stage1\ convergence) - (timestamp\ of\ iGCF\ convergence)}$$

$$Rate\ IS = \frac{(Objective\ of\ initial\ solution\ constructed\ by\ iGCF) - (Objective\ of\ stage2\ converged\ solution)}{(timestamp\ of\ stage2\ convergence) - (timestamp\ of\ iGCF\ convergence)}$$

The comparison of each of four metrics among t0 to t4 given a dataset can be visualized in a box plot shown in Figure 3.8. As before, each column of plots represents the plots for a dataset. Here, each row of plots represents the box plots of a specific metric across the 3 different datasets. The colors used in a box plot are consistent with the colors used to represent t0 to t4 in Figure 3.7. For example, the black color represents the t0 results. It is also worth noting that the dots with the diamond shape in a box plot represent the outliers that differ significantly from the rest of the data in the test.

Simply by inspection of Figure 3.8, we can observe some obvious trends. For example, t0 appears to have lower improvement rates (both Rate I and Rate IS) on average compared to the rest of the methods and this observation is consistent across the three datasets, since the black boxes are lower than the colored boxes in box plots for Rate I and Rate IS. On the other hand, it seems that there are no significant differences of Convergence I and Convergence IS among the 5 tests for each dataset, since the populations of the tests, or the spreads of the boxes are across each other in these plots. Based on these two observations, we can suspect that in general, the sequential swVND has a better improvement rate than direct VND with no significant difference in the converged values in the end, which implies the former improves a initial solution more efficiently than the latter. In fact, we applied one-way ANOVA with repeated measures to support our observations.

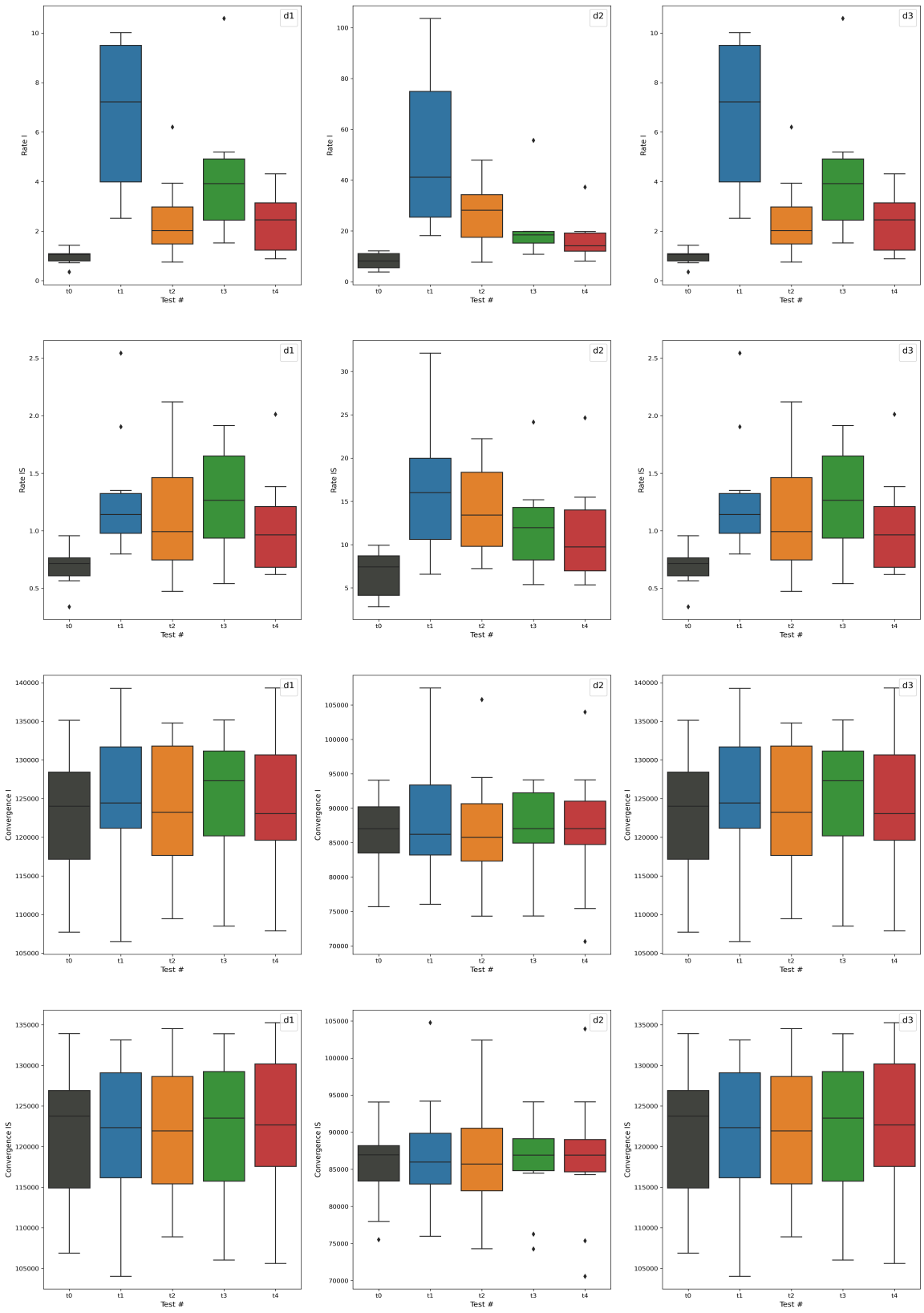

Figure 3.8: Box plots for Rate I, Rate IS, Convergence I, and Convergence IS for each dataset.

The statistics software we applied to conduct statistical tests is the open-source Python library: *Pingouin* (Vallat [2018]). Using its function for conducting one-way repeated measures ANOVA, we acquired a p-value or a corrected p-value for each statistical test, where the correction was based on, for example, whether the assumption of equal variances of differences between the groups were violated in ANOVA so the Greenhouse-Geisser correction may be applied to make a p-value more conservative. Thus, the p-values we report in our statistical tests are the corrected p-values, if applicable, for being the more conservative of statistical test.

The reason we applied one-way repeated measures ANOVA here was because this statistical test helped us to identify whether there was a significant difference between the means of the test groups. If the p-value of the test is less than 0.05, it means there is at least one group that has a mean different than another statistically significantly at 95% confidence level (Walpole *et al.* [2012]). In addition, the term "repeated measures" here means that the same subject is measured under different test conditions, i.e., the same replicate of initial solution constructed by iGCF is improved by different test methods (t0 to t4) where each resulting outcome is measured. It is different from the ordinary one-way ANOVA where different test methods use different random replicates for running the experiment. The results of one-way repeated measure ANOVA for the four metrics we are focusing on for the 3 different datasets can be summarized in Table 3.7.

Dependent Variable	d1 (p-value)	d2 (p-value)	d3 (p-value)
Rate I	0.000001	0.0006	0.000007
Rate IS	0.000002	0.002	0.004
Convergence I	0.90	0.60	0.023
Convergence IS	0.63	0.73	0.36

Table 3.7: p-values (corrected if applicable) of one-way repeated measures ANOVA for Rate I, Rate IS, Convergence I, and Convergence IS for 3 different datasets.

From Table 3.7, we can see that the p-values of one-way repeated measures ANOVA for

Rate I and Rate IS for all of the three datasets are all much lower than 0.05. This means that we are 95% confident that the mean difference of Rate I or Rate IS among the 5 test methods in any one of the three datasets is statistically significant. This statistical analysis supports the observation we made before where we saw that t0 had obviously lower Rate I and Rate IS distributions than other test methods in Figure 3.8. In addition, we can see that the p-values of one-way repeated measures ANOVA for Convergence I and Convergence IS for all of the three datasets are all much higher than 0.05, except for the case of Convergence I for dataset 3. This means that we are 95% confident that there is no statistically significant difference between the means of Convergence I and Convergence IS among the 5 test methods in any one of the three datasets, except for the case of Convergence I for dataset 3. For dataset 3, although there is statistical significance in the mean difference in Convergence I among the 5 test methods, there is still no statistically significant difference in the means of the overall convergence value (Convergence IS) among the 5 test methods. Combining the two facts where there is significant difference in the improvement rates but no significant difference in the final converged values between the 5 test methods, we can determine whether a test method is superior than another by simply comparing its average improvement rate with another. If a method has a better average improvement rate while converging at approximately the same objective as another method, it is a more efficient method to improve the solution quality. Based on the information we have so far, we can conclude that sequential swVND with parameters listed in t1 to t4 improves an initial solution constructed by iGCF more efficiently than direct VND. However, we have not been able to tell whether a particular parameter setting in sequential swVND is better than another. We will answer this question by conducting post-hoc tests and two-way repeated measures ANOVA described in the next subsections.

Results: Post-hoc Tests

If an ANOVA test is found to be significant, post-hoc tests, or pair-wise multiple comparisons, can be applied to find out which pairs of test groups have significantly different means of the dependent variable Walpole *et al.* [2012]. There are different methods for conducting

post-hoc tests. Here, we chose the Bonferroni method according to Mishra *et al.* [2019] for one-way repeated measures ANOVA. Given a pair of test groups, if the p-value of the post-hoc tests is less than 0.05, we are 95% confident that the two means from these groups are significantly different. The post-hoc tests we use here follows the two-sided hypothesis, meaning if the p-value is significant, the means of the pair of the groups are significantly different. However, the two-sided hypothesis cannot tell us which mean is larger out of the pair. Fortunately, we can use the box plots in Figure 3.8 to justify which mean is larger than the other. The corrected p-values from post hoc tests for Rate I and Rate IS for the 3 datasets are summarized in Table 3.8 and Table 3.9. Since the one-way repeated measures ANOVA for Convergence I and Convergence IS are mostly found to be insignificant, post-hoc tests are not useful for these dependent variables.

Group A	Group B	d1 (p-value)	d2 (p-value)	d3 (p-value)
t0	t1	0.00016	0.014	0.0025
t0	t2	0.0042	0.020	0.044
t0	t3	0.00022	0.041	0.018
t0	t4	0.00048	0.041	0.024
t1	t2	0.0031	0.041	0.0025
t1	t3	0.16	0.029	0.041
t1	t4	0.0023	0.020	0.0022
t2	t3	0.023	0.24	0.041
t2	t4	0.33	0.081	0.68
t3	t4	0.0034	0.11	0.025

Table 3.8: p-values (corrected by step-down method using Bonferroni adjustments) of post hoc tests (pair-wise multiple comparisons) for Rate I for 3 different datasets.

Group A	Group B	d1 (p-value)	d2 (p-value)	d3 (p-value)
t0	t1	0.0032	0.033	0.077
t0	t2	0.010	0.0064	0.30
t0	t3	0.000006	0.010	0.019
t0	t4	0.000084	0.15	0.22
t1	t2	1	0.54	1
t1	t3	1	0.32	1
t1	t4	0.42	0.15	0.22
t2	t3	1	0.54	1
t2	t4	1	0.22	1
t3	t4	0.053	0.54	0.088

Table 3.9: p-values (corrected by step-down method using Bonferroni adjustments) of post hoc tests (pair-wise multiple comparisons) for Rate IS for 3 different datasets.

From Table 3.8, we can see that t0 has the mean of Rate I that is significantly different from the means of t1 to t4, since the corrected p-values for these pairs are much less than 0.05 and the statement holds for all of the 3 datasets. Again, this supports our observation from Figure 3.8 where t0 has apparent lower distributions of Rate I than the rest of the tests. Therefore, sequential swVND in general has a faster improvement rate in the first-stage improvement process than direct VND.

Following the same interpretation for Rate I, from Table 3.9, we can see that t0 has the means of Rate IS that is significantly different from the means of t1 to t4 for d1, from t1 to t3 for d2, from t1 and t3 for d3. Although not all of the post hoc tests support the idea where the improvement rate including first and second stage improvement process of direct VND is different from that of sequential swVND, from the box plots in Figure 3.8 we can still observe that in general t0 has lower distributions of Rate IS than the rest of the tests.

In summary, post hoc tests assist us to confidently describe that direct VND has a different

improvement rate than sequential VND. To further investigate which parameter setting in sequential swVND makes it improve a solution more efficiently, the two-way repeated measures ANOVA can be applied to t1 to t4 for drawing statistical conclusions.

Results: Two-Way ANOVA with Repeated Measures

Before we conduct two-way repeated measures ANOVA, we can use interaction plots (3.9) to visualize how the two parameters used in sequential swVND affect the improvement rates. In an interaction plot, the x-axis shows the first parameter, *window_list*, and the y-axis shows the mean of a dependent variable, i.e, either Rate I or Rate IS in our case. The "-1" represents low value while "1" is high value of a parameter, following the same notation used in Table 3.5. The blue line in an interaction plot indicates the outcomes from a low value of *slide_list*, while the orange line in an interaction plot indicates the outcomes from a high value of *slide_list*.

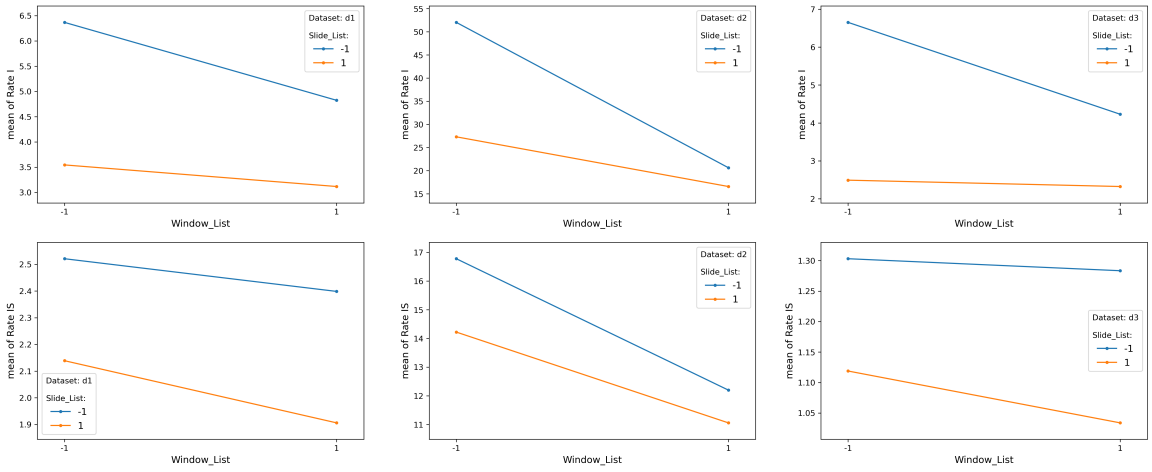

Figure 3.9: Interaction plots for Rate I and Rate IS for each dataset.

For any interaction plot in Figure 3.9, if we fix *window_list* as -1 or 1, a lower *slide_list* gives us a higher mean of Rate I or Rate IS since the blue line is above the orange line. In addition, if we only focus on either the blue or orange line on an interaction plot, a lower *window_list* gives us a higher mean of Rate I or Rate IS. Combining these two observations,

the interaction plots give us an impression that, on average, a lower *window_list* and lower *slide_list* used in sequential swVND improve a solution more effectively than sequential swVND with the high values of the parameters. We now apply two-way repeated measures ANOVA to support this idea.

Instead of analyzing the effect of one independent variable on a dependent variable like what one-way ANOVA does, two-way ANOVA analyzes how two independent variables affect a dependent variable, and whether there is any interaction between the two independent variables affecting the dependent variable Mishra *et al.* [2019]. Since the same initial solution constructed by iGCF is improved by 5 different methods where the outcomes are measured, this concept is classified as repeated measures and we use two-way repeated measures ANOVA to analyze how *window_list* and *slide_list* affect Rate I and Rate IS. The way to interpret the results of this statistical test is based on the p-value, similar to one-way repeated measured ANOVA that we described earlier. The p-values of two-way repeated measures ANOVA for Rate I and Rate IS for 3 datasets are summarized in Table 3.10 and 3.11 below.

Source	d1 (p-value)	d2 (p-value)	d3 (p-value)
window_list	0.062	0.00044	0.015
slide_list	0.0000001	0.0096	0.00013
window_list * *slide_list*	0.22	0.013	0.023

Table 3.10: p-values of two-way repeated measures ANOVA for Rate I for 3 different datasets.

Source	d1 (p-value)	d2 (p-value)	d3 (p-value)
window_list	0.46	0.037	0.72
slide_list	0.039	0.069	0.039
window_list * *slide_list*	0.77	0.52	0.76

Table 3.11: p-values of two-way repeated measures ANOVA for Rate IS for 3 different datasets.

From Table 3.10, we can see that the p-values for *window_list* and *slide_list* are mostly less than 0.05 for all the 3 datasets except for the p-value of *window_list* for d1 which has the value 0.062, a bit higher than 0.05. This means that we are quite confident that the parameters *window_list* and *slide_list* both affect the improvement rate in the first-stage improvement process significantly based on the statistical test. The *window_list* * *slide_list* in this table represents whether there is interaction between the two parameters, for example, when *window_list* is small, small *slide_list* may lead to better improvement rate but when *window_list* is large, magnitude of *slide_list* may not affect the improvement much. The p-values for *window_list* * *slide_list* for d2 and d3 are less than 0.05 while for d1 is more than 0.22, meaning there could be interaction between the two parameters. This result can be supported by seeing the trends in interaction plot in Figure 3.9. We can see that for Rate I interaction plots for d2 and d3, the distance between blue and orange lines is larger when *window_list* is -1 and is smaller when *window_list* is 1. On the other hand, the distance between the blue and orange lines do not change much for Rate I interaction plot for d1. Overall, with visualization from interaction plots and ANOVA, we can say that the lower *window_list* and lower *slide_list* lead to better improvement rate in the first stage of the improvement process.

The similar analysis can be conducted from Table 3.11. For the improvement rate including first and second stage of the improvement process (Rate IS), *slide_list* affects the outcome significantly while *window_list* may have less influence based on their p-values from the ANOVA test. In addition, there is no interaction between the two parameters in

this case, since the p-values for the interaction are all much above 0.05 across the three datasets. Therefore, the *slide_list* parameter in sequential swVND affects both the first-stage and overall improvement rates significantly while the *window_list* parameter in sequential swVND has more influence on the first-stage improvement rate than the overall improvement rate.

Recommendation of Parameters Used for Sequential swVND

Combining the facts from the observations made by visualization and analysis based on ANOVA tests, we conclude that sequential swVND with t1 to t4 parameters improve the same initial solution more efficiently than direct VND on average. Within sequential swVND, a low *window_list* that progress from smaller to larger *win_size* and a low *slide_list* that slides half of the current window make the algorithm improve an initial solution more efficiently on average. Overall, for solving a problem like LS-SMSP, we recommend applying sequential swVND with t1 parameters following iGCF to improve a constructed initial solution further.

3.5 Chapter Summary

This chapter addresses a challenging but practical single machine scheduling problem encountered in steel manufacturing that has the following properties: (1) it is large scale where more than 1000 jobs are to be scheduled, (2) sequence-dependent setup costs and times are involved, (3) earliness or tardiness of a job due to the difference between its due date and scheduled starting time is considered, and (4) a hard constraint where no jobs can be processed before their arrival dates is enforced. The solver is only given one hour to output the best solution it can find by then for practical implementation in industry.

We propose an algorithm where its core is based on variable neighborhood descent and modify it for solving our problem, where we refer to the modified method as sliding-window VND. The algorithm first applies an iterative greedy constructive heuristic to form a fea-

sible initial solution efficiently. The initial solution then is further improved by sequential swVND starting with a small window size and progressing to a larger window size. The converged solution by sequential swVND is further improved by VND with insertion and swap neighborhoods. The multiprocessing technique is used in swVND when the window size is large and in VND when the number of jobs to be sequenced is large. The results from visualization and ANOVA statistical tests show that sequential swVND outperforms direct VND on average, where we recommend a set of parameters used in sequential swVND to make the algorithm more efficient. The proposed algorithm is also readily parallelizable.

Chapter 4

Parallel Machine Scheduling

In this chapter, we introduce a parallel machine scheduling problem faced in the steel industry and propose an algorithm to address such a problem. Section 4.1 describes the specific problem structures of the parallel machine problem we are trying to solve. Section 4.2 demonstrates the algorithm components we propose to address the problem. Section 4.3 shows the performance with statistical analysis of our algorithm. Section 4.4 summarizes the chapter and provides the possible research directions to improve our methods.

4.1 Problem Statement

In this section, we start by describing the details of the parallel machine scheduling problem that we study in this research (Section 4.1.1), and then we discuss how we present a solution to this problem in Section 4.1.2. Last, we dive into how to evaluate a solution step-by-step in Section 4.1.3.

4.1.1 Problem Details

For the large-scale parallel machine scheduling problem (LS-PMSP) that we study in our research, there are n jobs to be processed on m machines. Each job has a unique identifier number and is prespecified to be processed on a specific machine or a set of machines. Each of the jobs also has unique job characteristics: job family number, processing time, arrival date, and due date.

Job family number represents how a job will be processed on a machine and decides the setup cost and time of the two consecutive jobs. If the two consecutive jobs have the same job family number, there will not be any setup cost and time associated with this kind of transition. If the two consecutive jobs have different job family numbers, there will be specific setup cost and time assigned to such a transition depending on the number difference since the machine now has a downtime for the changeover where workers have to be assigned for the machine clean-up and a specific cost is associated with it. Knowing the job family numbers for each job given a dataset, we can compute a $n \times n$ setup cost matrix and $n \times n$ setup time matrix where for example, $sc[1, 2]$ and $st[1, 2]$ indicates the setup cost and time between job 1 and 2. These matricies will be utilized in computing an objective function value of a schedule for a machine.

In LS-PMSP, each job comes with a prespecified processing time which is the same regardless of which machine the job is assigned to. They also have specific arrival dates depending on their upstream processes, so a job cannot be processed before its arrival date. Based on the customer requirement, each job has its own due date to meet. Processing a job earlier than its due date does not cause earliness cost in LS-PMSP, but processing a job later than its due date introduces a tardiness cost in LS-PMSP.

A simple illustration of LS-PMSP is shown in Figure 4.1 below. In this example, 6 jobs are to be scheduled on 2 machines. Jobs 1 and 2 can only be processed on machine 1, jobs 3 and 4 can only be processed on machine 2, and jobs 5 and 6 are flexible jobs that they can be processed on either machine 1 or 2. At the end of the scheduling algorithm, it returns a good processing order (sequence/schedule) of jobs for each machine as the solution to the

scheduling problem. A solution carries an objective function value, or the total cost of the schedule that sums up the cost of each schedule for each machine. We will explain how a solution is represented in LS-PMSP and how an objective function value is calculated for a solution in the following subsections.

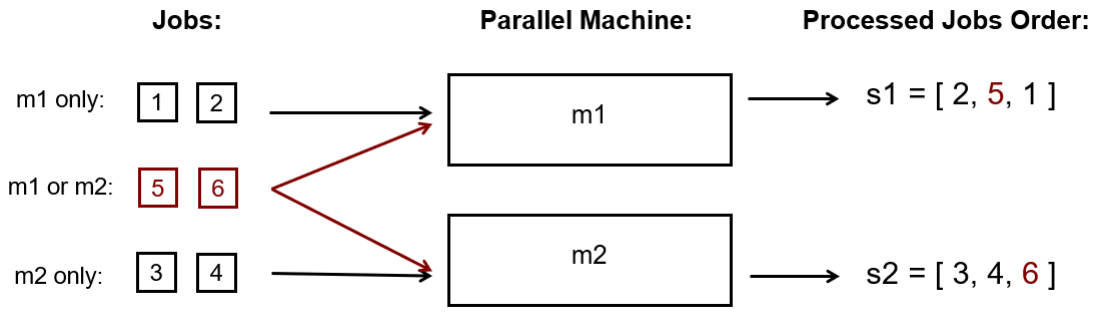

Figure 4.1: Illustration of parallel machine problem.

4.1.2 Solution Representation

A solution to LS-PMSP is represented as $x = \{S_1, S_2, ..., S_M\}$ where x is a master array that contains M arrays (M indicates the total number of machines), and S_m is an array of jobs following the specific processing order for machine m. Continuing the example shown in Figure 4.1, the solution x in this example and the recovered schedule for x are illustrated in Figure 4.2 below.

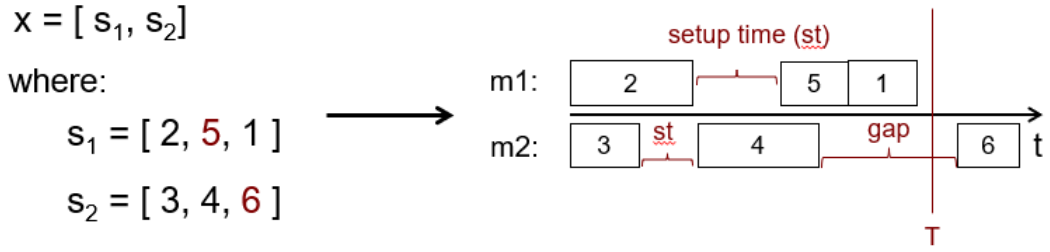

Figure 4.2: Visualization of the solution x.

From Figure 4.2, we can see that the solution to this scheduling problem is represented

as $x = \{S_1, S_2\}$, or specifically $x = \{[2, 5, 1], [3, 4, 6]\}$. This means the solution suggests that machine 1 should process a set of jobs following the order [2,5,1] and machine 2 should process process the other set of jobs following the order [3,4,6]. The actual schedule following these job processing orders for each machine can also be computed, that is, the processing start time and end time of each job in the schedule can be calculated following the solution x.

An example of visualizing the solution x is shown in the right schematic in Figure 4.2. Here, we can see that there are multiple gaps between the processing times of the jobs in this schedule. When computing a start time of a job, our algorithm sets the start time as early as possible so a job will be immediately processed right after the previous job if there is no setup time between them and its arrival date is earlier than the end time of the previous job. The algorithm does not intentionally introduce any gaps in the schedule. Thus, these gaps can be introduced by either the job family difference between the two consecutive jobs processed on the same machine, or the late arrival date of a job that constrains it to be processed at a earlier time. For example, both job 2 and 5 and job 3 and 4 belong to different job families, so there is a setup time involved for each of their changeovers causing the gaps in the schedule. In addition, job 6 has a arrival date that is much later than the processing end time of job 4, so a large gap exists between these two jobs. One of the major objectives for solving LS-PMSP is to reduce the gaps before the critical time, T, in a schedule as much as possible, which we will elaborate more in the next subsection.

4.1.3 Evaluation of a Solution

The quality of a solution x can be evaluated in a programming language easily. The cost of a solution $x = \{S_1, S_2, ..., S_M\}$ basically sums up each of the cost of the sub-schedules (S_m) for each machine, where the cost of a sub-schedule for a machine consists of the three parts: the sum of the costs of the gaps before the critical time T, the sum of the setup costs between the consecutive jobs, and the sum of the tardiness costs of the jobs on this machine. The objective of LS-PMSP is to search for a solution x that minimizes the total

cost of a schedule.

For the gap cost of a schedule, it is assumed to be equivalent to the gap time before the critical time T where other costs of a schedule such as setup cost or tradiness cost are scaled based on a unit of gap cost. We only consider the gap cost that is before the critical time T because for our application, a new batch of jobs will come to the system and rescheduling with the new jobs and the remaining jobs from the previous batch occurs after time T. In other words, we only care about filling in as many jobs as possible to our machines to avoid machine downtime early and not care so much about the later gaps because the new batch of jobs will arrive in the middle of the schedule to fill in the later gaps during the rescheduling.

Continuing our previous example where $x = \{[2, 5, 1], [3, 4, 6]\}$, we now demonstrate the calculation of the objective function value of such a solution assuming the start times and end times of each job on each machine have been computed by programming. Assuming the critical time $T = 80$, only job 6 has a arrival date that is 100 while the rest of the jobs arrive at time 0, and given the setup time and setup cost matrices (they do not have to be symmetrical) precomputed from the input datasets as follows:

$$st_{i,i'} = \begin{bmatrix} 0 & 20 & 20 & 20 & 0 & 20 \\ 20 & 0 & 20 & 20 & 40 & 60 \\ 20 & 20 & 0 & 20 & 0 & 10 \\ 20 & 20 & 20 & 0 & 0 & 20 \\ 0 & 40 & 0 & 0 & 0 & 0 \\ 20 & 60 & 40 & 20 & 0 & 0 \end{bmatrix}, \; sc_{i,i'} = \begin{bmatrix} 0 & 5 & 5 & 5 & 0 & 5 \\ 5 & 0 & 5 & 5 & 10 & 15 \\ 5 & 5 & 0 & 5 & 0 & 10 \\ 5 & 5 & 5 & 0 & 0 & 5 \\ 0 & 10 & 0 & 0 & 0 & 0 \\ 5 & 15 & 10 & 5 & 0 & 0 \end{bmatrix}$$

We can walk through the calculation of the objective value of x step-by-step with the help of visualization of x in Figure 4.2 as follows:

1. Gap costs before T: for machine 1, the gap cost between job 2 and 5 caused by the setup time is $st_{2,5} = 40$. Also, a gap exists between the end time of the last job on machine 1 and the critical time T, and let us say that this gap is $T - end\_time_1 = 80 - 70 = 10$. For machine 2, the gap cost between job 3 and 4 caused by the setup

time is $st_{3,4} = 20$. Although job 6 arrives and is processed later than T and there is a big gap between job 4 and 6, we only consider the gap cost before T, saying such the gap cost is $T - end\_time_4 = 80 - 50 = 30$.

2. Setup costs: the sum of the setup costs for machine 1 is $sc_{2,5} + sc_{5,1} = 10 + 0 = 10$, and the sum of the setup costs for machine 2 is $sc_{3,4} + sc_{4,6} = 20 + 20 = 40$.

3. Tardiness cost: assume all the 6 jobs have very late due date to meet, so the schedule generated by the solution x does not lead to any tardiness cost for either sub-schedule of a machine.

4. Total cost of x: the cost of S_1 is $gap\ costs + setup\ costs + tardiness\ cost = (40 + 10) + 10 + 0 = 60$. The cost of S_2 is $gap\ costs + setup\ costs + tardiness\ cost = (20 + 30) + 40 + 0 = 90$. By summing the costs of all the sub-schedules, the total cost of x is $cost\ of\ S_1 + cost\ of\ S_2 = 60 + 90 = 150$.

The exact value of T and the coefficients for calculating setup time and cost matrices that affects the gap cost, setup cost, and tardiness cost vary from application to application and are confidential. The problem setup of the objective and the arrival-date hard constraint in LS-PMSP makes the scheduling optimization problem unique compared to, for example, the common makespan minimization problem, but it is practical in an industrial setting. The methods that we applied and developed in our research are also generalizable and can be applied to other parallel machine scheduling problems with different objectives and constraints.

4.2 Algorithm Components

We extend the algorithm we developed to solve the large-scale single machine scheduling problem (Chapter 3) to address the large-scale parallel machine scheduling problem faced by our industrial partner. The solution process for solving LS-PMSP can be broken down into initialization and improvement phases. In the initialization phase, we tailor the greedy

constructive heuristic applied to the single machine problem (Algorithm 2) to deal with the parallel machine problem which constructs an initial solution efficiently. Next, we apply the iterative two-stage improvement process to further improve the initial solution obtained by the greedy constructive heuristic, where in each iteration, the first-stage improvement is done by intra-machine VND and the second-stage improvement is done by inter-machine VND. We will describe the initialization and improvement phases in detail in Section 4.2.1 and 4.2.2 respectively.

4.2.1 Initialization: Greedy Constructive Heuristic

The concept of greedy constructive heuristic (GCH) discussed in Section 3.2.1 can also be applied to the parallel machine problem, where a good and feasible initial solution can be constructed from a randomly permuted job sequence efficiently. The pseudo code for the GCH algorithm for the parallel machine problem is shown in Algorithm 14 below. We will walk through how GCH for LS-PMSP works and give a concrete example with the illustration of a figure in this subsection.

Algorithm 14: Greedy constructive heuristic.

```
1  Function GCH(x, waitlist):
2      M ← Length(x);
3      new_vals ← an array of M zeros;
4      new_seqs ← an array of M empty arrays;
5      while waitlist != [ ] do
6          job ← Remove(waitlist);
7          for m ← 1 to M do
8              if Feasible(job, m) then
9                  insert job to the best position in x[m];
10                 new_vals[m] ← Evaluate(x);
11                 new_seqs[m] ← Copy(x[m]);
12                 remove job from x[m];
13             else
14                 new_vals[m] ← a very large value;
15             end
16         end
17         best_m ← argmin_m(new_vals);
18         x[best_m] ← new_seqs[best_m];
19     end
20     return x
```

In Algorithm 14, line 1 shows the two inputs to GCH, where initially x is an array of M empty arrays and *waitlist* is an array that contains all the job IDs to be sequenced with any possible order. *waitlist* could be an array of randomly permuted job IDs or an array of job IDs presorted by any kinds of preferences. The idea of GCH is to iteratively remove the job located at the first position in current *waitlist* (line 5 and 6), and insert it to the current best and feasible position in x (line 7 to 18) until *waitlist* is empty. By that time, x contains good and feasible processing orders for each machine and is returned by GCH (line 20).

Specifically, in line 7 we iterate through every machine sequence in x, and whenever the job removed from the *waitlist* is feasible to be assigned to machine m (line 8), we insert it to the best position in that sequence (line 9) that leads to the lowest sequence cost while keeping the relative order of the rest of the jobs in that sequence the same. We then save the total cost of updated x (line 10) and save a copy of the updated machine sequence (line 11) for later reference. Next, we remove the job that was just inserted to the current machine sequence (line 12) before we try inserting it to the next machine sequence in x and see if that could lead to a lower total cost of the schedule. If the removed job is not feasible to be assigned to machine m, we simply save a very large value to the mth position in the storage array (line 14), so when we compare which machine is best for the removed job to be assigned to (line 17), the infeasible machine(s) are not comparable. In line 18, we update the machine sequence that is the most suitable for the current removed job to be inserted to. Then we continue the loop, remove the next job from *waitlist* (line 5 and 6), and repeat the same procedure for inserting the removed job to the current best and feasible position in x.

An example illustrating how GCH works for a parallel machine scheduling problem with 5 jobs to be sequenced on 2 machines is shown in Figures 4.3 and 4.4. For the illustration purpose, input *waitlist* contains job IDs that happen to be ordered from 1 to 5. Job 1 and 2 can only be assigned to machine 1, job 3 and 4 can only be assigned to machine 2, and job 5 is a flexible job that can be assigned to either machine. At the beginning of GCH, solution x is an array that contains 2 empty arrays because there are 2 machines to be considered in this problem.

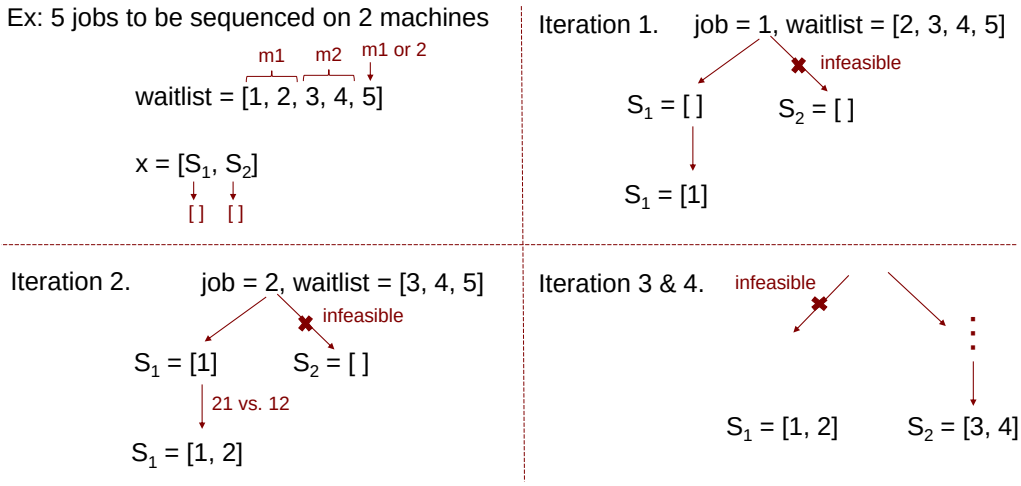

Figure 4.3: Illustration of GCH: part 1.

At the first iteration shown in Figure 4.3, job 1 is removed from *waitlist* and inserted to sequence 1 (processing sequence of machine 1). The algorithm does not allow job 1 to be inserted to sequence 2 because it is not feasible. It simply proceeds with $S_1 = [1]$ and empty S_2 to the next iteration. At the second iteration, job 2 is removed from *waitlist* and inserted to sequence 1 without considering sequence 2 since such an assignment is not feasible. In sequence 1, job 2 can be either inserted before or after job 1. The algorithm proceeds with the best insertion which leads to the lowest total cost of x. In this case, $S_1 = [1, 2]$ has a lower cost than $S_1 = [2, 1]$, so x is updated to be $x = [[1, 2], [\ ]]$ after the second iteration. The third and fourth iterations are similar to the first and second iterations, with the difference that job 3 and 4 can only be assigned to machine 2 instead of machine 1. Let us say that $S_2 = [3, 4]$ has a lower cost than $S_2 = [4, 3]$, so x is updated to be $x = [[1, 2], [3, 4]]$ after the fourth iteration.

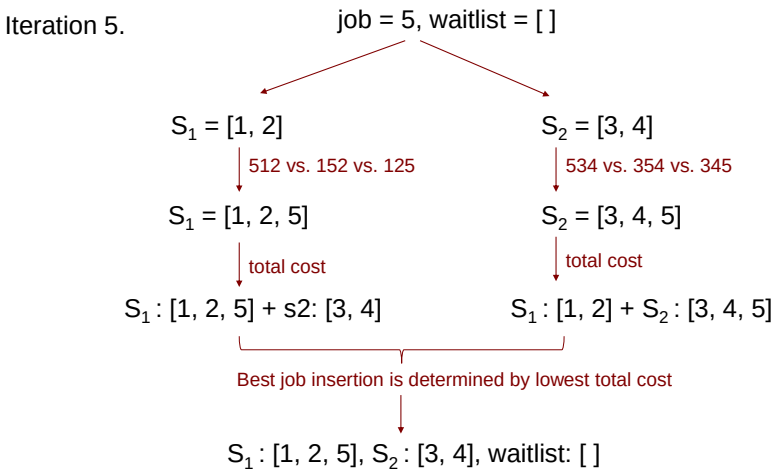

Figure 4.4: Illustration of GCH: part 2.

Finally, we remove job 5 from *waitlist* in the fifth iteration as shown in Figure 4.4. Job 5 is a flexible job, so it can be assigned to either machine 1 or 2. The algorithm first inserts it to the best position in sequence 1 while keeping the relative order of the rest of the jobs in sequence 1 the same. That is, job 5 can be inserted before job 1, between job 1 and 2, or after job 2, but there will not be a combination such as [5,2,1] to be considered here, since the algorithm is "greedy" and keeps the relative order, job 1 before job 2, the same when finding a best insertion position for job 5. Let us say that out of the three possible insertions of job 5 to sequence 1, $S_1 = [1, 2, 5]$ leads to the lowest cost of the sequence, so the algorithm will save the updated S_1 and compute and save the total cost of $x = [[1, 2, 5], [3, 4]]$ for later comparison. After recovering x to be its status at the beginning of iteration 5, i.e., $x = [[1, 2], [3, 4]]$, the algorithm now tries to insert job 5 to sequence 2. Following the same logic, the algorithm finds the best insertion of job 5 into sequence 2 to be $S_2 = [3, 4, 5]$, so it saves the updated S_2 and computes and saves the total cost of $x = [[1, 2,], [3, 4, 5]]$ for comparison. Last, by comparing the cost of $x = [[1, 2, 5], [3, 4]]$ and $x = [[1, 2,], [3, 4, 5]]$, let us say the former has a lower total cost than the latter, so the algorithm updates x to be $x = [[1, 2, 5], [3, 4]]$. Since *waitlist* now is empty, the algorithm terminates and returns

updated x.

The time complexity of GCH is $O(n^2)$ since finding a best insertion position costs $O(n)$ and we repeat it n times for n jobs in *waitlist*. This shows that GCH is an efficient algorithm for constructing an initial solution to LS-PMSP.

4.2.2 Improvement: Variable Neighborhood Descent

An initial solution constructed by GCH can be further improved by other metaheuristics. Having seen the success of applying variable neighborhood descent (VND) to improve a solution for the large-scale single machine scheduling problem (Chapter 3), we extend the VND-based metaheuristic to improve the solution for our large-scale parallel machine scheduling problem. The details of how VND works can be found in Section 3.2.2.

There are many different ways to design the neighborhoods and neighborhood order used in VND, or even coupled VND with other heuristic for solving a parallel machine scheduling problem. For example, Avalos *et al.* [2014] show that a two-stage improvement process where the first stage applies inter-machine movements with insertion and swap neighborhoods followed by intra-machine refinements with a local search method, and the second stage applies a composite movement that involves an insertion or swap move between machines followed by intra-machine optimization refinements, improves the solution efficiently for their parallel-machine scheduling problem.

For our LS-PMSP, we break down the improvement process by VND into two stages. The first stage applies intra-machine VND with insertion and swap neighborhoods on each machine for efficient individual improvements, and there are no interactions between machines in this stage. The second stage applies inter-machine VND with inter-insertion and inter-swap neighborhoods between machines for further improvements to the solution, and no movement within an individual machine in this stage. We then repeat this two-stage improvement process until the solution can be no longer improved by these methods. The details of the methods applied in each stage are described in the following subsections.

Intra-machine VND

In VND algorithm (Algorithm 6), a series of neighborhoods to be searched for have been pre-defined and during each iteration, and the algorithm looks for the "best neighbor" in the current neighborhood to improve the solution (line 4). Unlike single machine scheduling where the neighborhoods are limited to the movements within the single machine, neighborhood structures that involve interaction between machines can also be considered in parallel machine scheduling for further improvement when no improvement can be made by the intra-machine movements. Therefore, the series of neighborhoods to be considered in VND for LS-PMSP should include a combination of intra-machine movements and inter-machine movements. To distinguish the two types of neighborhood structures, we split the holistic VND for LS-PMSP into two parts: an intra-machine VND that includes only the intra-machine neighborhoods and an inter-machine VND that includes just the inter-machine neighborhoods.

For intra-machine VND, the two neighborhoods considered here are simply the intra-machine insertion and intra-machine swap neighborhoods that we introduced earlier in single machine scheduling, where the algorithms for finding a "best neighbor" in these neighborhoods (line 4 in Algorithm 6) have been shown in Algorithm 8 and 10. In intra-machine VND, we iterate through each machine sequence in solution x and apply VND with intra-insertion and intra-swap neighborhoods to improve each machine sequence separately. This process can be thought of as applying single machine VND to each individual machine in solution x. Once solution x can no longer be improved by intra-machine VND, we proceed with inter-machine VND for the second-stage improvement which will be discussed in the next subsection.

The time complexity for finding a best move from the insertion or swap neighborhood from one of the parallel machines is $O(n_m^2)$, where n_m represents the number of jobs on the mth machine. It is much less than $O(n^2)$, which is the time complexity for finding a best insertion or swap move from a single machine that combines all the jobs from the parallel machines. Note that if we split the total n jobs to M parallel machines and find

a best insertion or swap move for each machine, the total computation cost would be $O(n_1^2) + O(n_2^2) + ... + O(n_M^2) <= O(n^2)$, which is still less than finding a best move from a single machine containing all the jobs, if $n \neq M \neq 1$. This implies that intra-VND will actually run faster with more parallel machines being considered, given the same total number of jobs n to be sequenced on these machines. We will see such efficient algorithm performance in Section 4.3.

Inter-machine VND

In inter-machine VND, two neighborhoods, inter-insertion and inter-swap neighborhoods, are utilized in the algorithm. To find a best neighbor (move) in the inter-insertion neighborhood which considers all the possible inter-insertions between all the machines, we first discuss how to pick the best job from one machine and insert it to the best position in the other machine during inter-insertion movements between just the two machines. We call such an interaction between the two machines as finding the best inter-insertion sub-move ("sub-move" indicating interaction between only the two machines and not all the possible machines). The algorithm for finding a best inter-insertion sub-move between the two machines can be seen in Algorithm 15.

Algorithm 15: Best inter-insertion sub-move.

```
1  Function Best_InterInsertion_SubMove(S1, S2):
2      best_S1, best_S2 ← Copy(S1), Copy(S2);
3      best_val ← Evaluate(S1) + Evaluate(S2);
4      for i ← 1 to Length(S1) do
5          if Feasible(S1[i], S2) then      // if job S1[i] can be assigned to S2
6              for j ← 1 to (Length(S2) + 1) do
7                  remove S1[i] from S1, insert it to jth position in S2;
8                  temp_val ← Evaluate(S1) + Evaluate(S2);
9                  if temp_val < best_val then
10                     best_S1, best_S2 ← Copy(S1), Copy(S2);
11                     best_val ← temp_val;
12                 end
13                 remove S2[j] from S2, insert it to ith position in S1;
14             end
15         end
16     end
17     return best_S1, best_S2
```

The inputs to Algorithm 15 are the two arrays, S_1 and S_2, containing the current processing orders of the two machines. In line 2 and 3, we save the current machine sequences and the cost of the two machine sequences. We then iterate through the jobs in the first machine sequence (line 4), and if the job from S_1 is feasible to be assigned to the other machine (line 5), we then iterate through the positions in the other machine (line 6) for seeking cost improvement from inter-insertions. If an inter-insertion leads to a better cost of the solution (line 9), we update the the current best sequences and their total cost (line 10 and 11). By the end of the iterations, we find the best job to be picked from S_1 to be inserted to the best position in S_2, if there is any, and return the updated sequences (line 17).

We can now apply Algorithm *Best_InterInsertion_SubMove* to each pair of machines (the order of machines within a pair matters), and compare these best inter-insertion sub-moves

from these matches to decide the best inter-insertion move from the current inter-insertion neighborhood. Algorithm 16 below demonstrates the details to complete such the task.

Algorithm 16: Best inter-insertion move.

```
1  Function Best_InterInsertionMove(x):
2      best_x ← Copy(x);
3      best_val ← Evaluate(x);
4      for m1 ← 1 to Length(x) do
5          for m2 ← 1 to Length(x) do
6              if m1 != m2 then
7                  old_S1, old_S2 ← Copy(x[m1]), Copy(x[m2]);
8                  x[m1], x[m2] ← Best_InterInsertion_SubMove(x[m1], x[m2]);
9                  temp_val ← Evaluate(x);
10                 if temp_val < best_val then
11                     best_x ← Copy(x);
12                     best_val ← temp_val;
13                 end
14                 x[m1], x[m2] ← old_S1, old_S2;
15             end
16         end
17     end
18     return best_x
```

The input to Algorithm *Best_InterInsertionMove* is the current solution x to be improved (line 1). We first save the current x and its cost as the current best solution and cost for later comparison (line 2 and 3). We then iterate through all the possible pairs of the machine sequences in x where the order of the sequences matters (line 4 and 5). During each iteration, we apply Algorithm *Best_InterInsertion_SubMove* to find the best inter-insertion between the two machines (line 8). If the improvement from the current pair of the machines is better than the current best value, we update the current best solution and value (line 10 to 13). At the end of the algorithm, we find the best inter-insertion move in

the current inter-insertion neighborhood, if there is any, and return the updated solution x (line 18).

Just like intra-VND with insertion and swap neighborhoods, in inter-VND when a solution can no longer be improved by the inter-insertion neighborhood, a further improvement is sought from the inter-swap neighborhood. This switch is controlled by function $Neighborhood\_Change$ in VND (line 5 in Algorithm 6). To find a best neighbor in the inter-swap neighborhood, it follows the similar procedure for finding a best inter-insertion move, where we start by considering finding the best inter-swap sub-move between the two machines and apply such the technique to every pair of the machines in x to determine the best inter-swap move in the current inter-swap neighborhood. The algorithms for $Best\_InterSwap\_SubMove$ and $Best\_InterSwapMove$ can be found in Algorithm 17 and 18 below.

Algorithm 17: Best inter-swap sub-move.

```
1  Function Best_InterSwap_SubMove(S1, S2):
2      best_S1, best_S2 ← Copy(S1), Copy(S2);
3      best_val ← Evaluate(S1) + Evaluate(S2);
4      for i ← 1 to Length(S1) do
5          if Feasible(S1[i], S2) then      // if job S1[i] can be assigned to S2
6              for j ← 1 to Length(S2) do
7                  if Feasible(S2[j], S1) then  // if S2[j] can be assigned to S1
8                      swap S1[i] with S2[j];
9                      temp_val ← Evaluate(S1) + Evaluate(S2);
10                     if temp_val < best_val then
11                         best_S1, best_S2 ← Copy(S1), Copy(S2);
12                         best_val ← temp_val;
13                     end
14                     swap S2[j] with S1[i];
15                 end
16             end
17         end
18     end
19     return best_S1, best_S2
```

Algorithm 18: Best inter-swap move.

```
1  Function Best_InterSwapMove(x):
2      best_x ← Copy(x);
3      best_val ← Evaluate(x);
4      for m1 ← 1 to (Length(x) − 1) do
5          for m2 ← (m1 + 1) to Length(x) do
6              old_S1, old_S2 ← Copy(x[m1]), Copy(x[m2]);
7              x[m1], x[m2] ← Best_InterSwap_SubMove(x[m1], x[m2]);
8              temp_val ← Evaluate(x);
9              if temp_val < best_val then
10                 best_x ← Copy(x);
11                 best_val ← temp_val;
12             end
13             x[m1], x[m2] ← old_S1, old_S2;
14         end
15     end
16     return best_x
```

For *Best_InterSwap_SubMove*, one major difference from *Best_InterInsertion_SubMove* worth mentioning is that since an inter-swap move swaps the two jobs from their current positions in the two machines, when we try out such an movement we need to check whether both jobs are feasible to be assigned to their counterpart machines (line 5 and 7 in Algorithm 17). In addition, when we apply function *Best_InterSwap_SubMove* to each pair of the machines in x to find the best inter-swap move, the order of the machines being considered in a pair does not matter (line 4 and 5 in Algorithm 18) because between the two machines, swapping job a in the first machine with job b in the second machine is the same as swapping job b in the second machine with job a in the first machine. Other than these nuances, the inter-swap algorithms work in a similar fashion as the inter-insertion algorithms.

The time complexity for finding a best inter-insertion move or best inter-swap move is also capped at $O(n^2)$, and should be less than that for most of the cases since:

1. Not every job is feasible to be assigned to another machine, so many inter-insertion or inter-swap moves can be reduced.

2. Within an inter-insertion or inter-swap neighborhood, a job will not consider any intra-insertion or intra-swap move on the machine it is currently assigned to.

3. Finding a best inter-swap move has even less computation cost than finding a best inter-insertion move since swapping job a in the first machine with job b in the second machine is the same as swapping job b in the second machine with job a in the first machine and we do not have to swap a pair of jobs twice.

These again show that sequencing n jobs on M ($M > 2$) parallel machines with intra-VND or inter-VND should be faster than sequencing n jobs on a single machine with VND. The results of algorithm performance for intra-VND and inter-VND can be seen in Section 4.2.2.

4.2.3 Overall Algorithm for Solving LS-PMSP

After discussing the algorithm components, GCH, intra-VND, and inter-VND, we now propose an overall algorithm, PMS, for solving our large-scale parallel machine scheduling problem shown in Algorithm 19 below. The inputs to PMS (line 1) are x which initially is an array of M empty arrays (M stands for the number of machines to be considered), and *waitlist* which is an array that contains the total number of the jobs to be sequenced with any possible processing order. During the initialization phase, we first apply GCH to construct a good and feasible initial solution (line 2). Then, in the improvement phase, we repeatedly apply a two-stage improvement process where the first stage uses intra-VND and the second-stage uses inter-VND to improve x further, until no more improvements can be made by the either method (line 3 to 9). Last, the algorithm returns the converged solution with respect to intra-VND and inter-VND (line 10) as the solution to LS-PMSP. The performance of PMS can be found in the next section.

Algorithm 19: PMS

```
1  Function PMS(x, waitlist):
2      x ← GCH(x, waitlist);
3      x ← intra_VND(x);
4      x′ ← inter_VND(x);
5      while Evaluate(x') < Evaluate(x) do
6          x ← x′;
7          x ← intra_VND(x);
8          x′ ← inter_VND(x);
9      end
10     return x′
```

4.3 Computational Results and Discussion

In this section, we introduce the two questions we are trying to answer and the design of experiments for finding these answers in Section 4.3.1. We analyze the 2 actual industrial datasets received from our industrial partner and consider whether different input dataset characteristics lead to different experimental results in Section 4.3.2. Finally, in Section 4.3.4 and 4.2.2 we analyze the results from our experiments with the support of statistical analysis to answer the two questions we were trying to answer at the beginning.

The experiments in this section were conducted on an Intel Core i7-10700 CPU and 16GB RAM under WSL2 with Linux Ubuntu 20.04.3 where the algorithms are implemented in Python with Numpy and Numba packages.

4.3.1 Design of Experiment

Sorting an Input Sequence for PMS

As shown in Figure 4.5 below, Algorithm PMS can take any possible sequence that contains all the jobs to be scheduled as an input *waitlist* and return an improved solution to LS-PMSP. Although the algorithm is deterministic, that is, given the same inputs it will always return the same output solution x', starting with different *waitlist* may lead to different solution qualities. We are interested in the robustness of PMS, i.e., how the algorithm performs with different inputs.

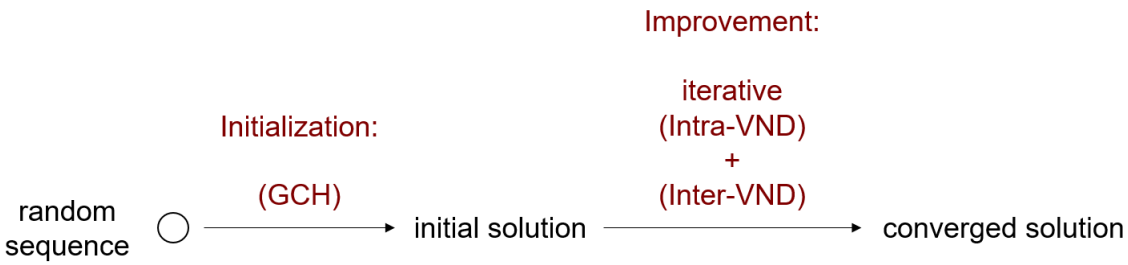

Figure 4.5: Illustration of PMS algorithm

In addition, we are interested in whether sorting input *waitlist* would lead to a better solution statistically. In particular, the sorting we do here follows the order of the machine number of the jobs from low to high, that is, *waitlist* contains the jobs starting from the machine-1 jobs (jobs that can only be assigned to machine 1), followed by machine-2 jobs, and so on. The last group of the jobs in a sorted *waitlist* would be the flexible jobs that can be assigned to any machines. Note that the jobs within the same machine group can be randomly ordered, so our sorting still introduces randomness to an input to the algorithm. The left *waitlist* shown in Figure 4.6 below illustrates the sorting concept. On the other hand, the right *waitlist* in the figure demonstrates a randomly permuted sequence without sorting. The idea of sorting is for GCH to process the jobs that can only be assigned to a specific machine first before considering the flexible jobs. This may or may not lead to a

better converged solution on average after the improvement phase by VND-based methods.

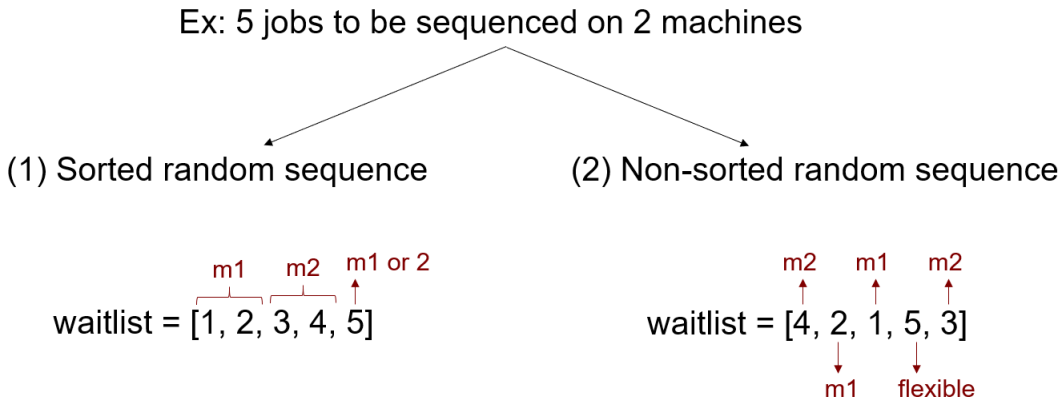

Figure 4.6: Illustration of sorted random sequence versus unsorted random sequence.

To find out whether sorting helps improve the solution quality, a statistical experiment can be conducted to draw a conclusion. Figure 4.7 below illustrates the design of experiment. Given an input dataset "d1", we can generate 10 random sorted *waitlist* and 10 random unsorted *waitlist*. Each replicate goes through the PMS algorithm and has an objective value of the converged solution and the time it takes for the algorithm to converge. We then can conduct t-test on these results and draw a statistical conclusion.

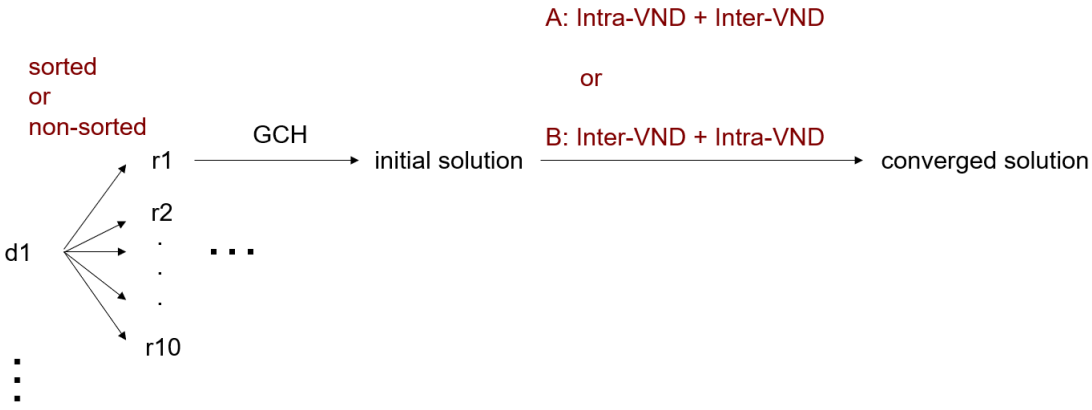

Figure 4.7: Illustration of design of experiment for investigating PMS performance.

Intra-VND or Inter-VND First in Improvement Phase

In addition to sorting, we are interested in investigating whether starting intra-VND or inter-VND first during the improvement phase affects the converged solution quality. The concept is also shown in Figure 4.7, where method A represents that we apply intra-VND first followed by inter-VND during the iterative two-stage improvement phase, and method B represents the improvement order that is the other way around. To compare these two methods, we can generate 10 sorted random sequences as 10 different replicates, where each replicate is improved in two ways: one by GCH and method A and the other by GCH and method B. We then can conduct paired t-test on these results to draw a statistical conclusion.

To summarize, the two questions we are trying to answer from our experiments are:

1. Whether sorting *waitlist* improves a solution quality or converged time.
2. Whether starting intra-VND first (method A) or inter-VND first (method B) leads to a better solution quality or converged time.

We are given 2 actual datasets from our industrial partner to conduct these experiments, where the characteristics of these two datasets will be discussed in the next subsection.

4.3.2 Characteristics of Input Datasets

The characteristics of the two input datasets, where we use "d1" and "d2" to refer to the first and the second dataset, are summarized in the histogram plots in Figure 4.8. The left column of histograms are for d1 characteristics while the right column of histograms are for d2 characteristics in this figure. d1 has 492 jobs and d2 has 395 jobs to be scheduled. The units for arrival date, due date, and processing time are confidential, but the features from these distributions can be analyzed.

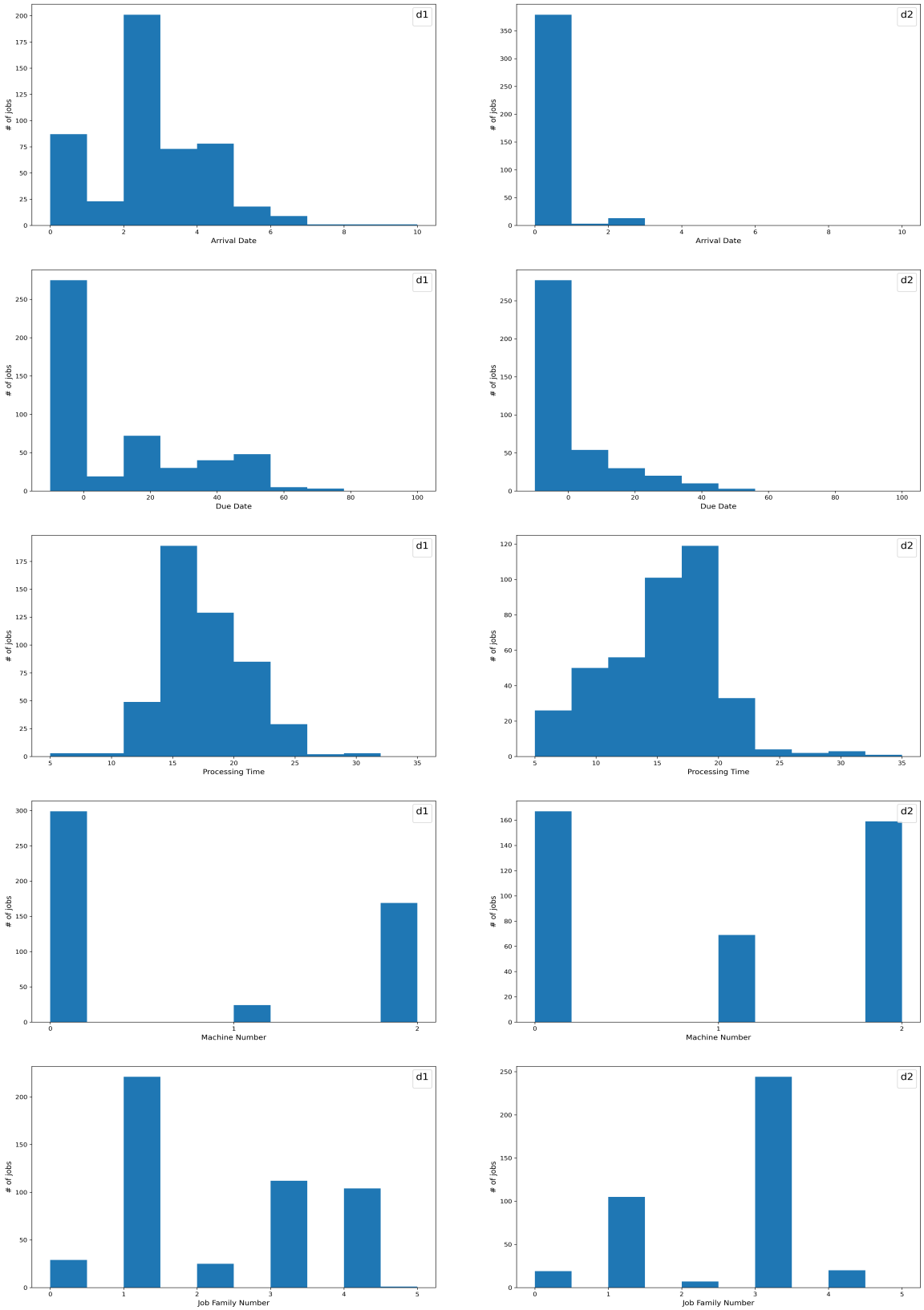

Figure 4.8: Characteristics of the two input datasets.

One of the main differences between the two datasets is the arrival date distribution. For d1, it does not have many jobs available to be sequenced at the early time (most of the jobs arrive later than 2 units of the arrival date). Recall that in LS-PMSP, one of the main objectives is to fill in the gaps in the early schedule. If not many jobs are available at the early time to be processed, we can expect to see some big gaps in the schedule created from the solution. On the other hand, d2 has most of the jobs available to be processed at the starting time of the schedule. Therefore, we expect to see that the schedule created from the solution for d2 would be packed in the early time. Examples of the schedules recovered from our solutions for d1 and d2 will be shown in the later subsection.

In terms of the due date distribution, both datasets have many jobs that are already due at the starting time of the scheduling horizon, so processing these jobs early will lead to a lower cost of a schedule. The distribution of the processing times of the jobs for each dataset also shows that some jobs take longer to process than another.

For the distribution of the machine number assignment for each dataset, we use machine number to indicate which machine a job can be assigned to. There are only 2 parallel machines to be considered in these datasets. If a job is labeled with machine number 0, it means the job is a flexible job that can be assigned to either machine 1 or 2. From the machine number distributions, d1 has about 62% of flexible jobs and d2 has about 42% flexible jobs, and d1 has fewer machine-1 jobs than d1.

In terms of the job family number distribution, we can see that both datasets have a wide spread of the jobs that belong to different job families. Recall that in LS-PMSP, job family number indicates a specific set of the processing requirements for a job, and processing jobs with different job family numbers consecutively will lead to a specific setup cost and time. A good schedule returned by our algorithm in general should have a small number of transitions of the job family numbers in a sequence, which can be seen in the examples of the schedules shown in the later subsection.

4.3.3 Overview of the Experimental Results

As discussed in Section 4.3.1, three methods based on the PMS algorithm are tested to see if any of them leads to a better algorithm performance: unsorted A, sorted A, and sorted B. For each method, we generate 10 replicates for each of the 2 datasets. For each dataset, the 10 inputs from the replicates for unsorted A are different from the 10 inputs for sorted A, since the former uses a sorted random *waitlist* as the input and the latter uses an unsorted random *waitlist* as the input. On the contrary, each replicate in sorted A and sorted B starts with the same sorted random *waitlist* but is improved by either method A (intra-VND + inter-VND) or method B (inter-VND + intra-VND) in the improvement phase of the algorithm.

The plots of the algorithm performance for each replicate of each method in d1 are shown in Figure 4.9, and these plots for d2 are shown in Figure 4.10. In each figure, the left column of the plots represents the unsorted A, the middle column of the plots is for sorted A, and the right column of the plots is for sorted B. The black dot in a plot indicates the improvement made by GCH, the blue dots are improvements made by intra-VND, and the red dots are improvements made by inter-VND. For unsorted A or sorted A methods, we can see the blue dots occur before the red dots in a plot, meaning the algorithm applies intra-VND before inter-VND in the improvement phase. For sorted B method, the red dots occur before the blue dots in a plot since the algorithm applies inter-VND before intra-VND in the improvement phase.

Having generated these results, we can apply statistical analysis on the converged values and converged times to compare the performance between the three methods, which will be discussed in the following subsections.

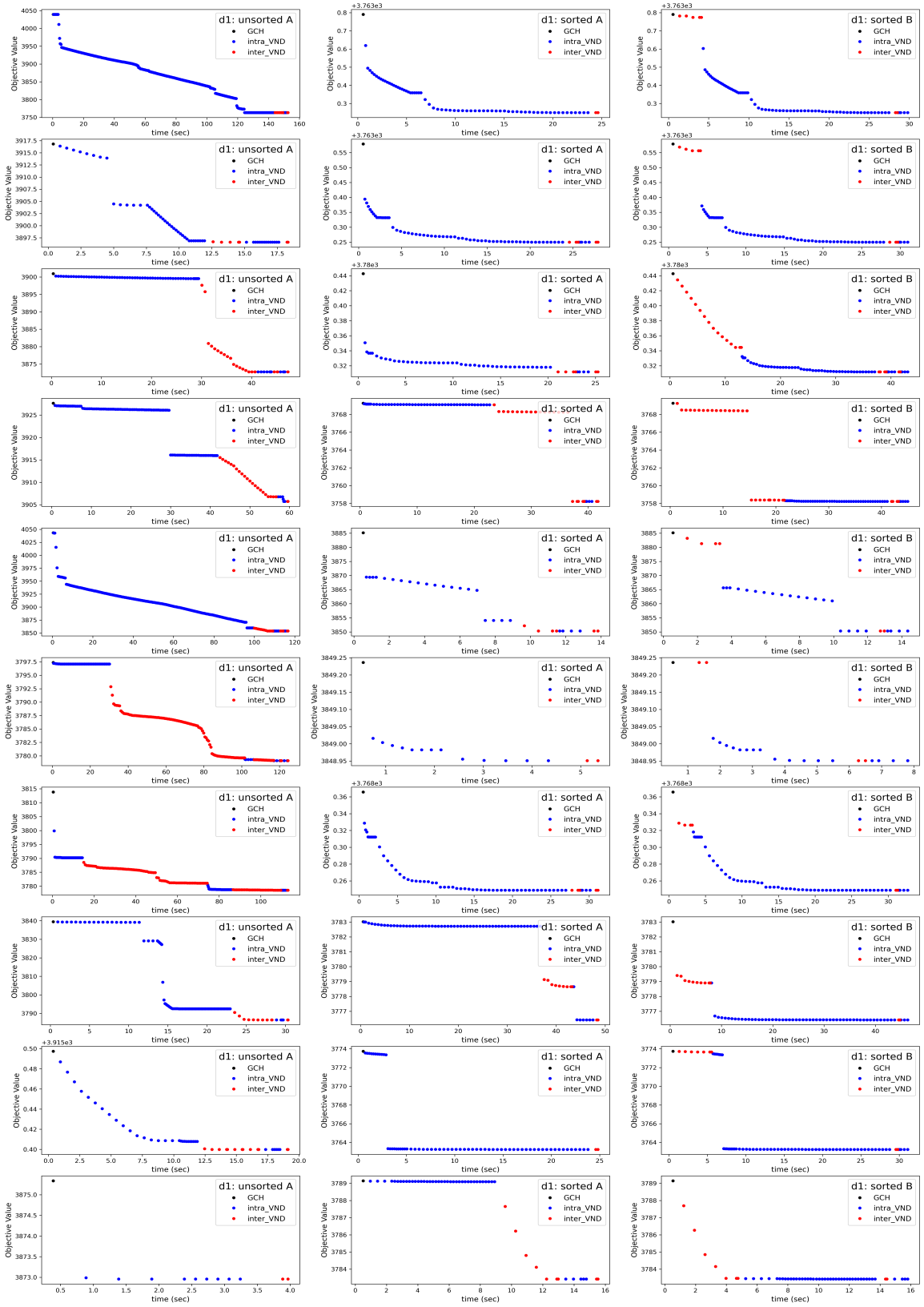

Figure 4.9: Algorithm performance for d1 unsorted A, d1 sorted A, and d1 sorted B.

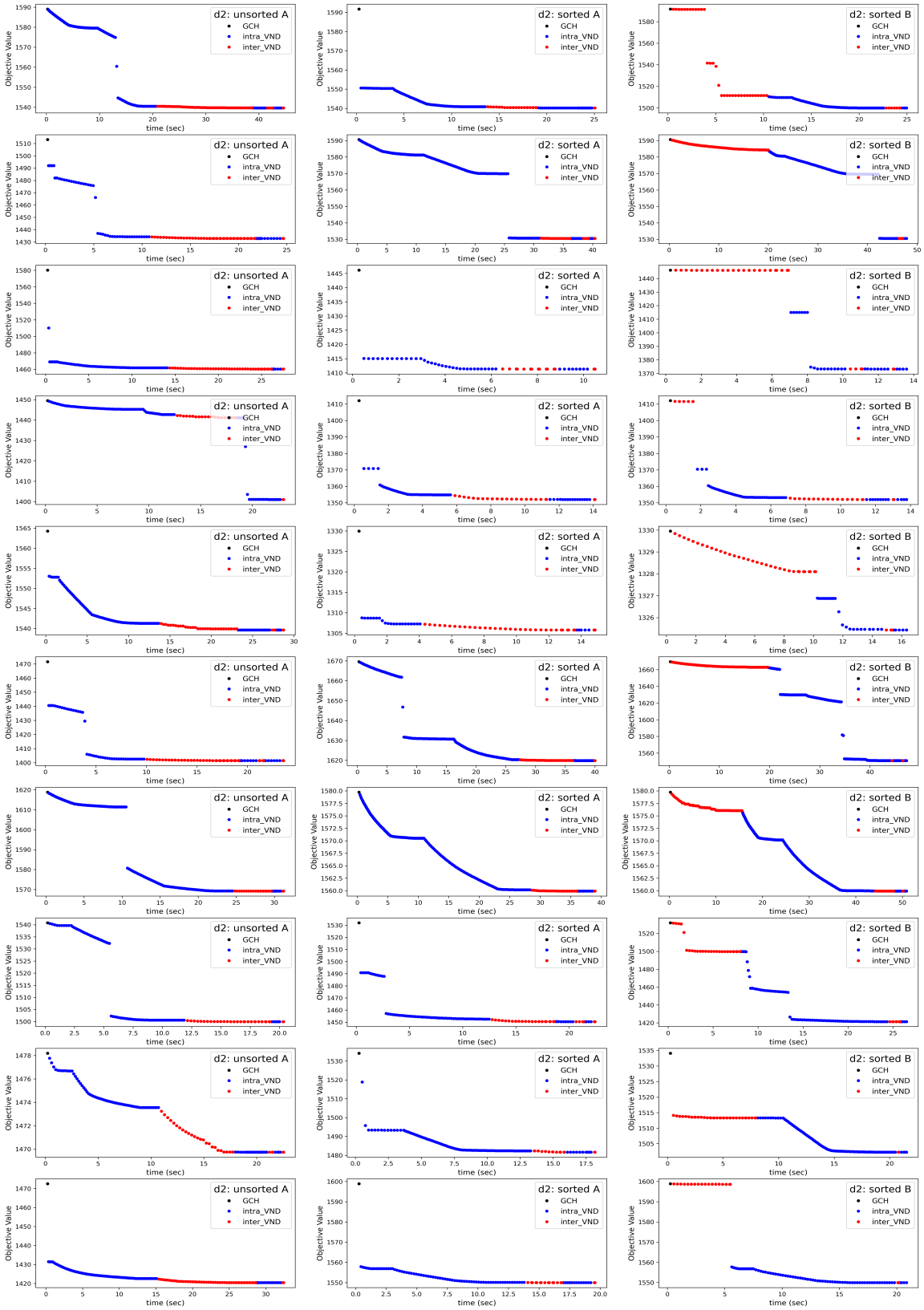

Figure 4.10: Algorithm performance for d2 unsorted A, d2 sorted A, and d2 sorted B.

4.3.4 Sorted vs. Unsorted Input Sequence

The performance comparison between the unsorted A and sorted A methods for each dataset can be visualized in the boxplots shown in Figure 4.11. In addition, the p-values for the two-tailed t-test that compares the two means of the methods for the two dependent variables, converged value and converged time, for each dataset are shown in Table 4.1. If a p-value is lower than 0.05, it means that we are 95% confident that the difference between the two means are statistical significant.

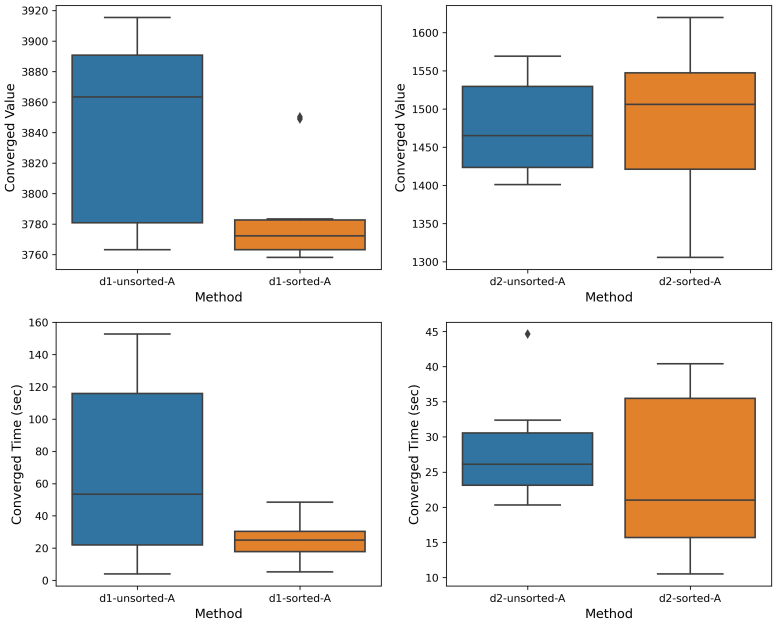

Figure 4.11: Boxplots for comparing converged values and times between unsorted A and sorted A.

Dependent Variable	d1 (p-value)	d2 (p-value)
Converged Value	0.017	0.86
Converged Time	0.024	0.43

Table 4.1: p-values of t-test on unsorted A and sorted A for d1 and d2.

From the left column of the boxplots in Figure 4.11, we can see that sorting *waitlist* helps the PMS algorithm converges faster with a better converged value than the unsorted method for dataset 1 on average, since a distribution of d1-sorted-A is significantly lower than a distribution of d1-unsorted-A. The interpretation from the boxplots is also supported by the p-values shown in Table 4.1, where both the p-values for converged value and time for d1 are less than 0.05.

For the d2 results, shown in the right column of the boxplots in Figure 4.11, the two distributions from the two methods are across each other for both the dependent variables. According to their p-values which are much greater than 0.05 as shown in Table 4.1, it suggests that there is no significant difference between the two means. Therefore, sorting *waitlist* does not improve the algorithm significantly for the second dataset.

However, we can see that between d2-unsorted-A and d2-sorted-A, the latter appears to have a lower minimum converged value than the former out of the 10 replicates, and both the methods have fast converged time (less than a minute) for solving LS-PMSP with 395 jobs. If we were to implement a multi-start algorithm where we repeatedly run the PMS algorithm with different inputs when one is converged until the time limit is reached, then the method that could generate a solution with the lowest cost than another within the time limit is best suited to solve the problem. That is, if sorting *waitlist* helps to find a solution with a lower cost in the long run, it is better to sort an input sequence than not sort it, even if the average solution qualities between the two methods might be the same.

Overall, we recommend a sorted *waitlist* in terms of the machine number of the jobs for solving our LS-PMSP based on our experiment on the two datasets. We now compare the sorted A method and sorted B method in the next subsection.

4.3.5 Intra-VND First or Inter-VND First

Figure 4.12 and Table 4.2 below show the boxplots for the comparison between the sorted A and sorted B methods and the associated p-values, following the same arrangement of a

figure and table in the previous subsection.

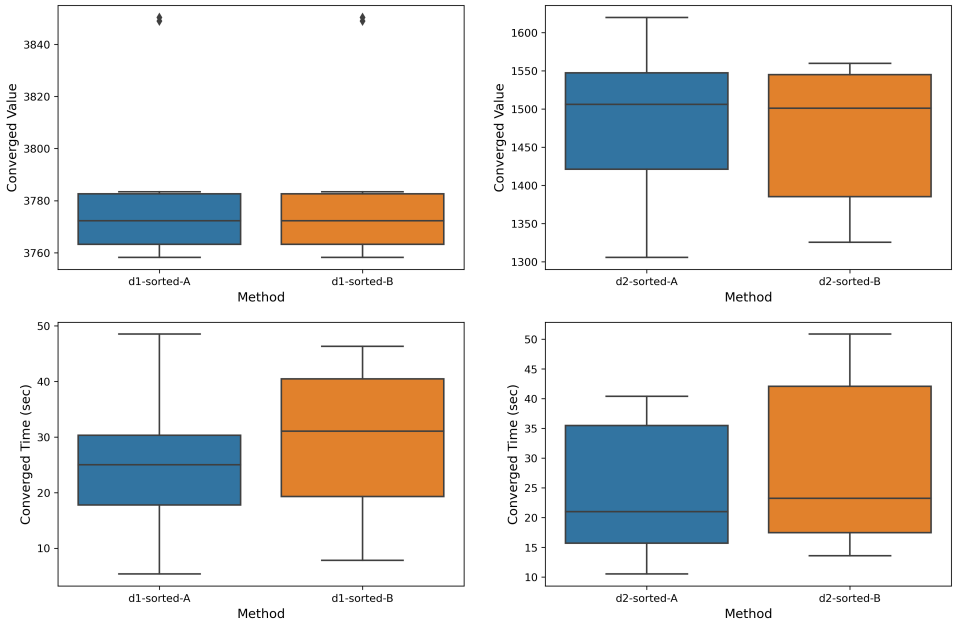

Figure 4.12: Boxplots for comparing converged values and times between sorted A and sorted B.

Dependent Variable	d1 (p-value)	d2 (p-value)
Converged Value	N/A	0.17
Converged Time	0.054	0.011

Table 4.2: p-values of paired t-test on sorted A and sorted B for d1 and d2.

There is no clear difference between the sorted A and sorted B methods in terms of converged values and times just by observing the boxplots shown in Figure 4.12. In fact, for dataset 1 d1-sorted-A and d1-sorted-B have the same distribution of the converged values. From the second and third columns of the plots in Figure 4.9, we can also see that for each replicate, although d1-sorted-A and d1-sorted-B apply intra-VND and inter-VND first respectively, they end up with the same converged solution within different converged times.

From Table 4.2, we notice that there is no need to apply paired t-test to converged value

of d1 since d1-sorted-A and d1-sorted-B have the same set of converged values, and the p-value for converged value for d2 is much higher than 0.05. This means that, for both datasets, the converged values between sorted-A and sorted-B methods are not significantly different. However, for the converged time d1 has a p-value 0.054 that is slightly higher than 0.05 and d2 has a p-value 0.011 that is less than 0.05. The paired t-test suggests that the two means of the converged times between the two methods are quite different. Based on the boxplots for the converged time shown in Figure 4.12, we can also notice that d1-sorted-A or d2-sorted-A appear to have a lower mean than d1-sorted-B or d2-sorted-B. Therefore, we can confirm that starting with intra-VND first instead of inter-VND first leads to a similar converged value as the other method but it converges faster on average. In addition, from the upper-right boxplot in 4.12, we see that d2-sorted-A appears to have a lower minimum converged value than d2-sorted-B based on the 10-replicate experiment. Following the same reason for the future usage of the multi-start algorithm mentioned in the previous subsection, sorted-A method is better than sorted-B method from this perspective.

Overall, we recommend to use sorted A method that sorts *waitlist* following the machine number of the jobs and applies intra-VND first then inter-VND in the improvement phase to solve our LS-PMSP.

4.3.6 Examples of the Algorithm Solutions

Given the job processing order for each machine contained in a solution returned by the PMS algorithm, we can recover a schedule for LS-PMSP following these orders and analyze the distributions of the job characteristics in a machine sequence such as the job family number and arrival date distributions. An example of the solution results for d1 obtained from one of the replicates of d1-sorted-A is shown in Figure 4.13 below.

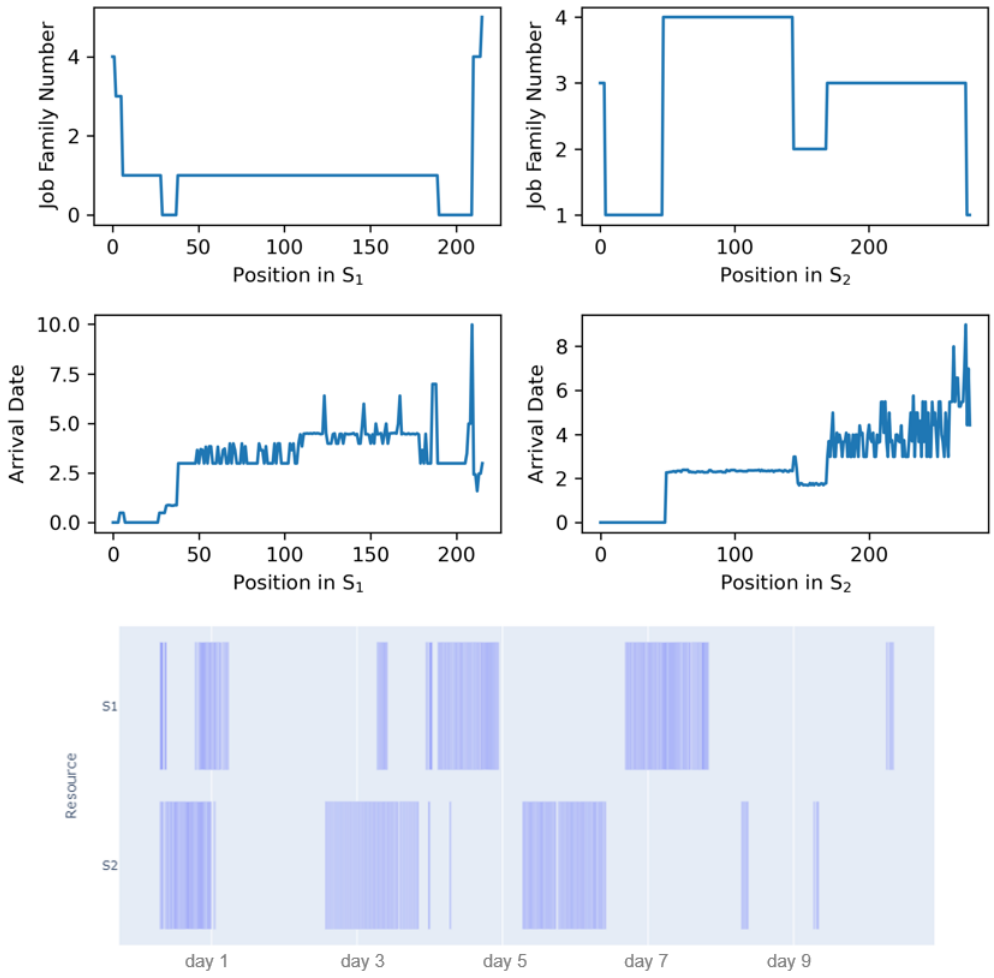

Figure 4.13: An example of the PMS solution results for d1.

From Figure 4.13, the first row of plots are for the job family number distributions in machine 1 and machine 2 following the processing orders of the jobs contained in a solution returned by the PMS algorithm. We can see that not many transitions of the job family number occur in S_1 and S_2 because a transition would cause an increase in the objective function value.

The second row of the plots in 4.13 is for the arrival date distribution of the jobs in a machine sequence. We can see that the algorithm schedules the jobs that arrive early first before sequencing the late-arrival jobs in a machine since the arrival date of a job is a hard

constraint in LS-PMSP and if a job arrives late, it simply cannot be used to fill in the gap in a schedule that is earlier than its arrival date. Recall that d1 does not have many jobs available to be processed at the early time of the scheduling horizon. This leads to some big gaps in the Gantt Chart as seen at the bottom of Figure 4.13.

For d2, an example of the solution results obtained from one of the replicates of d2-sorted-A is shown in Figure 4.14 below. We again see that the number of transitions of the job family number is small to ensure a low-cost schedule. The main difference from d1 where d2 has most of the jobs available at the early time of the scheduling horizon also leads to a Gantt Chart that looks much more packed and has fewer big gaps.

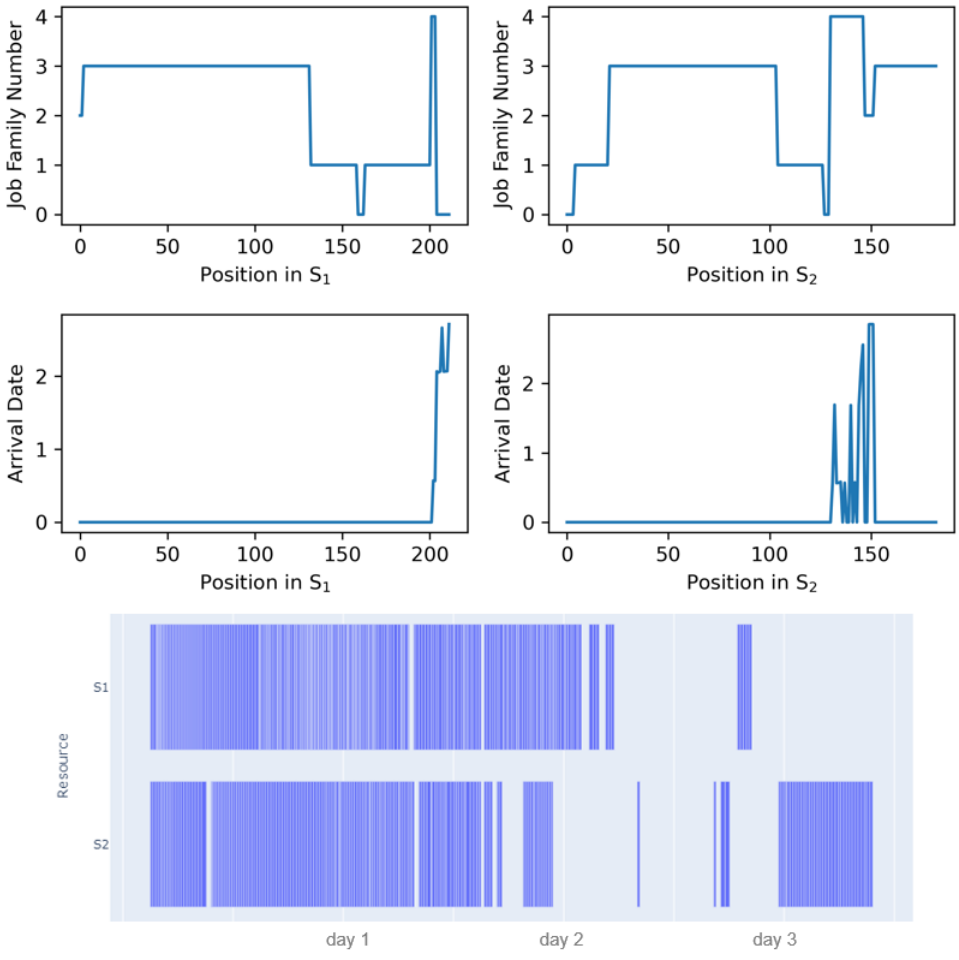

Figure 4.14: An example of the PMS solution results for d2.

Although both d1 and d2 solutions have big gaps in the later time of the schedule, this is not a major concern because rescheduling occurs at the critical time T in the current schedule, where a new batch of the jobs will come to the system to be sequenced with the remaining jobs from the previous batch. This ideally should lead to the current schedule always being packed with the jobs before T if most of the jobs considered in the current scheduling arrive before T, and therefore, minimizing the gaps before T in a schedule.

4.4 Chapter Summary

In this chapter, we introduce a practical large-scale parallel machine scheduling problem, abbreviated as LS-PMSP, faced by our industrial partner and propose an algorithm named PMS to solve such a problem. The objective of LS-PMSP is to minimize a combination of the gaps before a critical time T in a schedule, the total changeover cost of the jobs in each machine, and the total tardiness cost of the jobs in each machine. The PMS algorithm consists of two parts. It first applies a greedy constructive heuristic (GCH) algorithm to construct a good and feasible initial solution, then applies an iterative two-stage improvement process with intra-machine VND (intra-VND) and inter-machine VND (inter-VND). We also conducted experiments on the 2 actual datasets provided by our industrial partner. Combined with the statistical analysis, we found out that a sorted random sequence that follows the machine number of the jobs as an input to the PMS algorithm, and starting intra-VND instead of inter-VND first in the improvement process, helps to generate a better solution. The PMS algorithm proposed in this chapter is a good starting point to solve LS-PMSP efficiently. There are multiple research directions that can be taken to potentially improve the solution procedure even further.

4.4.1 Future Work

A variety of research directions can be taken to address LS-PMSP are:

1. Adding more neighborhood structures in intra-VND and inter-VND to improve a solution further. From our current work, we only apply (inter-)insertion and (inter-)swap neighborhoods in inter-VND and intra-VND. However, more complex neighborhood structures can also be implemented in the PMS algorithm to explore more possible improvements.

2. A multi-start algorithm combined with multiprocessing to improve Algorithm PMS. We mentioned in Section 4.3.4 that since our PMS algorithm runs efficiently to solve LS-PMSP with about 500 jobs given an input. If a time limit allows, we can restart the algorithm with different inputs to acquire multiple outputs and return the best one within the time limit.

3. The performance of rescheduling every critical time T would be interesting to investigate.

4. Given an input dataset with a specific set of characteristics, machine learning techniques might be applied and recommend that a specific order of the neighborhood structures might address a type of dataset better.

5. A scheduling problem with more than 2 parallel machines to be considered would be interesting to be investigated with our PMS algorithm. It is designed in a way that it can address a parallel machine problem with more than 2 machines but we have only investigated a 2-machine problem so far.

6. One can compare our VND-based PMS algorithm with other metaheuristics, or even combine them as hybrid-metaheuristics if it solves the problem better. In addition, an exact method using mathematical programming could be developed to address the specific problem structure of LS-PMSP and compare its performance against that using metaheuristics.

Chapter 5

Conclusions and Recommendations

5.1 Conclusions

The main conclusions to be drawn from this thesis are as follows:

1. **Single Machine Scheduling** – An efficient metaheuristic algorithm that applies an iterative greedy constructive heuristic to form an initial solution and uses VND-based metaheuristics with a sliding-window decomposition feature to improve an initial solution further is developed to solve the real-world large-scale ($\geq$1000 jobs) single machine scheduling problem in steel industry efficiently. A MIQP model based on the unit-specific general precedence formulation is developed to solve a small-scale instance of the single machine problem and benchmark the metaheuristics performance. The results show that the MIQP model is not able to find an optimal solution in a reasonable time with only 30 jobs to be sequenced on the single machine, so it is not practical to apply it to solve the actual large-scale problem with more than 1000 jobs to be scheduled. In addition, our metaheuristic algorithm that takes less than 0.05 seconds to solve a 30-job problem is able to find a good converged solution with a solution quality that is 50.7% better on average than the solution obtained from MIQP that is set to run for two hours and still far away from convergence. Statistical tests such

as ANOVA are conducted, and the results show that our metaheuristic algorithm that utilizes VND with a sliding-window decomposition technique improves a solution more efficiently than original VND algorithm. Based on the statistical analysis, a good set of parameters to use in the metaheuristic algorithm to solve the large-scale single machine problem more efficiently is also recommended.

2. **Parallel Machine Scheduling** – A real-world large-scale ($\geq$ 350 jobs) parallel machine scheduling problem faced in a steel plant that involves sequence-dependent setup times/costs, due dates, and release dates of jobs is addressed in this research. The problem poses a unique but practical objective that minimizes the gaps between jobs before a critical time in a schedule, the total changeover cost of the jobs, and the total tardiness cost of the jobs. A metaheuristic algorithm extended from the one used to solve the large-scale single machine problem (Chapter 3) is developed to solve the parallel machine problem efficiently. The first part of the algorithm utilizes a greedy constructive heuristic to form a good feasible initial solution fast, and the second part of the algorithm applies VND with intra and inter-machine movements iteratively to improve an initial solution further. Statistical tests like the t-test are applied and show that sorting an input data based on the machine number of the jobs, and applying intra-machine movements ahead of inter-machine movements, help the algorithm to solve the parallel machine problem more efficiently.

5.2 Recommendations for Further Work

Further avenues to explore for solving the single or parallel machine scheduling are:

1. **Stochastic Search** – The metaheuristic algorithms we have developed so far are deterministic, that is, given the same initial solution it will return the same solution. However, randomness can be added to a metaheuristic such as a perturbation move applied during the search in hope that a solution can jump out of an local optimum to seek for a global optimum, which may improve an algorithm further.

2. **Hybrid Metaheuristics** – Our single-state metaheuristic algorithm that starts with one initial solution and returns one improved solution can be combined with other population-based metaheuristics such as genetic algorithm to form a hybrid metaheuristic that balances the intensification and diversification during the intelligent search.

3. **Parallelization of Metaheuristics** – Depending on the computing resources for actual implementation, different parallelization techniques can be applied to our metaheuristics to find a better solution within the time limit. For example, a multi-start algorithm that utilizes multiple cores to start with different initial solutions and returns the best of the best solutions from each core when the time limit is reached, can be implemented in a production system.

4. **Machine Learning and Reinforcement Learning** – Artificial intelligence techniques are suitable to be supplemented to metaheuristics in a variety of ways. For example, machine learning can be applied to recommend a specific set of parameters used in a metaheuristic based on the input dataset characteristics to generate a better solution following the customization. In addition, research on reinforcement learning is increasing, and it might be a promising method to solve a single or parallel machine scheduling problem effectively.

List of References

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, **246**(2), 345–378.

Avalos, O., Ángel Bello, F., and Alvarez, A. (2014). Efficient metaheuristic algorithm and re-formulations for the unrelated parallel machine scheduling problem with sequence and machine-dependent setup times. *The International Journal of Advanced Manufacturing Technology*, **76**, 1705–1718.

Bianchi, L., Dorigo, M., Gambardella, L. M., and Gutjahr, W. J. (2009). A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, **8**, 239–287.

Cóccola, M. E., Cafaro, V. G., Méndez, C. A., and Cafaro, D. C. (2014). Enhancing the general precedence approach for industrial scheduling problems with sequence-dependent issues. *Industrial & Engineering Chemistry Research*, **53**(44), 17092–17097.

Davidović, T. and Crainic, T. G. (2013). Parallelization strategies for variable neighborhood search. *Technical report CIRRELT-2013-47*, .

Graham, R., Lawler, E., Lenstra, J., and Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In Hammer, P., Johnson, E., and Korte, B. (Eds.), *Discrete Optimization II*, Vol. 5 of *Annals of Discrete Mathematics*, pp. 287–326. Elsevier.

Hansen, P., Mladenovic, N., and Perez, J. A. M. (2010). Variable neighbourhood search: methods and applications. *Annals of Operations Research*, **175**, 367–407.

Hansen, P., Mladenović, N., Todosijević, R., and Hanafi, S. (2017). Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, **5**(3), 423–454.

Harjunkoski, I. and Grossmann, I. E. (2001). A decomposition approach for the scheduling of a steel plant production. *Computers & Chemical Engineering*, **25**(11), 1647–1660.

Harjunkoski, I., Maravelias, C. T., Bongers, P., Castro, P. M., Engell, S., Grossmann, I. E., Hooker, J., Mendez, C., Sand, G., and Wassick, J. (2014). Scope for industrial applications of production scheduling models and solution methods. *Computers and Chemical Engineering*, **62**, 161–193.

Kirlik, G. and Oguz, C. (2012). A variable neighborhood search for minimizing total weighted tardiness with sequence dependent setup times on a single machine. *Computers & Operations Research*, **39**(7), 1506–1520.

Kopanos, G. M., Laínez, J. M., and Puigjaner, L. (2009). An efficient mixed-integer linear programming scheduling framework for addressing sequence-dependent setup issues in batch plants. *Industrial & Engineering Chemistry Research*, **48**(13), 6346–6357.

Kopanos, G. M., Mendez, C. A., and Puigjaner, L. (2010). MIP-based decomposition strategies for large-scale scheduling problems in multiproduct multistage batch plants: A benchmark scheduling problem of the pharmaceutical industry. *European Journal of Operational Research*, **207**, 644–655.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA. Association for Computing Machinery.

Lin, C.-W., Lin, Y.-K., and Hsieh, H.-T. (2013). Ant colony optimization for unrelated parallel machine scheduling. *The International Journal of Advanced Manufacturing Technology*, **67**(1-4), 35–45.

Maravelias, C. T. (2012). General framework and modeling approach classification for chemical production scheduling. *AIChE Journal*, **58**(6), 1812–1828.

Mendez, C. A., Henning, G. P., and Cerda, J. (2001). An MILP continuous-time approach to short-term scheduling of resource-constrained multistage flowshop batch facilities. *Computers and Chemical Engineering*, **25**, 701–711.

Mishra, P., Singh, U., Pandey, C., Mishra, P., and Pandey, G. (2019). Application of student's t-test, analysis of variance, and covariance. *Annals of Cardiac Anaesthesia*, **22**(4), 407–411.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, **24**(11), 1097–1100.

Moon, S. and Hrymak, A. N. (1999). Scheduling of the batch annealing process — deterministic case. *Computers & Chemical Engineering*, **23**(9), 1193–1208.

Méndez, C. A., Cerdá, J., Grossmann, I. E., Harjunkoski, I., and Fahl, M. (2006). State-of-the-art review of optimization methods for short-term scheduling of batch processes. *Computers & Chemical Engineering*, **30**(6), 913–946.

Okano, H., Davenport, A. J., Trumbo, M., Reddy, C., Yoda, K., and Amano, M. (2004). Finishing line scheduling in the steel industry. *IBM Journal of Research and Development*, **48**(5.6), 811–830.

Pinto, J. M. and Grossmann, I. E. (1995). A continuous time mixed integer linear programming model for short term scheduling of multistage batch plants. *Industrial & Engineering Chemistry Research*, **34**(9), 3037–3051.

Roslöf, J., Harjunkoski, I., Westerlund, T., and Isaksson, J. (2002). Solving a large-scale industrial scheduling problem using milp combined with a heuristic procedure. *European Journal of Operational Research*, **138**, 29–42.

Sioud, A., Gravel, M., and Gagné, C. (2012). A hybrid genetic algorithm for the single machine scheduling problem with sequence-dependent setup times. *Computers & Operations Research*, **39**(10), 2415–2424.

Talbi, E. (2009). *Metaheuristics: From Design to Implementation*. John Wiley.

Tang, L., Liu, J., Rong, A., and Yang, Z. (2001). A review of planning and scheduling systems and methods for integrated steel production. *European Journal of Operational Research*, **133**(1), 1–20.

Vallat, R. (2018). Pingouin: statistics in python. *The Journal of Open Source Software*, **3**(31), 1026.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. (2012). *Probability and statistics for engineers and scientists*. Pearson Prentice Hall.

Wang, X. and Tang, L. (2010). A hybrid metaheuristic for the prize-collecting single machine scheduling problem with sequence-dependent setup times. *Computers Operations Research*, **37**(9), 1624–1640.