Constraint Directed Scheduling

Author

Riahi, Vahid

Published

2019-07-24

Thesis Type

Thesis (PhD Doctorate)

School

Inst Integrated&IntelligentSys

DOI

10.25904/1912/1826

Constraint Directed Scheduling

Vahid Riahi

BSc, MSc

Institute for Integrated and Intelligent Systems

School of Information and Communication Technology

Griffith University

Submitted in fulfillment of the requirements of the degree of

Doctor of Philosophy

April 2019

Abstract

Scheduling is a decision-making process, which is employed to allocate resources to tasks in a given time. Scheduling problems are in general NP-hard. In order to solve scheduling problems, three common types of methods have been used: exact methods (e.g., branch & bound and dynamic programming), population based metaheuristics (e.g., genetic algorithm and ant colony optimisation), and local search (LS) algorithms (e.g., simulated annealing and iterated local search). Exact methods are not able to address the practical-sized problems effectively with regard to both CPU times and solution quality. LS algorithms have recently attracted much more attention because of their simplicity, being easy to implement, robustness, and high effectiveness. However, the available LS algorithms in the literature typically use a generic structure for specific problems. In other words, the biggest disadvantage of those methods is the lack of problem specific components into their algorithmic structures. To fill in this gap, in this thesis, we consider constraint-based local search (CBLS) algorithms to solve scheduling problems because of their effectiveness and also because they are not used much in the scheduling literature. The key difference of CBLS with other LS algorithms is in the use of the problem specific information in the search process. CBLS helps the search focus more on areas where efforts will bring more effect, and thus increase the scalability of the search. In other words, CBLS attempts to exploit the essence of the problem and, based on the specificities of the problem, defines the procedures that will guide the search towards better local optima. The effectiveness of our proposed CBLS techniques is shown throughout this thesis by solving several scheduling problems, such as flowshops with blocking constraints, aircraft operations, and customer order problems.

The first scheduling problem is permutation flowshop scheduling problem (PFSP). It is one of the most thoroughly studied scheduling problems. However, mixed blocking PFSP (MBPFSP) is a generalised and more realistic version of PFSP with real-life applications such as cider industry. MBPFSP is an important branch of 'zero capacity buffer' scheduling problems. The second scheduling problem is aircraft scheduling problem (ASP). ASP involves allocation of aircraft to runways for arrival and departure flights, minimising total delays. In this thesis, we focus on both single-runway and multiple-runway ASP cases. The third scheduling problem is customer order scheduling problem (COSP), which has many applications including the pharmaceutical industries and the paper industries.

All of the three above-mentioned scheduling problems are NP-hard. They have made significant progress in recent years. However, within practical time limits, existing algorithms still either find low quality solutions or struggle with practical-sized problems. In this thesis, we aim to advance their search by better exploiting the problem specific structural knowledge, extracted from the constraints and the objective functions. We run our experiments on a range of respective standard benchmark problem instances. Experimental results and comprehensive analyses show that our new algorithms significantly outperform respective state-of-the-art scheduling algorithms.

Statement of Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Vahid Riahi

List of Publications

The main contributions of this study published in either the reputed journals or conferences in the field.

Journal Papers

- Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. 'Constraint guided accelerated search for mixed blocking permutation flowshop scheduling.' Computers & Operations Research 102 (2019): 102-120.
- Vahid Riahi, MA Hakim Newton, MMA Polash, Kaile Su, and Abdul Sattar. 'Constraint guided search for aircraft sequencing.' Expert Systems with Applications 118 (2019): 440-458.
- Vahid Riahi, MA Hakim Newton, MMA Polash, and Abdul Sattar. 'Tailoring customer order scheduling search algorithms.' Computers & Operations Research 108 (2019): 155-165.
- MA Hakim Newton, Vahid Riahi, Kaile Su, and Abdul Sattar. 'Scheduling blocking flowshops with setup times via constraint guided and accelerated local search.' Computers & Operations Research. 109 (2019): 64-76.
- Vahid Riahi, Mostafa Khorramizadeh, MA Hakim Newton, and Abdul Sattar. 'Scatter search for mixed blocking flowshop scheduling.' Expert Systems with Applications 79 (2017): 20-32.

Conference Papers

- Vahid Riahi, MA Hakim Newton, and Abdul Sattar. 'Exploiting setup time constraints in local search for flowshop scheduling.' In Pacific Rim International Conference on Artificial Intelligence (PRICAI), 2019.
- Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. 'Local search for flowshops with setup times and blocking constraints.' In Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS). 2018.

- Vahid Riahi, MMA Polash, MA Hakim Newton, and Abdul Sattar. 'Mixed neighbourhood local search for customer order scheduling problem.' In Pacific Rim International Conference on Artificial Intelligence (PRICAI), pp. 296-309. Springer, Cham, 2018.

- Vahid Riahi, MA Hakim Newton, and Abdul Sattar. 'Customer order scheduling by scattered wolf packs.' In 7th International Conference on Metaheuristics and Nature Inspired Computing, Morroco, 2018.

- Vahid Riahi, MA Hakim Newton, and Abdul Sattar. 'Constraint-guided local search for single mixed-operation runway.' In Australasian Joint Conference on Artificial Intelligence, pp. 329-341. Springer, Cham, 2018.

Acknowledgement

This work has benefited greatly from the guidance and support of many people over the past few years. I would like to express my gratitude to everyone who contributed to it in one way or another. First of all, I would like to thank the Almighty for giving me the strength to complete this work. And also thanks to my family members especially my wife, for their unconditional support and love throughout my life.

I would like to thank my main supervisor Professor Abdul Sattar whose wonderful temperament and canny advice are always appreciated. His constructive discussions and brilliant suggestions helped a lot throughout my PhD candidature. A big thank from the bottom of my heart must go to my associate supervisor Dr. M.A. Hakim Newton. Whenever I faced a difficult challenge, I found him beside me. His continuous guidance and constant support always kept me in the right direction. I owe a great debt of gratitude to him for his supervision as well as greatly assisting in various other ways.

I am also thankful to my lab mates for creating such an wonderful working environment. I would like to acknowledge the generous financial assistance from Griffith University, without which this work would not have been possible. Special thanks also to the Research Computing Services of Griffith University for providing the computing infrastructure and IT support during my PhD.

Finally, I thank my friends in Brisbane for the fantastic time we have had, and my friends on the other side of the world for staying in touch despite the distance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is to advance scheduling problems using constraint based approaches. Scheduling is a decision-making process dealing with allocation of resources to tasks over given time periods in order to optimise one or more objective functions. Scheduling problems commonly involve finding values for a set of decision variables. These variable-value assignments are often restricted by a set of constraints. Scheduling problems are in general mostly NP-hard. However, it does not mean that it is impossible to be solved effectively. Generally, to address and solve these problems, scheduling search algorithms explore large part of the solution space by exploring it efficiently.

1.1 Motivation

Scheduling problems in general are mostly NP-hard [5]. Search-based algorithms typically use some convenient and generic techniques such as metaheuristics in order to solve scheduling problems. Within practical time limits, existing algorithms still either find low quality solutions or struggle with large-sized problems. In this thesis, we attempt to show that one key reason behind this is the typical way of using generic algorithms that usually lack problem specific structural knowledge. Based on definition given by [6], constraint-based search (CBS) transforms the constraints into an objective function and uses search algorithms to optimise the objective function. The effectiveness of these CBS techniques is shown throughout this thesis by solving several scheduling problems,

such as permutation flowshop scheduling problems (PFSP), aircraft scheduling problems (ASP) and customer order scheduling problems (COSP).

The first research work presented deals with solving mixed blocking permutation flowshop scheduling problem (MBPFSP) which is a generalised and realistic version of PFSP, a well-known machine scheduling problem. MBPFSP has important applications in real-life industries such as cider industry [7], treatment of industrial waste and the manufacture of metallic parts [8]. MBPFSP is an important branch of PFSP which has attracted much attention in recent years. The second research work studied here focuses on aircraft scheduling problem (ASP) which is involved allocation of aircraft to runways for landing and takeoff flights in order to minimise total flight delays. Because of its huge importance for all air transportation stakeholders such as airlines, airports, and government, ASP has made significant progress in recent years. Third research work presented in this thesis deals with customer order scheduling problem (COSP). This problem has many applications including the pharmaceutical industries and the paper industries. COSP focused on the customer satisfaction and short delivery times, and the pre eminence of make-to-order production environments [9].

All three above-mentioned problems have made some progress in recent years. Local search algorithms and evolutionary algorithms are the most common solving techniques. Local search [10] is known to be a powerful solving technique especially for the problems with a huge search space. Evolutionary algorithms [11] are population based algorithms that mimic biological processes . However, within practical time limits, existing algorithms still either find low quality solutions or struggle with large problems [9, 12–17]. In this thesis, we aim to advance those scheduling problems' search by better exploiting the problem specific structural knowledge. The constraints and the objective functions are used to obtain such problem specific knowledge. Such knowledge is exploited both in constructive search methods and in search algorithms. The motivation comes from the constraint optimisation paradigm in artificial intelligence, where instead of random decisions, constraint-guided more informed optimisation decisions are of particular interest.

In the next sections, we introduce the problems dealt with throughout the whole research, and we establish the boundaries of our work and present their main contributions.

1.2 Problems Addressed

Mixed Blocking Permutation Flowshop Scheduling Problem (MBPFSP) is a generalised and more realistic scenario of PFSP. On the contrary of PFSP where it is assumed there is an unlimited capacity buffers between each two consecutive machines, in MBPFSP, there is zero capacity buffer between consecutive machines. It has been shown that MBPFSP has several practical examples such as cider industry [7]. Different approaches have been proposed in order to solve MBPFSP. In particular, genetic algorithms [12, 18] and bee colony optimisation algorithms [16] have been applied.

Aircraft Scheduling Problem (ASP) is to find the best sequence and schedule times for arrival and departure flights and try to reduce their delays. This problem has attracted much attention in recent years. Because flight delays are a very important growing challenge that needs to be tackled with proper emphasis. For example, 58.93 million passengers travelled in year 2016 in Australia, which was 2.5% more compared to that in year 2015 [19]. According to federal aviation administration (FAA), the total cost of all US air transportation delays in 2007 was estimated to be $31.2 billion [20]. To deal with this problem, different search algorithms have been employed including simulated annealing (SA) [15, 21], iterated local search (ILS) [13], scatter search (SS) [22], and particle swarm optimisation (PSO) [23].

Customer Order Scheduling Problem (COSP) is a branch of machine scheduling with a number of non-identical machines in a parallel layout. In COSP, a customer order is composed of different products that have to be processed by a dedicated machine [24]. It has several real-life applications including car repair shops [25], pharmaceutical industry [26], paper industry [27], and manufacturing of semi finished lenses [24]. Several search algorithms such as tabu search (TS) [27], SA [28], and greedy search algorithm (GSA) [9] are proposed to solve this problem.

1.3 Overview of the Contributions

This thesis indicates that the problem specific structural knowledge can be employed to increase the effectiveness of the search algorithms of scheduling problems. We also aim to show that different techniques can cooperate in a single algorithm to yield high

quality results. Thus, the scope of this research is problem modelling, problem solving, and search algorithms using the problem specific information.

This research work deals mainly with problem solving, and every chapter reports top results in the literature for the various scheduling problems addressed. The research presented in this thesis contributes to the state-of-the-art in presenting high quality results for the permutation flowshop scheduling, aircraft scheduling, and customer order scheduling problems.

1.3.1 Mixed Blocking Permutation Flowshop Scheduling Problem

First of all, a new evolutionary algorithm based on SS algorithm is proposed. It includes four main steps; initial population, combination method, improvement method, and reference set update. As the main contributions, for the initial population and improvement method steps, we propose new constraint-based approaches. Also, we show that this problem can be appeared in the cider industry as a real-life example. The proposed SS algorithm has been published as a journal article [7].

Then, we propose a local search-based algorithm for this problem. First, we present an acceleration method to efficiently evaluate insertion-based neighbourhoods of MBPFSP. Then, for the local search algorithm, we propose a constraint-based intensification and diversification methods. The core idea is giving more priorities to jobs created higher blocking for other jobs which are probably the most problematic parts of the solution. The proposed algorithm has been published as a journal article [29].

Next, we study the cider industry in more detail, and propose a new PFSP variant considering both mixed blocking constraints as well as sequence-dependent setup times constraints. We show that the new problem belongs to NP-hard complexity class. We also propose a computational model for the problem. We then develop a constraint-guided local search algorithm uses a new intensifying restart technique along with variable neighbourhood descent and greedy selection. This new problem and method have been published as an article in a conference proceeding [30], and its extended version has been accepted as a journal paper [31].

1.3.2 Aircraft Scheduling Problem

The main goal of this problem is to find the best sequence and also schedule times for arrival and departure flights in order to reduce their delays. In general, it is assumed that a set of aircraft are supposed to be operated on an airport with a single or multiple runways. Because of having different characteristics of multiple-runway and single-runway ASP cases, we perform a study on each of them. For the former one, we propose several constructive heuristics considering both constraints and objective function. During the local search, we design operators taking the constraints into account, we select runways to generate revised schedules taking the objective function into account. The new algorithms have been published as a journal article [32].

For the single-runway case, we also propose a new constraint-based local search algorithm advancing the search by injecting the specific knowledge of the problem into its different phases. In the intensification phase, we propose a greedy approach that gives more priorities to aircraft that are more problematic and create more delays. In the diversification phase, we employ a bounded-diversification technique that controls the new position of each selected aircraft and does not allow them to move very far away from their current positions. This new method has been published as an article in a conference proceeding [33]

1.3.3 Customer Order Scheduling

For this problem, we propose several deterministic constructive heuristics as well as a constraint-based local search algorithm. In the proposed deterministic constructive search algorithm, we use processing times in various ways to obtain initial dispatching sequences that are later used in prioritising customer orders during search. We also augment the construction process with solution exploration.

In the proposed stochastic perturbative search, we intensify its diversification phase by prioritising rescheduling of customer orders that are affected badly. Our tailoring of the search in this case is to make informed decisions when the search has lost its direction. On the contrary to that, in the intensification phase, we then take diversifying measures and use multiple neighbourhood operators randomly so that the search does not get stuck very quickly.

This new problem and method have been published as two articles in two conference proceedings [34, 35], and their extended version has been accepted as a journal paper [36].

1.4 Outline of the Thesis

Following this introductory chapter, the rest of the thesis is organised in a number of related chapters.

- Chapter 2 reviews the key concepts of constraint-based local search algorithms, and evolutionary algorithms.
- Chapter 3 identifies and discusses the research challenges of the mixed blocking permutation flowshop scheduling problem.
- Chapter 4 presents the work corresponding to the aircraft scheduling problem.
- Chapter 5 presents the work related to the customer order scheduling problem.
- Chapter 6 presents a summary of this work, outlines some future directions to extend this work and concludes the thesis.

Chapter 2

Preliminaries

This chapter presents the preliminaries of some basic search approaches that have been used throughout this thesis. It describes different search techniques to handle scheduling problems. Also, an overview of constraint-based local search and evolutionary algorithms is given. We also briefly introduces the scheduling problems addressed in this thesis.

Before discussing about local search and evolutionary algorithms, it must be mentioned that there are scheduling problems that are basically easy. It means that they can be formulated as linear programs. In this case, they can be solved optimally using some existing effective algorithms. These effective algorithms usually called polynomial time algorithms. Dynamic programming and branch-and-bound methods are the examples of these algorithms. In fact, those easy scheduling problems even with a large sized instance can be solved in a relatively short time on a computer [72]. However, there are many scheduling problems such as those three problems addressed in this thesis that are intrinsically very hard. Those problems are known as NP-hard. In fact, there are not any simple rules or algorithms that obtain optimal solutions in a practical time limits. Note that, above-mentioned algorithms, dynamic programming, and branch-and-bound, can be used in order to obtain optimal solutions for NP-hard problems [72] but solving optimally of those problems may need a huge amount of time. However, in practice that amount of time is not always available. Therefore, one is usually prefer to have a good feasible solution that is not necessary optimal but it is not far from optimal. The algorithms obtained a good feasible solution (but not necessary an optimal solution) in a short time are known as incomplete algorithms. Heuristic and meta-heuristic algorithms

are belongs to the class of incomplete algorithms. In this chapter, we have discussed about meta-heuristic algorithms. Since heuristics have specific characteristics that are proposed exactly for one particular problem, they are hugely discussed on their respective chapters.

2.1 Local Search Algorithms

Local search (LS) is one of the most effective non-systematic approaches [37]. It involves generation of neighbouring solutions around the current solution and evaluation of the neighbouring solutions. From the current solution, local search then moves to one of the neighbouring solutions, preferably to the best one as per given criteria. The neighbouring solutions are often generated from the current solution by using a random or an exhaustive approach [9, 13, 21, 38–40]. The local search algorithms are incomplete which means that there is no guarantee that the optimal solution is found. Although it sacrifices quality guarantee, it is greatly useful for large-scale problems where good solutions must be found within a given restricted time.

However, constraint-based local search (CBLS) is an effective variant of the typical LS methods. It has been successfully applied for solving computationally hard problems from computing science, operations research and various application areas such as routing and scheduling problems, and travelling salesman problem [41]. The core idea of CBLS is using constraints to describe and guide local search [6]. The motivation comes from the constraint optimisation paradigm in artificial intelligence, where instead of random decisions, constraint-guided more informed optimisation decisions are of particular interest. In fact, in the constraint optimisation paradigm in artificial intelligence, neighbourhood generation and evaluation are considered to be part of an integrated process. So the evaluation functions i.e. the constraints and the objective functions are used more purposefully in the generation of the neighbourhoods. The current solution is analysed to identify the problematic parts of a solution in terms of the constraint violation or the objective value and neighbourhood solutions are generated to improve that part of a solution.

The CBLS methods include several components, but possibly diversification and intensification steps are two most important ones. The former ensures that CBLS algorithms

explore the promising areas more thoroughly to find better solutions in terms of objective values, while the latter ensures that CBLS algorithm is not limited to only a reduced number of solution areas by exploring the non-visited areas. Therefore, to have an effective CBLS algorithm, an appropriate balance between diversification and intensification steps is critical.

Although CBLS algorithms have several advantages such as easy-to-design and easy-to-implement, they have some disadvantages that their biggest disadvantage is possibly convergence toward local optimum. To deal with this problem, three well-known approaches have been introduced:

- **Iterating from different initial solutions (positions) in the search space:** In this case, algorithm iteratively move to a different position in the solution space in order to escape from local optimum. This process is also called perturbation. For example, this approach is used in iterated local search (ILS) which is explained in detail in Section 2.1.3.
- **Using different neighbourhoods:** In this case, algorithm consistently moves from a neighbourhood to another one. The main idea is that different neighbourhoods create different landscapes and consequently different local optima. The use of different neighbourhoods thus enables the algorithm to navigate through the search area, and to explore more regions around the current solution. Variable neighbourhood descent (VND) is an example of this technique which is explained later in Section 2.1.2.
- **Accepting non-improving neighbouring solutions:** In this method, unlike the typical LS methods that accept only improving solutions, the inferior solutions may be accepted either with some probabilities (e.g., simulated annealing) or using some predefined mechanisms (e.g., late acceptance hill climbing [42]) to increase the chance of escaping from a local optimum.

2.1.1 Simulated Annealing

Simulated annealing (SA) inspired from an analogy of physical annealing process of solids in the metallurgical industry [43]. In annealing, applying slow cooling to metals leads

to a low energy-state crystallisation, while fast cooling leads to poor crystallisation. Therefore, in annealing, a high temperature heat is first applied to metals and then slowly cooled down to the room temperature.

SA is a stochastic algorithm. It starts from an initial solution π. and then it goes through a loop. In the loop, SA generates a new neighbouring solution π' from the predefined neighbourhood of the current solution. Next, the objective value difference ΔE is calculated $\Delta E = f(\pi') - f(\pi)$. If $\Delta E \leq 0$, it means that the objective value of new generated solution is better than that of current solution (smaller in the case of minimisation) and the new solution is accepted and considered as the current solution. However, If $\Delta > 0$, it means that the objective value of new generated solution is not better than that of current solution. But π' can be accepted as the current solution with probability $exp(-\Delta E/T)$ where T is a parameter known as temperature. T starts from an initial temperature T_{max}, and then iteratively reduced. The procedure of SA is given in Algorithm 1.

Algorithm 1 Simulated Annealing

1: $\pi \leftarrow \pi^0$ ▷ generate the initial solution
2: $T \leftarrow$ Initial Temperature (T_{max})
3: **while** *termination criteria not satisfied* **do**
4: $\pi' \leftarrow$ Generate a random neighbour solution of π
5: $\Delta E \leftarrow f(\pi') - f(\pi)$
6: **if** $\Delta E \leq 0$ **then** ▷ if better than the current solution
7: $\pi \leftarrow \pi'$ ▷ update the current solution
8: **else**
9: with a probability $e^{\frac{-\Delta E}{T}}$, $\pi \leftarrow \pi'$
10: Update temperature T
11: **return** The global best solution found so far

2.1.2 Variable Neighbourhood Descent

Variable neighbourhood descent (VND) algorithm was formalised by [44]. Its general idea is using multiple neighbourhoods in a local search, and switching iteratively from one neighbourhood to another. VND contains a collection of defined neighbourhoods $N_l(l = 1, \cdots, l_{max})$. Then it uses one neighbourhood (e.g., N_l) to improve the current solution π. If it fails then it uses the following neighbourhood, N_{l+1}. Once an improving solution is found by any of the neighbourhoods, the process resets by returning to first defined neighbourhood N_1. This mechanism is efficient when different neighbourhoods

are used since a local optimum for a neighbourhood N_l is not possibly a local optimum for another neighbourhood N_{l+1}.

The procedure of VND algorithm is shown in Algorithm 2. The performance of VND algorithm has a direct relationship with the selection of neighbourhoods and also the order of neighbourhoods. Typically, neighbourhoods are increasingly ordered based on their complexity and size [37].

Algorithm 2 Variable Neighbourhood Descent

A set of neighbourhoods $N_l(l = 1, \cdots, l_{max})$
$\pi \leftarrow \pi^0$ ▷ generate the initial solution
$l \leftarrow 1$ ▷ to denote operator N_1 will be used
while $l \leq \mathcal{N}$ **do**
    $\pi' \leftarrow$ the best neighbouring solution of π from neighbourhood N_l
    if $f(\pi') < f(\pi)$ **then**
        $\pi \leftarrow \pi'$; $l \leftarrow 1$
    else
        $l \leftarrow l + 1$
return The global best solution found so far

2.1.3 Iterated Local Search

The iterated local search (ILS) is a stochastic algorithm [45] proposed for solving optimisation problems. Its core idea is to perform a randomised walk in the space of local optimum and generate different initial solutions for intensification phase. ILS has four main steps: initialisation, intensification (local search), diversification (perturbation), and acceptance. The algorithm starts either with a randomly generated solution or a solution generated by a constructive approach. The initial solution is then improved by an intensification procedure. Later, it goes through a loop in which at each iteration, a new solution is generated by using a perturbation (diversification) method and the same intensification method is applied on the generated solution. The solution found in each iteration of the loop is either accepted or discarded using an acceptance criterion.

The procedure of ILS algorithm is given in Algorithm 3. In ILS algorithm, the typical diversification method consists of a number of random moves (called length) on the current solution (changing randomly the positions of some elements). So, the length of diversification is a key parameter. A small length leads to staying in a reduced area of search space and returning to previously visited solutions. While, with a large length,

algorithm may behave like a random restart method which usually gives low quality solutions.

Algorithm 3 Iterated Local Search

```
π ← Initialisation()
π ← LocalSearch(π)                                   ▷ Local Search
while termination criteria not met do
    π′ ← Perturbation(π)                             ▷ Perturbation of local optimum
    π″ ← LocalSearch(π′)                             ▷ Local Search
    π  ← AcceptCriterion(π″)   ▷ Decide if new solution replaces the current solution
return The global best solution found so far
```

As already mentioned, the diversification of ILS is typically performed by perturbing the current solution. However, there is a greedy variant of this procedure in which a number of elements, DS, is removed from the current solution and then, one by one, are placed in the best position of the partial solution. As mentioned by [46], this greedy diversification has been referred to by various names in the literature e.g. evolutionary heuristic [47], iterative flattening [48], ruin-and-recreate [49], iterative construction heuristic [50], and recently, iterated greedy algorithm [51].

2.2 Population-based Algorithms

The population-based algorithms can be seen as an iterative improvement in a population (group) of solutions [10]. All of population-based algorithms have several common concepts. All of them first initial from a population of solutions. Then, they generate a new population of solutions. Finally, they employ some selection procedures to decide about the new generated solutions, and also replacement procedures. They continue this process until given criteria are satisfied. This group of algorithm includes many algorithms such as genetic algorithm (GA) [52], scatter search (SS) [53], and grey wolf optimisation (GWO) [54].

In the population-based algorithms, there are some components which are needed to be carefully observed. The first one is the solution representation. It plays an important role in the effectiveness and efficiency of any algorithm. Because it has a direct effect on almost all other steps of algorithm. This step determines how algorithm components such as neighbourhoods, combination phases, and even evaluation function must be developed.

Evaluation function is one of the important and common phases of these algorithms. This function assigns a value to every solution in the population, and corresponds to the quality of that solution. The efficiency of evaluating the quality of a solution is different under different representations, and that is important because of the number of times this is called. The evaluation function is often referred to as fitness function.

Another important component is initial population. The population-based algorithms are more exploration search algorithms. So, the diversity of their initial solutions is critical. Otherwise, a premature convergence can happen. For this reason, these algorithms typically use a fully randomly generated population. In some of these algorithms such as SS, some diversification criteria are employed to guarantee the diversity of solutions in the population.

Another component is the stopping criteria. Each algorithm needs at least one stopping condition to stop the algorithm. Two common methods are static and adaptive procedures. The former one includes fixed number of iterations, a limit on CPU times etc,. The former one includes a fixed number of iterations with no improvement or the time of finding an optimum solution [10].

In the following parts, we review 4 population-based algorithms: genetic algorithms, scatter search algorithms, path relinking algorithms, and grey wolf optimisations. We have discussed genetic algorithms and that are either well-known or base of our thesis.

2.2.1 Genetic Algorithms

The genetic algorithm (GA) belongs to the evolutionary algorithms (EAs). Evolutionary algorithms are the algorithms that are based on the evolution of the species; based on Darwinian evolution theory [55]. They are considered as a component of evolutionary computation in artificial intelligence. EAs have a similar basic ideas, although their implementation mechanisms are different. EAs search for optimal solutions by evolving a multi-set of candidate solutions in which the least fit members of the population are eliminated, whereas the fit members are allowed to survive and continue until better solutions are determined [55].

GAs are stochastic algorithms that are successfully applied on many difficult optimisation problems. They are based on notion of competition [10] that mimics the evaluation

of species. They start from a population of solutions. Then the objective value of all solutions are calculated. Then, with a given selection mechanism, solutions are selected which will be combined to produce new solutions (offspring solutions). To do that, different operators such as crossover and mutation are used [56]. The crossover operator is a binary operator that its main role is to provide combination of solutions and convergence in a subspace e.g., by producing new solutions (offsprings) which share the characteristics of parents [56]. However, mutation is an unary operators applying on only one solution (parent). This operator represents small changes of parents. It is also considered as a mechanism for escaping from a local optimum and increasing the diversity of the population. Finally, a replacement procedure is applied to choose the solutions of new population among the parents (old solutions) and offspring solutions (new produced solutions). This process is continued until a stopping condition is met. The outline of GA is shown in Algorithm 4.

Algorithm 4 Genetic Algorithm

```
1: Generate(P(0))                                   ▷ Generate initial population
2: t ← 0
3: while termination criteria not satisfied do
4:     Evaluate (P(t))
5:     P'(t) ← Selection (P(t))
6:     P''(t) ← Reproduction(P'(t))
7:     Evaluate (P''(t))
8:     P(t + 1) ← Replace(P(t),P''(t))
9:     t ← t + 1
10: return The global best solution found so far
```

The selection step is one of the major GA search elements. The typical procedure is to give higher priority to solutions with better objective values to produce a high quality offspring. In other words, if f_i is the fitness of solution i in the population, its probability of being selected is $p_i = f_i/(\sum_{j=1}^{PopSize} f_j)$. This method is called Roulette-wheel Selection [57]. Although this simple method can lead to producing good solutions in the population, it can lead to losing the diversity. Therefore, the worse solutions in the population should have chance to be picked. Another selection method is K-way Tournament Selection [58]. In this method, first a number of solutions k are fully randomly selected from the population and then, the two with the best objectives amongst those selected solutions are picked. If the tournament size k is large, low quality solutions have a smaller chance to be selected since they have to compete with high quality solutions. So, k demonstrates the rate of convergence of the GA [58].

2.2.2 Ant Colony Optimisation Algorithms

The ant colony optimisation (ACO) algorithm belongs to the swarm intelligence that are inspired from the collective behaviour of species such as ants, wolfs, sharks, and monkeys [10]. The basic idea of ACO is given from operative behaviour of real ants to perform complex tasks such as transportation of food and finding shortest paths to the food sources [59]. ACO is considered as a multi-agent systems that each ant is considered as a single agent. This technique is widely used for solving computational problems which can be reduced to finding good paths through graphs [10]. In fact, ants deposit some chemical trails called pheromone on the ground in order to mark some favourable path that should be followed by other ants of the colony [59]. ACO algorithms uses a similar mechanism for solving optimisation problems. Algorithm 5 presents the prototype of ACO algorithm.

Algorithm 5 Ant Colony Optimisation Algorithm

```
1: Set parameters, initialise pheromone trails
2: while termination criteria not satisfied do
3:     ConstructAntSolutions
4:     ApplyLocalSearch (optional)
5:     UpdatePheromones
6: return The global best solution found so far
```

ACO includes three main steps. First, a set of solutions are constructed. Those solutions are then improved by using a local search algorithm (it is optional), and finally ACO updates the pheromone trails.

After initialisation of parameters, the metaheuristic iterates over three phases: at each iteration, a number of solutions are constructed by the ants; these solutions are then improved through a local search (this step is optional), and finally the pheromone is updated. Among these three steps, *UpdatePheromones* is the most important step. The aim of that step is to increase the pheromone values associated with high-quality solutions, and to decrease those that are associated with low-quality ones. This step helps algorithm to memorise the characteristics of "good" generated solutions that will be helpful for other ants created their solutions in next iterations.

2.2.3 Scatter Search Algorithms

Scatter search (SS) algorithm is an evolutionary method that was first introduced as a metaheuristic for integer programming [60]. There are similarities between SS and GA such as combination method in SS (called crossover in GA). The similarities and differences are discussed in detail by [61]. SS starts from a population ensuring both quality and diversity. Next, on the contrary of GA, an improvement method (local search) is applied on population. Next, it creates two sets of solutions called reference set, that one of them only keeps high quality solutions while another only keeps diverse solutions. SS then intelligently combines the solutions with each other to incorporate both quality and diversity, and to reach better solutions. The combination step is same as the crossover phase of GA. In each iteration, SS applies certain improvement methods on the combined solutions and updates the reference set. This process is repeated by the algorithm until the stopping criteria are satisfied. Algorithm 6 presents the prototype of SS algorithm.

Algorithm 6 Scatter Search

1: Initialise a **population of** P solutions.
2: Apply the **improvement method** on each solution of the population.
3: Update the **reference set**.
4: **while** *termination criteria not satisfied* **do**
5: Use the **subset generation method** to generate new subsets.
6: Apply the **combination method** to new generated solutions.
7: Update the **reference set**.
8: **return** The global best solution found so far

The template is made up of five following phases:

- **Diversification generation method**: This method creates a set of initial solutions with certain diversity. In fact, different procedures are used to ensure the diversity of solutions in the initial population.
- **Improvement method**: This method enhances the quality of solutions. In fact, this method is related to the intensification phase of local search algorithms. Thus, the intensification techniques are mostly used for this method.
- **Reference set update method**: Reference set update is one of the main parts of an SS algorithm. This is because when solutions are closed to each other, SS cannot improve the best solution even using the most complex and efficient

combination and improvement methods. Solutions are included in the reference set (R) both for quality and diversity. The subset R_1 of quality solutions contains n_1 best solutions. On the other hand, the subset R_2 of diverse solutions contains n_2 solutions that are all diverse from the solutions in R_1.

- **Subset generation method**: This method is similar to the selection method of GAs. In this step, solutions are selected in different ways from subset R_1 and R_2 to create combined solutions. The typical method is to consider all $n_1 \times n_2$ pairs of solutions (p, q), where $p \in R_1$ and $q \in R_2$.
- **Solution combination method**: This method is similar to the crossover operator in GAs. In this step, solutions already selected in the subset generation method are combined in which the new solution share the properties of both selected solutions.

Among the above-mentioned components, the reference set update method and subset generation method are generic, while other three are problem specific.

2.2.4 Path Relinking Algorithms

Path relinking method [62] was originally designed in the tabu search to incorporate intensification and diversification strategies, and was later suggested as combination method for scatter search [63]. However, this algorithm can be applied to any search algorithm with the aim of generating a set of good solutions.

The main idea of path relinking is to create a route between two solutions and explore the neighbourhood of the route for better solutions. The route between two solutions creates a pool of solutions sharing common properties of two input solutions. This method can be viewed as an extension of the crossover operators of GAs or combination method of SS algorithm. Instead of directly generating only one new solution when combining two selected solutions, path relinking generates routes between the selected solutions in the neighbourhood space. The procedure of PR algorithm is shown in Algorithm 7.

In designing the PR algorithm, there are some issues which are needed to be addressed as follows:

Algorithm 7 Path Relinking

1: Let s be starting point and t target solution.
2: $x = s$.
3: **while** $distance(x, t) \neq 0$ **do**
4: Find the move m which decreases $distance(x \oplus m, t)$.
5: $x = x \oplus m$ ▷ Apply move m to the solution x
6: **return** Best solution found in route between s and t

- **Path direction:** let π_1 and π_2 are two selected solutions. The intermediate solutions created by PR method is not same when start from π_1 with that of π_2. So, different strategies are used to determine starting and target solutions. Typically, the better solution among π_1 and π_2 is selected as the starting point and the another is selected as the target one. Because, neighbourhood of starting solution usually is more explored which lead to finding solutions with higher quality. However, another option is creating two paths considering each solution as the starting point. While this method needs more computational time.

- **Path move:** the performance of PR method highly depends on the selected neighbourhoods to move from the starting solution to the target solution. Different moves will create different solutions even using same starting and target solutions. Therefore, usually, for a given problem, to ensure the quality of given PR method, it is better to do some experiments and try some of the effective neighbourhoods and then pick the one with highest efficiency.

- **Path solution selection:** after creating a collection of intermediate solutions, it must be decided that which of them should be selected as the output solution. Following [64], if the quality of output solution is more important, the typical method is selecting the best solution in terms of quality among the intermediate solutions. However, these solutions usually very close to one of the input solutions especially to the better one. Thus, since solutions close to input solutions (π_1 and π_2) can probably lead to the same local optimum [64], researchers try to pick the intermediate solutions by considering both the quality and diversity. Therefore, in this case, a distance measurement is used to ensure that the selected intermediate solution is the best solution among those that are far enough (diverse) from the two input solutions.

2.2.5 Grey Wolf Optimisation Algorithms

Grey wolf optimisation (GWO) algorithm is a recently emerged population-based meta-heuristic for continuous optimisation problems. GWO borrows its ideas from basic living and hunting behaviours of grey wolves [54]. Wolf packs exhibit a leadership hierarchy: Alpha (α), Beta (β), and Delta (δ) wolves are the three highest level leaders in the order, while the rest of the wolves in the pack are Omega wolves (ω). Lower level wolves always follow higher level wolves in performing activities that include hunting, sleeping, group protection, and territorial inspection. GWO starts with a random initial population of wolves. Each wolf represents a solution. In each iteration, the three best solutions are considered to be Alpha, Beta, and Delta wolves in the order and GWO computes each Omega solution combining itself with the Alpha, Beta, and Delta solutions. The procedure of GWO algorithm is shown in Algorithm 8.

Algorithm 8 Grey Wolf Optimisation

1: Initialise a population of solutions.
2: Three best solutions in the population are α, β, and δ respectively. Others are ω
3: **while** *termination criteria not satisfied* **do**
4: Update position of each solution ω with regards to α, β, and δ.
5: Calculate the objective value of all solutions ω.
6: Update α, β, and δ.
7: **return** The global best solution found so far

There are similarities between GWO and GA. Same as GA, GWO starts from an initial population, then creates new solutions from those available in the population, and then, updates three best solutions in population and continues this process. However, there are differences as well. For example, all current solutions in the population, except three best ones, are directly replaced with a new solution combining itself and three best solutions e.g., applying crossover on four solutions.

2.3 Scheduling Problems Addressed

In this section, we introduce the scheduling problems addressed in this thesis: mixed blocking permutation flowshop scheduling problem (MBPFSP), aircraft scheduling problem (ASP), and customer order scheduling problem (COSP). In this section, we just briefly review these scheduling problems, but their real-life applications, their literature review, and problem solving approaches are widely discussed on each separate chapter.

2.3.1 Mixed Blocking Permutation Flowshop Scheduling Problem

Permutation flowshop scheduling problem (PFSP) is a well-known machine scheduling problem. In PFSP, there are m machines and n jobs. Each job $1 \leq j \leq n$ has a given non-negative and deterministic processing time $P_{i,j}$ when it is processed by a given machine $1 \leq i \leq m$. The machines are in a given fixed order but the order of the jobs is to be determined. There is a buffer i of a given capacity between each two successive machines i and $i+1$. A job j after its processing is completed by machine i can stay in buffer i until machine $i+1$ is available. Moreover, a machine $i+1$ can process a job j only after machine i has processed it. Eventually each job j needs processing at each machine i. At any time, each machine can process at most one job and a job can be processed by at most one machine. When the processing of a job has started at a given machine, its processing must be completed without any interruption. Moreover, once an ordering of the jobs is determined, jobs are processed in the same order by all machines. The PFSP is to find a permutation of the jobs such that all jobs one by one in the permutation order will be processed by all the machines and a given objective function will be optimised. In this paper, we minimise the makespan, which is the completion time point of the last job at the last machine.

In a PFSP, the buffer capacity between each two successive machines is considered unlimited; which is referred to by Wb (Without blocking). In this case, machines are able to process jobs as soon as the jobs are available. However, in many real-life industries, capacity of an intermediate buffer might be zero; which essentially leads to a blocking constraint. Under a blocking constraint, a job must be blocked at the current machine if the next machine is not free. A PFSP with blocking constraints is called a BPFSP. Inspired by real-world situations, three different types of blocking constraint have been introduced in BPFSP. RSb (Release when Starting Blocking) is the classical blocking constraint: machine i remains blocked by job j and cannot start processing of job $j+1$ until machine $i+1$ starts processing job j. Another blocking constraint is RCb (Release when Completing Blocking): machine i cannot start processing an available job $j+1$ until job j is finished by machine $i+1$, and then it leaves machine $i+1$ and thus its processing is started by machine $i+2$. The third blocking constraint is RCb* (Release when Completing Blocking*), which is a variant of RCb with one key difference. In RCb*, to start processing job $j+1$ by machine i, job j is to be finished by machine

$i+1$ regardless of whether job j leaves machine $i+1$ or not. MBPFSP allows multiple types of blocking constraint in the same problem whereas in a BPFSP only one type of constraint is allowed. We will use B_i to denote the blocking constraint between machines i and $i+1$.

It has been shown that MBPFSP has several practical examples such as cider industry [7]. Different approaches have been proposed in order to solve MBPFSP e.g., genetic algorithms [12, 18]

2.3.2 Aircraft Scheduling Problem

Aircraft scheduling problem (ASP) is to find the best sequence and schedule times for arrival and departure flights and try to reduce their delays. Assume there are N aircraft $\{1, \ldots, N\}$ either arriving or departing and one runway to perform the operations on. At any time, the runway can be used by only one aircraft. To solve ASP, we have to determine the *operation time* OT_j of the aircraft j. There are two generic constraint categories: **hard** and **soft** constraints that are to be satisfied to produce a feasible schedule. The aim is to satisfy all the hard constraints and attempt to accommodate the soft constraints as much as possible in order to produce a high-quality schedule.

Hard constraints are *Time window* and *Safety separation time*. In the former, because of several factors such as fuel restriction, airspeed, and possible manoeuvres, the operation time of each aircraft must lie within a specified time window. This time window is bounded by the *desired operation time* DOT_j and *latest operation time* LOT_j i.e. $OT_j \in [DOT_j, LOT_j]$. In the latter, since each aircraft creates wake turbulence that the following aircraft need to avoid, a certain minimum separation time is required between any pair of aircraft. The separation times depend on the aircraft classes (heavy, large, and small) and the aircraft operation types (landing or takeoff).

Soft constraint is *Deviation from desired operation times*. For each aircraft j, DOT_j is its desired operation time; which means that operation at that time has no delays and extra fuel burn. However, because of the capacity limit of runways and the hard constraints mentioned, some flights cannot operate at their DOT_j. Therefore, the operation times of some flights are deferred from their desired times. If an aircraft j operates after its desired time DOT_j, it would be penalised by $(OT_j - DOT_j)$.

Objective function of ASP is to minimise total delay cost of the aircraft resulting from the deviation of their operation times from the respective desired times.

This problem has attracted much attention in recent years. Because flight delays are a very important growing challenge that needs to be tackled with proper emphasis. For example, 58.93 million passengers travelled in year 2016 in Australia, which was 2.5% more compared to that in year 2015 [19]. According to federal aviation administration (FAA), the total cost of all US air transportation delays in 2007 was estimated to be $31.2 billion [20]. To deal with this problem, different search algorithms have been employed including simulated annealing (SA) [15, 21], and scatter search (SS) [22].

2.3.3 Customer Order Scheduling

Customer order scheduling problem (COSP) is a branch of machine scheduling with a number of non-identical machines in a parallel layout. COSP has n different customer orders and m machines arranged in a parallel layout. Each customer order comprises m jobs (product types) in which each job can be processed on one dedicated machine without interruptions. Let p_{ij} denote the processing time of ith job of customer order j to be processed on machine i. A machine can at any time process at most one customer order. Moreover, a machine can start processing another customer order as soon as it completes one. The aim is to find a permutation of customer orders that given objective is optimised. It is proved that there is always an optimal solution where all machines process the orders in the same sequence [27]. So, a COSP solution is represented by a permutation π of the customer orders indicated the sequence of the orders are executed.

It has several real-life applications including car repair shops [25], pharmaceutical industry [26], paper industry [27], and manufacturing of semi finished lenses [24]. Several search algorithms such as SA [28], and greedy search algorithm (GSA) [9] are proposed to solve this problem.

2.4 Conclusions

In this chapter, we review the basic structure of different local search algorithms and population-based algorithms which are mostly used in this thesis to handle challenging

scheduling problems. The common and different components of algorithms are mentioned and compared together. In the following chapters, we will discuss how these techniques are applied to solve different scheduling problems e.g., permutation flowshop scheduling problem, aircraft scheduling problem and customer order scheduling problem.

Chapter 3

Blocking Flowshops

In this chapter, the permutation flowshop scheduling problems with mixed blocking constraints called MBPFSP are discussed. A real example for this problem is introduced. Also, an acceleration technique for makespan calculation of insertion based neighbourhoods for local search and evolutionary algorithms are proposed. Next, two search algorithms based on scatter search (SS) and constraint-guided local search (CGLS) are proposed to solve this problem efficiently. Also, a new realistic variant for MBPFSP is introduced that includes the sequence dependent setup times as another constraint. So, this chapter is separated by MBPFSP without and with sequence-dependent setup times (SDST) constraints, based on following publications.

- Mixed blocking flowshops (no SDSTs)
  - Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. 'Constraint guided accelerated search for mixed blocking permutation flowshop scheduling.' Computers & Operations Research 102 (2019): 102-120.
  - Vahid Riahi, Mostafa Khorramizadeh, MA Hakim Newton, and Abdul Sattar. 'Scatter search for mixed blocking flowshop scheduling.' Expert Systems with Applications 79 (2017): 20-32.
- Mixed blocking flowshops with SDSTs
  - MA Hakim Newton, Vahid Riahi, Kaile Su, and Abdul Sattar. 'Scheduling blocking flowshops with setup times via constraint guided and accelerated local search.' Computers & Operations Research. 109 (2019): 64-76.

– Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. 'Local search for flowshops with setup times and blocking constraints.' In Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS). 2018.

3.1 Mixed Blocking Flowshops

In this section, we discuss about mixed blocking permutation flowshop scheduling problem (MBPFSP) without any sequence-dependent setup times (SDST) constraints. We first formulate the problem and propose the methods in order to solve this problem.

3.1.1 MBPFSP preliminaries

There are m machines and n jobs in a PFSP, all available beforehand. Each job $1 \leq j \leq n$ has a given non-negative and deterministic processing time $P_{i,j}$ when it is processed by a given machine $1 \leq i \leq m$. The machines are in a given fixed order but the order of the jobs is to be determined. Without loss of generality, assume the machines are in the order $1, 2, \ldots, m$. There is a buffer i of a given capacity between each two successive machines i and $i+1$. A job j after its processing is completed by machine i can stay in buffer i until machine $i+1$ is available. Moreover, a machine $i+1$ can process a job j only after machine i has processed it. Eventually each job j needs processing at each machine i. At any time, each machine can process at most one job and a job can be processed by at most one machine. When the processing of a job has started at a given machine, its processing must be completed without any interruption. Moreover, once an ordering of the jobs is determined, jobs are processed in the same order by all machines. The PFSP is to find a permutation of the jobs such that all jobs one by one in the permutation order will be processed by all the machines and a given objective function will be optimised. In this paper, we minimise the makespan, which is the completion time point of the last job at the last machine.

Blocking Constraints in PFSP

In a PFSP, the buffer capacity between each two successive machines is considered unlimited; which is referred to by Wb (Without blocking). In this case, machines are able to process jobs as soon as the jobs are available. However, in many real-life industries, capacity of an intermediate buffer might be zero; which essentially leads to a blocking constraint. Under a blocking constraint, a job must be blocked at the current machine if the next machine is not free. A PFSP with blocking constraints is called a BPFSP.

Inspired by real-world situations, three different types of blocking constraint have been introduced in BPFSP. RSb (Release when Starting Blocking) is the classical blocking constraint: machine i remains blocked by job j and cannot start processing of job $j+1$ until machine $i+1$ starts processing job j. Another blocking constraint is RCb (Release when Completing Blocking): machine i cannot start processing an available job $j+1$ until job j is finished by machine $i+1$, and then it leaves machine $i+1$ and thus its processing is started by machine $i+2$. The third blocking constraint is RCb* (Release when Completing Blocking*), which is a variant of RCb with one key difference. In RCb*, to start processing job $j+1$ by machine i, job j is to be finished by machine $i+1$ regardless of whether job j leaves machine $i+1$ or not. MBPFSP allows multiple types of blocking constraint in the same problem whereas in a BPFSP only one type of constraint is allowed. We will use B_i to denote the blocking constraint between machines i and $i+1$.

Operators and Neighbourhood

We define the following two operators that take a permutation as input and produces a permutation as output. Let $[k]$ denotes the job in the kth position of a permutation π.

1. remove(π, k): Given a permutation π with n jobs, remove job $[k]$ to obtain a permutation π' with $n-1$ jobs.
2. insert(π, j, k): Given a permutation π with $n-1$ jobs, insert job j at position k to produce a permutation π' with n jobs.

Using the operators mentioned above, for a given permutation, we define the following two neighbourhoods, each containing a number of neighbours.

1. insert-neighbourhood(π, k): Given a permutation π with n jobs, remove job $j = [k]$ from π to get an intermediate permutation π'. Then insert job j separately at each position k' of π' to produce n permutations. Each permutation π'' of the n permutations has n jobs.

2. insert-neighbourhood(π): Get a collection of all permutations produced by applying insert-neighbourhood(π, k) for each k.

NEH-based Flowshop Heuristics

The original NEH algorithm [65] is the most used heuristic in the flowshop literature. The effectiveness of the NEH method in flowshop scheduling has been shown in the literature [66, 67]. The NEH method consists of two steps. At the first step, it arranges all n jobs in a non-increasing order of their total processing times $P_j = \sum_{i=1}^{m} P_{i,j}$ over all machines. Then, at the second step, it improves the initial sequence of n jobs, obtained from the first step, by a prefix-insertion procedure. The prefix-insertion procedure, in each iteration $1 \leq k \leq n$, considers first k jobs in the whole sequence to be its current permutation and any jobs after position k are considered unscheduled. The prefix-insertion procedure then removes the job at position k of the current permutation and reinserts the job into all positions $1 \leq k' \leq k$ to obtain new permutations. It then selects the best permutation obtained this way to be the part of the whole sequence for the next iteration. The time complexity of the prefix-insertion procedure is $\mathcal{O}(n^3 m)$.

There exist some NEH variants that outperform the original one. In one variant, it has been focused on the first step of NEH and NNEH, a new variant for MBPFSP, has been proposed [7]. NNEH benefits from a new priority rule that combines initial orderings of Raj [68] and NEH-WPT [69] heuristics. In another variant, it has been focused on the prefix insertion procedure of the NEH and proposed five methods [70] , referred to as FRB1-FRB5 for PFSP, not for MBPFSP. However, all FRB methods need much larger CPU times than what the original NEH does, and only FRB4 is one of the best methods, producing promising results but using only small additional CPU time. The idea of FRB4 is that when an unscheduled job is inserted at position k, p jobs before and p jobs after position k are also removed and then reinserted in all positions to get a better fitting of those p jobs. These techniques will be studied further in Section 3.1.5.

MBPFSP Makespan calculation

Let π be a permutation of the given n jobs and $[k]$ denotes the job in the kth position of π. Also, let $S_{i,[k]}$ and $C_{i,[k]}$ respectively denote the starting and completion time points of job $[k]$ at machine i. The makespan $C(\pi)$ for a given permutation π for a given MBPFSP can be calculated by (3.1)–(3.8). For convenience of computation, in (3.1), $S_{i,[k]} = 0$ and $C_{i,[k]} = 0$ whenever $i < 1$ or $i > m$ or $k < 1$ or $k > n$.

$$S_{i,[k]} = 0, C_{i,[k]} = 0 \qquad i < 1 \vee i > m \vee k < 1 \vee k > n \tag{3.1}$$

$$C_{i,[k]} = S_{i,[k]} + P_{i,[k]} \qquad k = 1, ..., n,\ i = 1, ..., m \tag{3.2}$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, S_{i+2,[k-1]}) \quad k = 1, ..., n,\ i \leq m-2,\ B_i = \mathsf{RCb} \tag{3.3}$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, C_{i+1,[k-1]}) \quad k = 1, ..., n,\ i \leq m-1,\ B_i = \mathsf{RCb*} \tag{3.4}$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, S_{i+1,[k-1]}) \quad k = 1, ..., n,\ i \leq m-1,\ B_i = \mathsf{RSb} \tag{3.5}$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, C_{i,[k-1]}) \qquad k = 1, ..., n,\ i \leq m-1,\ B_i = \mathsf{Wb} \tag{3.6}$$

$$S_{m,[k]} = \max(C_{m-1,[k]}, C_{m,[k-1]}) \quad k = 1, ..., n \tag{3.7}$$

$$C(\pi) = C_{m,[n]} \tag{3.8}$$

For a given schedule π, the computational complexity of calculating the makespan $C(\pi)$ is $\mathcal{O}(mn)$. Therefore, the objective of the MBPFSP with makespan criterion is to find a permutation π_* in the set of all permutations such that π_* has the smallest possible makespan over all π.

$$C(\pi_*) \leq C(\pi) \qquad \forall \pi \tag{3.9}$$

Lemma 3.1. *Computing $C(\pi)$ from scratch using (3.1)–(3.8) has a time complexity of $\mathcal{O}(nm)$ and a space complexity of $\mathcal{O}(nm)$ as well.*

Proof. Each equation needs $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space for each machine each job. Thus, for n jobs and m machines, computing the makespan needs $\mathcal{O}(nm)$ time and $\mathcal{O}(nm)$ space. □ □

3.1.2 Proposed Accelerated Makespan Computation

Insertion-based operators are the most frequently used neighbourhood operator in the flowshop scheduling literature. As per Lemma 3.1, computing $C(\pi)$ from scratch takes $O(mn)$ time. So the time complexities of calculating makespan from scratch for all permutations in insert-neighbourhood(π, k) and insert-neighbourhood(π) are $\mathcal{O}(mn^2)$ and $\mathcal{O}(mn^3)$ respectively. Thus using insert-neighbourhood(π, k) and insert-neighbourhood(π) are computationally very costly particularly for large-sized problems. Inspired by the acceleration method proposed for PFSP [71], in this paper, we develop an acceleration method to efficiently compute $C(\pi')$ for each π' produced by insert-neighbourhood(π, k) and insert-neighbourhood(π).

Forward and Backward Calculations

Computation of $C(\pi)$ by using (3.1)–(3.8) is performed by iterating over all the jobs at position k from 1 to n, and for each job $[k]$, iterating over machines i from 1 to m. As such each job is scheduled at each machine at the earliest possible time. We call this procedure *forward computation*. We then find the latest time point $T_{i,[k]}$ when each job $[k]$ could be scheduled at each machine i. For this, we have to compute the schedule from the last job $[n]$ to the first job [1], and for each job $[k]$ from the last machine m to the first machine 1. We call this procedure *backward computation*. For convenience, *for the backward calculation, we assume time axis to be in the opposite direction.*

$$T_{i,[k]} = 0, P_{i,[k]} = 0 \qquad i < 1 \vee i > m \vee k < 1 \vee k > n \tag{3.10}$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, T_{i-1,[k+1]}, (T_{i-2,[k+1]} \text{ when } B_{i-2} = \mathsf{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \mathsf{RSb} \tag{3.11}$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, T_{i-1,[k+1]} + P_{i,[k]}, (T_{i-2,[k+1]} \text{ when } B_{i-2} = \mathsf{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \mathsf{RCb*} \tag{3.12}$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, (T_{i-2,[k+1]} \text{ when } B_{i-2} = \mathsf{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \mathsf{RCb} \tag{3.13}$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]},$$
$$(T_{i-2,[k+1]} \text{ when } B_{i-2} = \mathsf{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \mathsf{Wb} \tag{3.14}$$

$$T_{1,[k]} = \max(T_{2,[k]}, T_{1,[k+1]}) + P_{1,[k]} \quad n \geq k \geq 1 \tag{3.15}$$

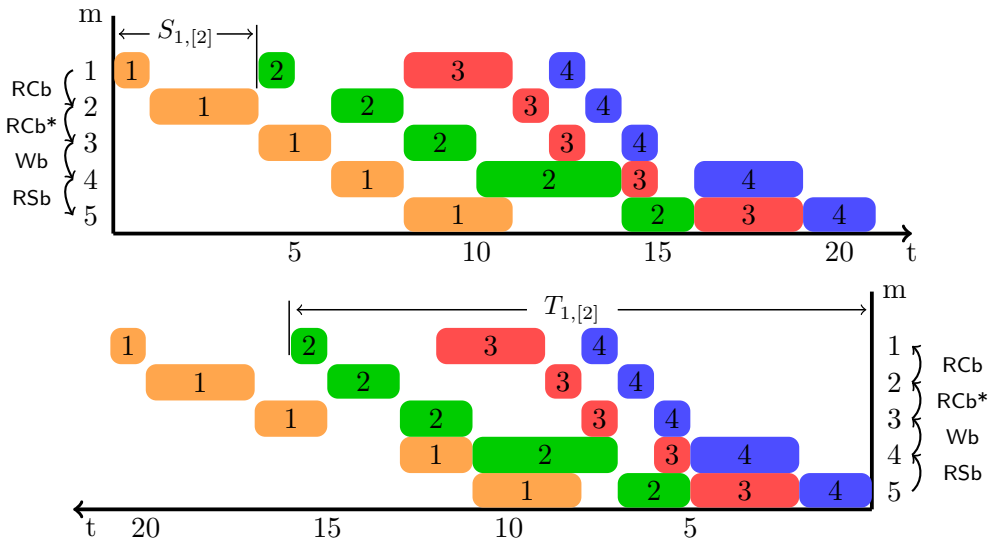


FIGURE 3.1: MBPFSP schedules by forward (top) and backward calculation (bttom)

Example MBPFSP schedules with 4 jobs and 5 machines are shown in Figure 3.1 to illustrate the forward (top) and the backward (bottom) calculations of the makespan. According to the reversibility property of flowshops [72], which also holds for MBPFSP, the makespan found by the forward calculation is the same as that found by the backward calculation. However, the starting and completion times for some jobs might not be same in the two calculations. For example, from $S_{1,[2]}$ and $T_{1,[2]}$, we see that the starting time points of job [2] at machine 1 are not the same in the two calculations. If a job is inserted at position 2 or earlier, the gap between $T_{1,[2]}$ and $S_{1,[2]}$ helps reduce the deferment of the latter jobs in the schedule.

One important issue with the MBPFSP is the order of the blocking types to be applied on successive machines. In [12], a repeated sequence of (*RCb, RSb, RCb*, Wb*) is used for ordering the blocking types of successive machines. This order is also used exactly by [16]. However, there is an important drawback of this blocking sequence and standard benchmark MBPFSP problems should take the drawback into account. The RSb constraint applied to a machine that is immediately after another machine running

under the RCb constraint cannot make any difference in the makespan. The following lemma proves this.

Lemma 3.2. *Applying RSb to machine $i+1$ does not create any blocking and has no impact on the makespan, if RCb is applied at machine i.*

Proof. When RCb is applied at machine i, we have $S_{i,[k]} \geq L_{i+1,[k-1]}$. Since $L_{i+1,[k-1]} = S_{i+2,[k-1]}$, we can write $S_{i,[k]} \geq S_{i+2,[k-1]}$. We also have $S_{i+1,[k]} > S_{i,[k]}$. So combining these relations, we obtain $S_{i+1,[k]} > S_{i+2,[k-1]}$, which always satisfies the condition $S_{i+1,[k]} \geq S_{i+2,[k-1]}$ for RSb at machine $i+1$. □

Computing Makespan Incrementally

The main idea behind the incremental computation is the use of the reversibility property of the MBPFSP. We not only use $S_{i,[k]}$ and $C_{i,[k]}$ from the forward calculation but we also use $T_{i,[k]}$ from the backward calculation.

Assume π is a permutation and $S_{i,[k]}$, $C_{i,[k]}$, and $T_{i,[k]}$ are given for each job $[k]$ and each machine i for permutation π. Assume $j = [k']$ in π and we have applied $\mathsf{remove}(\pi, k')$ to produce an intermediate permutation π', and then we also have applied $\mathsf{insert}(\pi', j, k'')$ to produce π''. We will now show how $S'_{i,[k]}$, $C'_{i,[k]}$, and $T'_{i,[k]}$ for permutation π', and $S''_{i,[k]}$, $C''_{i,[k]}$, $T''_{i,[k]}$, and hence $C(\pi'')$ for permutation π'' can be incrementally computed.

Incremental Computation of S', C', and T' for π': Using the two steps below, we get all $S'_{i,[k]}$, $C'_{i,[k]}$, $T'_{i,[k]}$ for permutation π'.

1. Notice that $S'_{i,[k]} = S_{i,[k]}$, $C'_{i,[k]} = C_{i,[k]}$ for each $k < k'$ and for each i. Compute $S'_{i,[k]}$ and $C'_{i,[k]}$ for $k \geq k'$ for each i using (3.1)–(3.8).
2. Similarly $T'_{i,[k]} = T_{i,[k+1]}$ for each $k \geq k'$ and for each i. Compute $T'_{i,[k]}$ for $k < k'$ for each i using (3.10)–(3.15).

Incremental Computation of S'', C'', and T'' for π'': In the steps below, for permutation π'', we get $S''_{i,[k]}$ and $C''_{i,[k]}$ for $k < k''$ and $T''_{i,[k]}$ for $k \geq k''$.

1. Moreover, $S''_{i,[k]} = S'_{i,[k]}$ and $C''_{i,[k]} = C'_{i,[k]}$ for each $k < k''$ and for each i. Compute $S''_{i,[k'']}$ and $C''_{i,[k'']}$ for each i using (3.1)–(3.8).

2. Similarly, $T''_{i,[k+1]} = T'_{i,[k]}$ for $k \geq k''$ and for each i.

Incremental Makespan Computation for π'': To compute $C(\pi'')$, we need two values: (i) the time point $L''_{i,[k'']}$ when machine i will be free after processing job $[k'']$ in $C(\pi'')$ and the job will leave the machine, and (ii) how much time is required to process the jobs after position k'' *i.e.* $T''_{i,[k'']}$ in this case. We compute $L''_{i,[k'']}$ in (3.16)–(3.19) below.

$$L''_{i,[k'']} = S''_{i+2,[k'']} \quad B_i = \mathsf{RCb} \tag{3.16}$$

$$L''_{i,[k'']} = S''_{i+1,[k'']} \quad B_i = \mathsf{RSb} \tag{3.17}$$

$$L''_{i,[k'']} = C''_{i+1,[k'']} \quad B_i = \mathsf{RCb*} \tag{3.18}$$

$$L''_{i,[k'']} = C''_{i,[k'']} \quad B_i = \mathsf{Wb} \tag{3.19}$$

Given all $L''_{i,[k'']}$ and $T''_{i,[k'']}$, in (3.20), we can compute $C(\pi'')$ from the time points when the machines complete processing all jobs.

$$C(\pi'') = \max_{i=1}^{m}(L''_{i,[k'']} + T''_{i,[k'']}) \tag{3.20}$$

An Illustrative Example: As an example, assume for a given π, after removing one job, we get a permutation π', which is shown in Figure 3.1. Permutation π' has 4 jobs and 5 machines. Now we insert job 5 at position 3 to get a permutation π'', which is shown in Figure 3.2 (top). Moreover, $L''_{i,[k]}$, $T''_{i,[k]}$ values for π'' in Figure 3.2 (bottom) demonstrates how the makespan is essentially calculated. The same examples shown graphically in Figure 3.1 and 3.2 are shown numerically below. The blocking constraints between successive machines are $\langle \mathsf{RCb}, \mathsf{RCb*}, \mathsf{Wb}, \mathsf{RSb} \rangle$. The processing times for each job at each machine are in the matrix $[P_{i,j}]_{5\times 5}$ below.

$$[P_{i,j}]_{5\times 5} = \begin{bmatrix} 1 & 1 & 3 & 1 & 2 \\ 3 & 2 & 1 & 1 & 2 \\ 2 & 2 & 1 & 1 & 1 \\ 2 & 4 & 1 & 3 & 3 \\ 3 & 2 & 3 & 2 & 1 \end{bmatrix} \qquad [S'_{i,[l]}]_{5\times 4} = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 6 & 11 & 13 \\ 4 & 8 & 12 & 14 \\ 6 & 10 & 14 & 16 \\ 8 & 14 & 16 & 19 \end{bmatrix}$$

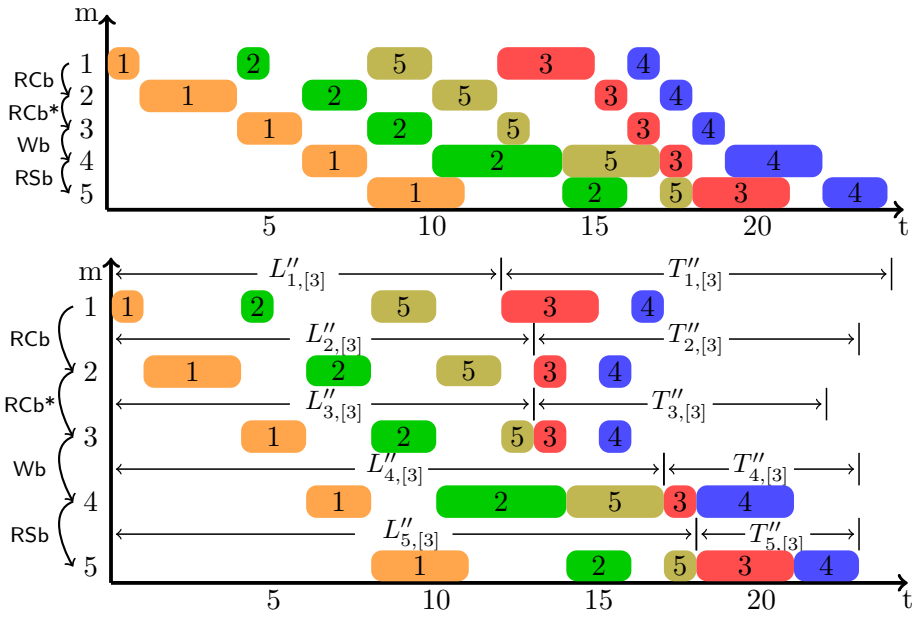


FIGURE 3.2: An example of MBPFSP schedule after inserting job 5 at position 3 (top) and an illustration of its incremental makespan calculation (bottom)

$$[C'_{i,[l]}]_{5\times4} = \begin{bmatrix} 1 & 5 & 11 & 13 \\ 4 & 8 & 12 & 14 \\ 6 & 10 & 13 & 15 \\ 8 & 14 & 15 & 19 \\ 11 & 16 & 19 & 21 \end{bmatrix} \qquad [T'_{i,[l]}]_{5\times4} = \begin{bmatrix} 21 & 16 & 12 & 8 \\ 20 & 15 & 9 & 7 \\ 17 & 13 & 8 & 6 \\ 13 & 11 & 6 & 5 \\ 11 & 7 & 5 & 2 \end{bmatrix}$$

Suppose the current solution is $\pi = \{1, 2, 3, 5, 4\}$. We insert job 5 at position 3, so $k' = 4$ and $k'' = 3$. First, job 5 is removed from π and $\pi' = \{1, 2, 3, 4\}$ is obtained. Then, using (3.1)–(3.8) of forward calculation and (3.10)–(3.15) of backward calculation, S', C', and T' are calculated as above. After inserting job 5 at position 3, permutation $\pi'' = \{1, 2, 5, 3, 4\}$ is obtained. Values of S'', C'' and next L'' are calculated as follows:

$$[S''_{i,[3]}]_{5\times1} = \begin{bmatrix} 8 \\ 10 \\ 12 \\ 14 \\ 17 \end{bmatrix} \quad [C''_{i,[3]}]_{5\times1} = \begin{bmatrix} 10 \\ 12 \\ 13 \\ 17 \\ 18 \end{bmatrix} \quad [L''_{i,[3]}]_{5\times1} = \begin{bmatrix} 12 \\ 14 \\ 14 \\ 17 \\ 18 \end{bmatrix} \quad [T''_{i,[3]}]_{5\times1} = \begin{bmatrix} 12 \\ 9 \\ 8 \\ 6 \\ 5 \end{bmatrix}$$

Finally, the makespan of permutation π'' is $C_{\max}(\pi'') = max(12+12, 14+9, 14+8, 17+6, 18+5) = 24$.

Incremental Evaluation of Insertion Neighbourhood

Given the incremental makespan computation, we develop an efficient evaluation method for the permutations in the insertion neighbourhoods. Note that for incremental computation of all permutations in insert-neighbourhood(π, k') where $j = [k']$ in π, we need to apply remove(π, k') only once to get π' and then we apply insert(π', j, k'') for each k''.

1. remove(π, k'): Remove job $j = [k']$ from the permutation π of n jobs and obtain a permutation π' of $n-1$ jobs by excluding job j.
2. Using incremental computation of S', C', and T' for π'
   (a) Calculate $S'_{i,[k]}$ and $C'_{i,[k]}$ for $i = 1, ..., m$ and $k = 1, ..., n-1$ for π'.
   (b) Calculate $T'_{i,[k]}$ for $i = m, ..., 1$ and $k = n-1, ..., 1$ for π'.
3. For $k'' = 1, ..., n$ do the following steps:
   (a) insert(π', j, k''): Insert job j at position k'' of π' to obtain π''.
   (b) Using incremental computation of S'', C'', and T'' for π''
      i. Calculate $S''_{i,[k'']}$ and $C''_{i,[k'']}$ for $i = 1, 2, ..., m$ for π''.
      ii. Calculate $T''_{i,[k'']}$ for $i = m, m-1, ..., 1$ for π''.
      iii. Calculate $L''_{i,[k'']}$ for $i = 1, 2, ..., m$ for π'';
      iv. Calculate $C(\pi'')$ for π'' using (3.20).

Given the above evaluation procedure for insert-neighbourhood(π, k'), for insert-neighbourhood(π), we follow the same procedure for each $1 \leq k' \leq n$.

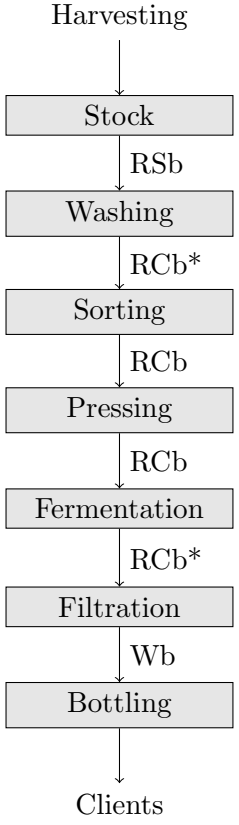
Lemma 3.3. *The above incremental method computes makespan of the permutations produced by* insert-neighbourhood(π, k') *and* insert-neighbourhood(π) *in* $\mathcal{O}(mn)$ *and* $\mathcal{O}(mn^2)$ *time respectively, both with space complexity of* $\mathcal{O}(n)$*. Without the incremental method, the complexities are* $\mathcal{O}(mn^2)$ *and* $\mathcal{O}(mn^3)$ *respectively.*

Proof. We directly obtain certain S, C, and T values of π' and π'' from those of π and π' respectively; so we need not compute those. In Step 2 above, we need $\mathcal{O}(mn)$ time to compute S, C, T values for π'. In Step 3b above, we need $\mathcal{O}(m)$ time to compute S, C, T, L and the makespan of π''. So in Step 3, the time complexity is $\mathcal{O}(mn)$.

Thus, the time complexities of evaluating all permutations in insert-neighbourhood(π, k') insert-neighbourhood(π) are $\mathcal{O}(mn)$ and $\mathcal{O}(mn^2)$ respectively. The space complexity of both methods is $\mathcal{O}(n)$. □

3.1.3 MBPFSP in Cider Production Process

Cider is a natural, liquid beverage that is obtained from the pressing of a finely ground fruit such as apples. A full-bodied cider requires the use of several different types of apples to give it a balanced flavour. This is because certain varieties of apples have flavour characteristics that work well together. There are four different types of apple juices: aromatic, astringent, acid-tart, and neutral tasting. Along with a few auxiliary steps, the cider making process typically involves three main stages including washing the fruit, pressing out the juice, and allowing it to ferment. Client orders determine a set of production jobs that must be processed in this system. The complete technological process [73] is shown in Figure 3.3:



1. Harvesting the apples from apple trees.
2. Stacking apples by transferring them to large storage bins.
3. Washing the apples to remove leaves, twigs, insects, spray residues, and harmful bacteria.
4. Sorting the apples based on their types.
5. Pressing the apples to get the juice from the pulp and collecting the juice in jars.
6. Using the fermentation process by adding certain amount of yeast to convert the apple sugars into ethyl alcohol and carbon dioxide.
7. Using the filtration process to clean up a cloudy cider and separate pulp from juice.
8. Bottling the cider and packaging to transport them to the clients.

FIGURE 3.3: Flowshop structure of the cider production process.

In the cider processing system, all jobs (client orders) have to pass through all seven stages. There are also defined job processing times for each stage. As can be seen, different types of blocking do appear in this industry. For an example, apples should stay at stock until washing machine is available (RSb constraint). Since, the apples should be washed carefully, all of them can not be poured to the tub simultaneously. In the pressing process, at first, apples are to be put in nylon cloth. Nylon is used because it is easy to clean and sturdy enough to withstand many pressings. So, the apples on the sorting machine, should stay on it until pressing is finished and pressed apples leave this stage (RCb constraint). Similarly, there is an RCb constraint between the pressing and the fermentation stages. To speed up fermentation, yeast nutrients such as ammonium sulfate and thiamine, or maybe extra sugar, honey or other sweeteners, are to be added. The amount of these yeast is predetermined from juice amount. So, when some amount of juice is transferred to the fermentation stage, next job should stay at pressing stage until juice leaves the fermentation stage (RCb constraint). Then, the juice should be filtered to remove pulps from juices. In this section, owing to the lack of capacity, juice should stay at fermentation stage until jobs at the filtration stage are finished. Assuming the same human operator could run both the fermentation and filtration stages, there could be RCb* constraint after the fermentation stage. There is no limitation among filtration and bottling stages (without blocking).

One important problem in the cider industry is thus to find a production schedule that allows minimising the makespan of completing all the jobs. Nevertheless, given this cider industry example, one can assume that mixed blocking could also appear in many other production processes.

3.1.4 Related Work

An effective iterated greedy algorithm (IGA) for PFSPs [51]. In each iteration of IGA, a number of jobs are randomly removed from the current solution and then are greedily reinserted again one by one. Recently, a hybrid algorithm based on a new evolutionary algorithm, backtracking search algorithm [74] is proposed for PFSPs.

The classical blocking constraint RSb (Release when Starting Blocking) [75] arises in many industries that include the chemical and pharmaceutical industry. Research on PFSP with RSb constraints is long. A local search algorithm called iterated greedy

algorithm (IGA) is proposed [76]. The main contribution of IGA algorithm is using a new local search based on swap neighbourhood operator. Another proposed algorithm is called three phase algorithm (TPA) which is based on simulated annealing (SA) algorithm [14]. The main contribution of TPA is proposing a new constructive heuristic based on NEH algorithm for initialisation. Another proposed method is a population-based algorithm based on an artificial bee colony (ABC) algorithm [77]. The main contribution of ABC algorithm was using another algorithm called differential evaluation (DE) in order to increase the diversity of solutions. Very recently, a local search algorithm called iterated greedy with iteration jumping (IG_IJ) is proposed [39]. This algorithm has two main contributions. The first main contribution was a new NEH based heuristic for initialisation. Another contribution was in the local search method. In fact, since there is an accelerated method for insertion move, most of the algorithms in the literature used that neighbourhood operator. However, they showed that using swap neighbourhood operator besides the insertion operator but with very small probability will increase the algorithm performance. IG_IJ produces the current state-of-the-art results.

Constraints RCb and RCb* have lesser background than constraints RSb and Wb. The RCb (Release when Completing blocking) constraint was introduced by [78]. RCb appears in some production environments such as waste treatment and aeronautics parts fabrication industries. Methods developed to handle RCb include an integer linear programming (ILP) model along with a meta-heuristic algorithm [79], and an electromagnetism-like (EM) algorithm [80, 81]. Recently, a new blocking model, RCb* (Release when Completing Blocking*), has been introduced [18]. RCb* is seen at cider factories [7].

In all above-mentioned studies, it is assumed that we have only one type of blocking constraint between all machines. However, real life production lines are mixed indicating presence of MBPFSPs. To deal with MBPFSPs, a new heuristic named TSS is proposed [12], which is competitive with the well-known NEH [65] heuristic. They also proposed a genetic algorithm (GA) and tested it on a set of generated benchmark with maximum 12 jobs and 100 machines. Later, a bee colony optimisation algorithm (BCO) is proposed [16] and showed it to be outperforming the GA [12] on the taillard benchmarks [1], which include medium and large-sized problem instances.

3.1.5 Proposed New Constructive Heuristic

In this thesis, we propose an NEH-based constructive heuristic named Improved NEH or INEH that uses elements from two other NEH-variants NNEH [7] and FRB4 [70]. We then further improve INEH by using a constraint-guided tie-breaking method.

Improving NEH Heuristic

In this paper, we combine ideas of NNEH and FRB4 respectively for the first and second steps of our Improved NEH denoted by INEH. In other words, INEH in the first step, produces an order of the jobs as the same as NNEH with a parameter value of $\alpha = 0.1$. INEH then improves the initial sequence using the prefix-insertion procedure of FRB4. Note that the higher the value of p, the more the jobs that get removed and reinserted. This produces better results but with a cost of more CPU time. In this work, we used $p = 1$, which means when a job at position k is reinserted in position k', jobs at position $k' - 1$ (if $k' > 1$) and $k' + 1$ (if $k' < k$) are also reinserted. Algorithm 9 presents the INEH algorithm.

Algorithm 9 INEH algorithm

1: *// Initial ordering by NNEH below*
2: **for** $j = 1$ to n with $\alpha = 0.1$ **do**
3: compute $A[j] = \alpha \times (\sum_{i=1}^{m}(m - i + 1) \times P_{ij}) + (1 - \alpha) \times (\sum_{i=1}^{m} P_{ij})$
4: Sort the jobs in non-decreasing order of $A[j]$ breaking ties in favor of the lowest j to get a sequence π.
5: *// Insertion procedure by FRB4 below*
6: **for** $k = 1$ to n with a given tie-breaking strategy **do**
7: Find the schedule π' with lowest $C_{m,[k]}$ by inserting job $[k]$ in all the possible positions $k' \leq k$ in π. Let $\bar{k}$ is k' with the lowest $C_{m,[k]}$.
8: Find the schedule π'' with lowest $C_{m,[k]}$ by inserting job $[k']$ of π' in all the possible positions $k'' \leq k$ in the sequence π' where $k' \in [\bar{k} - p, \bar{k} + p]$ and $1 \leq k' \leq k$ and $p = 1$.
9: **return** π''.

Lemma 3.4. *Given a constant p, the INEH algorithm shown in Algorithm 9 has a time complexity $\mathcal{O}(n^3m)$, which is reduced to $\mathcal{O}(n^2m)$ with the proposed acceleration technique. The space complexity of the INEH algorithm is $\mathcal{O}(n)$.*

Proof. Computing $A[j]$s needs $\mathcal{O}(nm)$ time and $\mathcal{O}(n)$ space. Then, sorting the $A[j]$s needs $\mathcal{O}(n \lg n)$ time and $\mathcal{O}(n)$ space. In the insertion procedure, in each iteration, $2p$

jobs are reinserted again in all possible positions in the partial solution. This produces $O(pn^2)$ partial solutions and each of these needs $O(nm)$ time to compute the makespan value from scratch using $O(n)$ space. Assuming a constant p, the INEH algorithm thus takes $O(n^3m)$ time (if the makespan is computed from scratch) and $\mathcal{O}(n)$ space. However, with the proposed acceleration method, the time complexity will be $\mathcal{O}(n^2m)$. □

Constraint-Guided Tie-Breaking

In the insertion phase of the INEH, at each iteration, several partial permutations may have the same objective. Obviously, each of those partial solutions leads to a different solution with different quality at the final step. As a result, a tie-breaking method can hugely affect the results of this heuristic [82, 83]. Different simple tie-breaking method in the literature are used: selecting the first tie [30, 84], or selecting random one [76]. However, as mentioned by [84] and [83], we can hardly find any problem specific tie-breaking for BPFSP. In this paper, we propose a constraint-guided tie-breaking strategy and the INEH version with this strategy is referred to as INEHtie. INEHtie uses the wastage time due to machines being idle or blocking as the tie breaker. This is because minimising the total wastage time in the following iterations could help minimise the overall makespan of the final sequence. To that end, when two partial solutions have the same makespan then the Total Wastage Time (TWT) of these two solutions are calculated by Equation 3.21, and the solution with the smallest TWT value is considered to be better than the other.

$$\mathrm{TWT}(\pi) = \sum_{i=1}^{m} L_{i,[n]} - \sum_{j=1}^{n} P_j \tag{3.21}$$

An Illustrative Example: We show an example of the tie breaking strategy.

TABLE 3.1: A MBPFSP instance with 3 jobs and 6 machines.

Job	Machine i						Processing
j	1	2	3	4	5	6	Time P_j
1	1	1	2	1	1	1	7
2	2	1	2	2	2	1	10
3	2	2	1	3	2	1	11

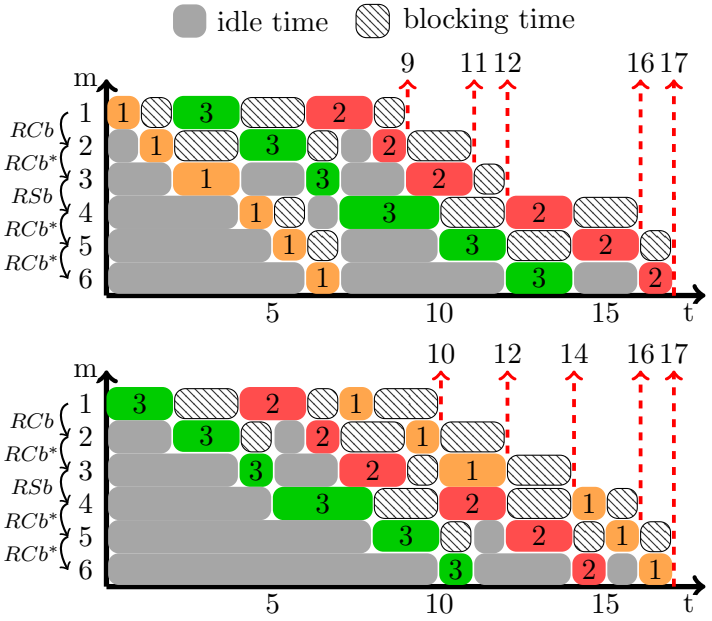


FIGURE 3.4: An example of the problem-dependent tie breaking

Consider the problem instance described in Table 3.1. The blocking constraints between the 6 machines are $\langle \mathsf{RCb}, \mathsf{RCb*}, \mathsf{RSb}, \mathsf{RCb*}, \mathsf{RCb*} \rangle$. Let $\pi_1 = \langle 1, 3, 2 \rangle$ and $\pi_2 = \langle 3, 2, 1 \rangle$ are two possible solutions of this problem instance. These two solutions are depicted in Figure 3.4.

As can be seen from Figure 3.4, the makespan of both solutions are 17 while their total wastage time is different. Considering 28 as the total processing time, the total wastage time of these two solutions are as follows:

- $\text{TWT}(\pi_1) = (17 + 17 + 16 + 12 + 11 + 9) - (7 + 10 + 11) = 54$
- $\text{TWT}(\pi_2) = (17 + 17 + 16 + 14 + 12 + 10) - (7 + 10 + 11) = 58$

Between π_1 and π_2, clearly, π_1 will be selected in this case.

3.1.6 Proposed Scatter Search

Our SS algorithm starts by generating an initial population of size P. Then, the improvement method is applied to improve the quality of the individuals in the population. In the next step, SS builds a reference set consisting of both high quality solutions and diverse solutions. After that, the subset generation method produces a subset of the reference

set as a basis for creating combined solutions. The solution combination method then generates new solutions and the improvement method enhances the solutions. Next, the reference set update is applied. The algorithm stops when the stopping criteria are satisfied. In the rest of the section, we describe the SS procedure that minimises makespan of an MBPFSP problem.

Initial Population

To generate an initial population with a certain level of quality and diversity, we use a new heuristic method named INEH, while the rest of the solutions in the initial population are created randomly.

Improvement Method

In our SS algorithm, we use an iterated local search (ILS) algorithm. The pseudo-code of an iterated local search method that we use is given in Algorithm 10.

Algorithm 10 Iterated local search algorithm

1: Let π be a solution from the initial population or is obtained from the combination method.
2: Apply NEH-based perturbation method to π to obtain a perturbed solution π'.
3: Apply insertR on π' repeatedly until no improvement found.
4: Apply swapR on π' repeatedly until no improvement found.
5: If the acceptance criteria hold then let $\pi = \pi'$.
6: Return π when the stopping criteria are satisfied else go to Step 2.

The perturbation algorithm to be used has a significant impact on the performance of ILS. In the literature, different methods have been used for perturbation algorithms. Examples include applying several random adjacent swap moves [85], inserting a random job to another random position [86], inserting two random jobs in two different random positions [38], and removing several jobs and then inserting them based on a constructive heuristic [51]. In this paper, we use an Insertion-based-NEH method, named IbNEH, as the perturbation phase. Let $\pi = (\pi_1, \ldots, \pi_n)$ be the current solution. At first, we select a random position $1 \leq p_1 \leq [\frac{n}{2}]$. Then, we apply the IbNEH heuristic algorithm on the vector $(\pi_{p_1}, \pi_{p_1+1}, \ldots, \pi_n)$. Note that the first insertion made by IbNEH is considered to be a forced one so that a new solution is always returned. Let $(\pi'_{p_1}, \ldots, \pi'_n) \neq (\pi_{p_1}, \ldots, \pi_n)$ be the best vector obtained from the application of

IbNEH algorithm and let $\pi' = (\pi'_1, \ldots, \pi'_{p_1-1}, \pi'_{p_1}, \ldots, \pi'_n) = (\pi_1, \ldots, \pi_{p_1-1}, \pi'_{p_1}, \ldots, \pi'_n)$. Next, we select a random position $[\frac{n}{2}] \leq p_2 \leq n$ and apply the IbNEH algorithm on $(\pi'_1, \ldots, \pi'_{p_2})$. Similarly, Let $(\pi''_1, \ldots, \pi''_{p_2}) \neq (\pi'_1, \ldots, \pi'_{p_2})$ be the best vector obtained from the application of the IbNEH algorithm. The output of the perturbation method is $\pi'' = (\pi''_1, \ldots, \pi''_{p_2}, \pi'_{p_2+1}, \ldots, \pi'_n)$. The current solution is always replaced by the solution returned by the perturbation method.

Let us illustrate the IbNEH perturbation procedure with an example (See Figure 3.5). Let permutation $\pi = (2, 6, 1, 5, 4, 3)$ be and $(3, 4)$ are two random positions. Based on the IbNEH perturbation that can be seen in Figure 3.5, $\pi = (5, 2, 6, 1, 4, 3)$ is finally selected as the solution for the following iterations. Note that the first change made from $(2, 6, 1, 5, 4, 3)$ to $(2, 6, 5, 1, 4, 3)$ is forced to ensure that a new different solution is always obtained.

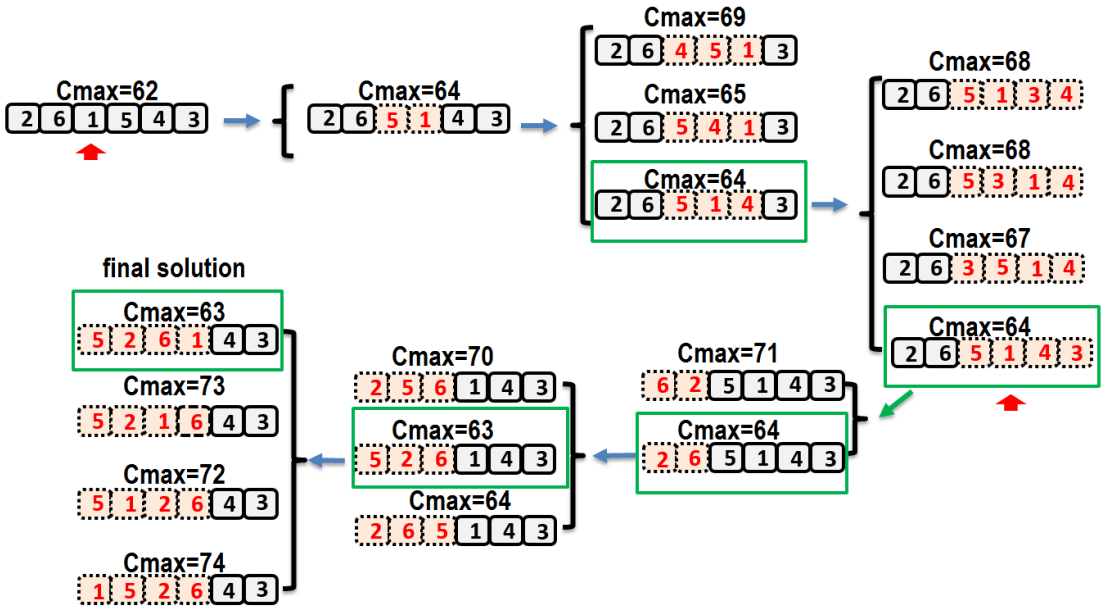


FIGURE 3.5: An example for NEH-based perturbation

The performance of a local search depends greatly on the selection of neighbourhood structures. Among several kinds of move in the literature, because of their effectiveness, we choose *insert* and *swap* moves. Insert move is already discussed. Based on swap move, two jobs from positions j and j' exchange their positions.

Instead of using insert-neighbourhood(π), in this paper, we simplify that insertion procedure to make it faster and more suitable for our local search, and denote it by InsertR() and use the same procedure for swap move and denote that as swapR(). In insertR(),

we do not repeat the removal and insertion procedure for each job. We rather select a job to be removed, consider all insertion positions, and if a better position is found then perform the removal and insertion. This procedure continues until for the first time for the selected job no better position is found in the sequence. Algorithm 11 shows the pseudo-code of insertR(). Because of similarity between insertR() and swapR(), the procedure of swapR() is not shown.

Algorithm 11 insertR() algorithm

1: **Input:** Sequence π.
2: Sort the jobs in a list DBT in decreasing order of their differential blocking time (it is explained later).
3: Select k=1 and the corresponding job DBT[k].
4: Remove the job from the sequence s.
5: Find the best permutation, π', by inserting job [k] in all other positions of π
6: If π' better than π, $\pi = \pi'$, $k = k + 1$ and go to Step 2.
7: **Output:** Sequence s.

One of the main issues in local search is the job selection method that can be done either randomly or greedily. In the literature, the random selection is more common [87]. In this paper, jobs are not chosen randomly but in the order given by a referenced permutation. It has been shown by [88] that RCb has created extra blocking time defined as *differential blocking time*. Two different differential blocking times are introduced as *differential blocking time "before"* and *differential blocking time "after"* that are shown in Figure 3.6. The differential blocking time "after" job $[k]$ on machine i will arise because of the insufficient execution time of job $[k-1]$ on machine $i+2$. In other words, when its completion time is earlier than the completion time of job $[k-1]$ on machine $i+2$. However, the differential blocking time "before" job $[k]$ on machine i is due to insufficient execution time of job $[k]$ on machine $i-1$ than the execution time of job $[k-1]$ on machine $i+1$. In fact, it will be created when its completion time of machine $i-1$ is less than the completion time of job $[k-1]$ on machine $i+1$. In this paper, we consider these differential blocking times for both RCb and RCb*. In fact, they will create the same blocking in some machines [12] because they have a very similar procedure. Therefore, after obtaining the *differential blocking time "before"* and *differential blocking time "after"* for each job, jobs are sorted on a decreasing order of the sum of these two *differential blocking time*. In fact, the jobs with more *differential blocking time* have more priorities. Figure 3.6 shows an example with 3 jobs and 6

machines with mixed blocking constraints. In this example, *differential blocking time "before"* and *differential blocking time "after"* are also shown.

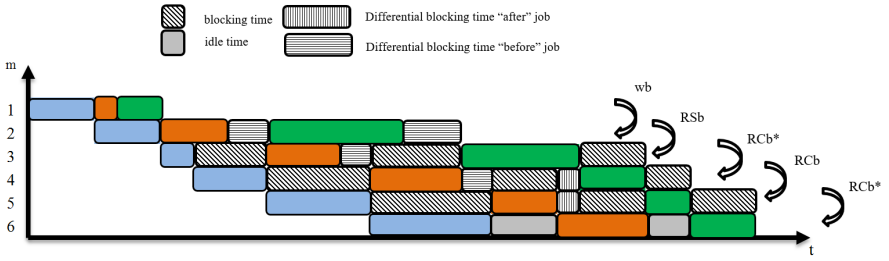


FIGURE 3.6: An example with 3 jobs and 6 machines with Mixed blocking constraints

Combination method

In this paper, path relinking [62] is used as a combination method. The main idea of path relinking is to create a route between two solutions and explore the neighbourhood of the route for better solutions. Path relinking method was originally designed in the tabu search to incorporate intensification and diversification strategies, and was later suggested as combination method for scatter search [63]. Path relinking has been successfully applied to the permutation flowshop scheduling problem. For example, the path relinking is used as the crossover method to generate offspring solutions [89]. The path relinking is used as a crossover operator as well [90]. In addition, this method is widely used as a solution combination method in scatter search algorithms [91–94].

The path relinking implementation used in this paper is based on the path relinking of [16]. To produce new solutions, they used a swap operator, but in this paper, we applied an insert operator. This is because in the literature, it is shown that insert moves are stronger than swap moves [71, 89]. Besides we also use a local search to create some other solutions. In fact, it is showed that applying local search in the path relinking improves its performance [53]. However, since there are similar solutions in each path, the use of local search for all solutions is not necessary. So, in this paper, after each 5th step of producing solutions (it is fixed with a preliminary test), local search is applied. Suppose $\pi = (2, 5, 4, 1, 3)$ and $\pi' = (3, 4, 2, 1, 5)$ are initial and target solutions respectively. Based on insert moves $\mathsf{insert}(\pi', 3, 1)$, $\mathsf{insert}(\pi', 4, 2)$, and $\mathsf{insert}(\pi', 1, 4)$, two candidate solutions will be created as follows: job 3 placed at position 5 of π will be moved to position 1, which is the position of this job in π'. So, a new intermediate solution

$(3, 2, 5, 4, 1)$ is obtained. Then, with the same procedure, another new intermediate solution $(3, 4, 2, 5, 1)$ is created.

Subset Generation and Reference Set Update

Reference set update is one of the main parts of an SS algorithm. This is because when solutions are close to each other, SS cannot improve the best solution even using the most complex and efficient combination and improvement methods. In fact, in that case the SS algorithm will get trapped in the local optima. Solutions are included in the reference set (R) both for quality and diversity. The subset R_1 of quality solutions contains n_1 best solutions. On the other hand, the subset R_2 of diverse solutions contains the n_2 solutions that are all at least $\mathsf{dist}_{\mathsf{min}}$ distance (described in the following paragraph) apart from s_{best}, the best solution so far.

The distance measure that is used in [95] is defined by the number of edges by which the two solutions differ from each other. For example, if there are two solutions $s_1 = (1, 2, 3, 4, 5, 6)$ and $s_2 = (1, 4, 3, 2, 5, 6)$, then $\mathsf{dist}(s_1, s_2) = 4$ because the edges $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 5)$ in s_1 are different from edges $(1, 4)$, $(2, 5)$, $(3, 2)$, $(4, 3)$ in s_2. Note that edges are considered directed and so $(2, 3)$ and $(3, 2)$ are not the same. In this paper, we change this procedure to be proper with our conditions and consider the edges to be undirected. Therefore, for the previous example, in our distance measure $\mathsf{dist}(s_1, s_2)$ is 2 because only the edges $(1, 2)$, $(4, 5)$ in s_1 did not repeat in s_2. So, the distance, dist, for two solutions lies between 0 and $n-1$ and the larger values represent the more diversity. In this paper, we allow dist to be at least $\mathsf{dist}_{\mathsf{min}} = n/2$.

In this paper, for subset generation, we consider all $\frac{n_1(n_1-1)}{2}$ pairs of solutions (p, q), where $p, q \in R_1$ and all $n_1 \times n_2$ pairs of solutions (p, q), where $p \in R_1$ and $q \in R_2$. Like most other studies [91, 96–98], we update the reference set by replacing the worst solution of R_1 with the new solution if the latter is better. However, continuing this process in updating the reference set although leads to the gathering of good solutions but those solutions appear to be very close to each other. To deal with this problem, we adopt two alternative approaches: *i*) we include a new solution in R_1 if it is $n-1$ distance away from s_{best} (highest possible distance between two solutions based on defined distance measurement), the intuition is that a new solution can only be added in the list only if that solution is greatly diverse from the best solution. It can help the algorithm

to keep up the diversity of search or *ii*) we include any new solution in R_1 with a small probability. In either case, the worst solution in R_1 gets replaced. To obtain R_2, we take the new solutions that are at least $\mathsf{dist}_{\mathsf{min}}$ distance away from the s_{best}. If a sufficient number of new solutions are not $\mathsf{dist}_{\mathsf{min}}$ distance away from s_{best}, we generate new solutions randomly to obtain n_2 solutions in total in R_2.

3.1.7 Proposed Constraint Guided Local Search (CGLS)

To apply our constraint guided strategies on top of a typical local search approach, we use the local search framework shown in Algorithm 12. This local search procedure is a single solution iterative method that has four elements: initialisation, intensification, diversification, and acceptance. The algorithm starts either with a randomly generated solution or a solution generated by a constructive approach. The initial solution is then improved by an intensification procedure. Later, it goes through a loop in which at each iteration, a new solution is generated by using a diversification method and the same intensification method is applied on the generated solution. The solution found in each iteration of the loop is either accepted or discarded using an acceptance criterion.

In MBPFSP problem, a machine could be blocked with the currently finished job until the subsequent machine is available to process the same job. So, the blocking can cause delays in starting the operations of the given jobs on different machines. The blocking constraints thus affect the makespan and so an MBPFSP search algorithm should consider steps explicitly taking the blocking constraints into account. However, existing search algorithms proposed for MBPFSP and closely related problems only use the makespan minimisation as the main criterion to guide the search. The makespan of course include the time wastage due to blocking, but the search otherwise is not aware of the blocking constraints. In this paper, we thus inject blocking constraint-based guidance in different phases of the local search framework in Algorithm 12. We name the resultant algorithm as Constraint Guided Local Search (CGLS) for MBPFSP.

Perform Intensification

In this work, we use an insertion operator since we have developed a very efficient acceleration method for such operators. Moreover, our intensification procedure iteratively

Algorithm 12 Local Search Framework

1. $\pi \leftarrow$ initialiseSolution()
2. $\pi \leftarrow$ performIntensification(π)
3. **while** *termination criteria not satisfied*
   $\pi' \leftarrow$ performDiversification(π)
   $\pi' \leftarrow$ performIntensification(π')
   $\pi \leftarrow$ acceptForNextIteration(π', π)

return the global best solution found so far

improves the current solution: as soon as a better solution is found, it is considered as the current solution. However, our contribution in the intensification procedure is to use a greedy job selection using constraint guidance (greedy neighbourhood generation approach). Because of blocking constraints, machines are blocked with the currently finished job until subsequent machines are available to process the same job. We associate this blocking period of the machine to the job it is currently holding. This way the job that causes the most total blocking time could be considered as the most problematic job in the current sequence. Since, the more the created blocking by a job, the more the delays in the starting times of the next jobs in the sequence. Our idea is to reschedule this job and thus fix the sequence.

In the following equations 3.22, $B(k)$ denotes the total blocking caused by a job $[k]$, $B(i,k)$ denotes the blocking caused by a job $[k]$ at machine i. Jobs are then sorted in descending order of $B(k)$.

$$
\begin{aligned}
B(k) &= \sum_{i=1}^{m-1} (B(i,k)) && (3.22)\\
&\text{where} \\
B(i,k) &= S_{i+1,[k]} - C_{i,[k]} && B_i = \mathsf{RSb} \\
B(i,k) &= S_{i+2,[k]} - C_{i,[k]} && B_i = \mathsf{RCb} \\
B(i,k) &= C_{i+1,[k]} - C_{i,[k]} && B_i = \mathsf{RCb}* \\
B(i,k) &= 0 && B_i = \mathsf{Wb}
\end{aligned}
$$

Once $B(k)$ is obtained for each job $[k]$, we sort the jobs on the descending order of $B(k)$, breaking ties randomly, to get a sequence $\widehat{\pi}$.

Assume a solution $\pi = \langle 2, 3, 1, 4 \rangle$ of 4 jobs that are processed on 5 machines with blocking

order $\langle \mathsf{RCb*}, \mathsf{RSb}, \mathsf{RCb}, \mathsf{RCb*} \rangle$ is depicted in Figure 3.7. As shown, the total blocking times associated with the jobs 2, 3, 1 and 4 are $3+3+2=8$, $2+1+2+2=7$, $2+1+1+2=6$ and $2+2+3=7$ respectively. Therefore, the job ordering is $\widehat{\pi} = \langle 2, 4, 3, 1 \rangle$.

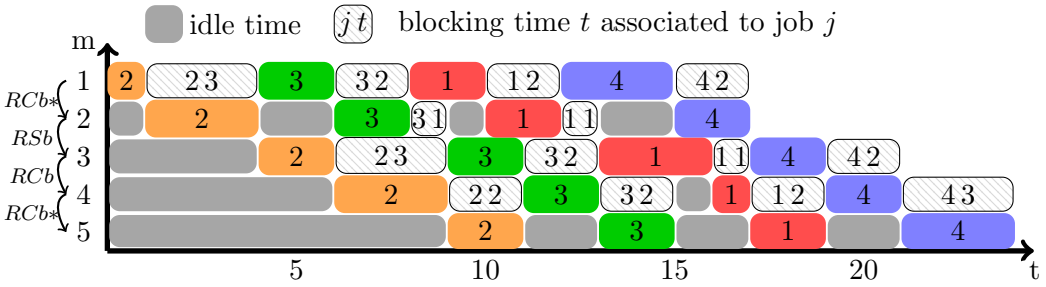


FIGURE 3.7: Blocking times associated with jobs

The outline of the proposed intensification method is given in Algorithm 13. In its first and second steps, it obtains the job ordering $\widehat{\pi}$. In its third step, for each job j in $\widehat{\pi}$ in order, it explores the permutations in $\mathsf{insert\text{-}neighbourhood}(\pi, k')$ where $[k'] = j$. If an improving solution is found, the procedure restarts.

Algorithm 13 performIntensification(π)

1. **foreach** job j, calculate the blocking created by job j.
2. Sort the jobs in the descending order of amount of blocking created by the job to get a permutation $\widehat{\pi} = (\widehat{\pi}_1, \widehat{\pi}_2, ..., \widehat{\pi}_n)$.
3. **foreach** $k = 1$ **to** n **do**
   (a) Take the job $\widehat{\pi}_k$ which is $[k']$ in π and find the permutation π' with the lowest makespan by removing job $[k']$ from π and then reinserting it in all other positions in π. This essentially explores the permutations in $\mathsf{insert\text{-}neighbourhood}(\pi, k')$.
   (b) **if** $C(\pi') < C(\pi)$ **then** let $\pi = \pi'$ and go to Step 1

Perform Diversification

The diversification procedure introduces randomness in search. A number of jobs are removed from a permutation and then reinserted in it. Typically the jobs that are to be removed are selected randomly. Also, the positions where the jobs are inserted are also selected randomly. In this work, we intent to inject greediness in the diversification procedure. In the literature, a number of randomly selected jobs are selected but then inserts each job greedily by finding its best position (just as in NEH) [51]. This procedure

has been used by [34, 76, 99–102]. We introduce greediness even in the selection of jobs for removal restricting randomness. We consider two parameters to control greediness and randomness: TDS is the total diversification size i.e. the number of jobs to be removed and reinserted, and GDS is greedy diversification size i.e. the number of jobs selected greedily. So the number of jobs selected randomly is TDS - GDS.

Greedy Job Selection: The greediness criterion for job selection is based on the wastage time for the job due to blocking constraints and idle times. The simple idea is that a job that has the most gap in its turn should be relocated to a better position. A job $[k]$ should get its turn when job $[k-1]$ is finished by machine 1 and the machine become available. The turn for the job $[k]$ remains active until it is finished by machine m. We define the *turn-wastage time* for a job $[k]$ to be $W_{[k]} = C_{m,[k]} - C_{1,[k-1]} - P_{[k]}$, which also takes $P_{[k]}$ the time spent for processing the job into account. After computing $W_{[k]}$ for each job $[k]$, we select GDS jobs that have the top $W_{[k]}$ values. For the MBPFSP in Table 3.2 with blocking constraints $\langle \mathsf{RSb}, \mathsf{Wb}, \mathsf{RCb}, \mathsf{RCb*} \rangle$, assume $\pi = \langle 1, 2, 3, 4 \rangle$ is permutation. The corresponding schedule is shown in Figure 3.8. We calculate $W_{[1]} = 10 - 0 - 10 = 0$, $W_{[2]} = 15 - 1 - 11 = 3$, $W_{[3]} = 19 - 3 - 10 = 6$, $W_{[4]} = 23 - 6 - 12 = 5$. Therefore, if GDS $= 2$, then jobs [3] and [4] are selected greedily.

TABLE 3.2: An example of 4 jobs and 5 machines.

Job	Machine i					Processing
j	1	2	3	4	5	Time P_j
1	1	3	1	3	2	10
2	2	3	1	2	3	11
3	2	2	2	3	1	10
4	3	4	2	1	2	12

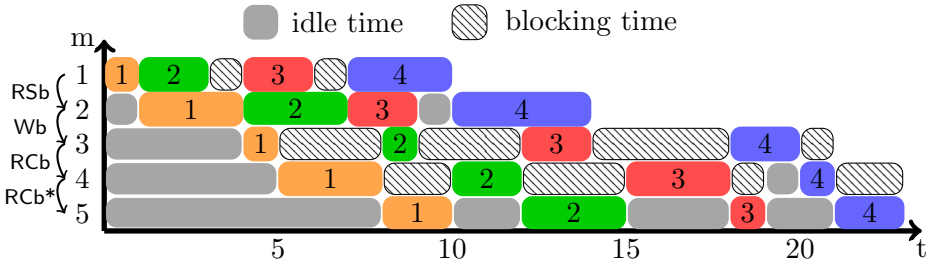


FIGURE 3.8: An example schedule for an MBPFSP in Table 3.2

Intensification of Partial Solutions: Local search have been applied in various ways on partial sequence by [70, 100, 103]. In this work, we first remove the selected TDS jobs from the current solution π to get a permutation π'. We apply the same intensification

procedure in Section 3.1.7 on π' to obtain a better permutation. We then insert each of the $\mathsf{TDS} - \mathsf{GDS}$ randomly selected jobs in the permutation, finding the best position for each job inserted. In the same way, we insert each of the GDS jobs, finding the best position for each job inserted. Algorithm 14 shows the pseudo-code of the procedure.

Algorithm 14 performDiversification(π)

1: Let TDS, and GDS are the parameters where $\mathsf{GDS} \leq \mathsf{TDS} \leq n$.
2: Calculate the wastage time for each job. List the jobs in the descending order of wastage time.
3: Take the first GDS jobs greedily from the list mentioned above and then take $\mathsf{TDS} - \mathsf{GDS}$ jobs randomly from the rest of the jobs to construct a sequence $\widetilde{\pi}$.
4: Assume π' is obtained after removing the selected TDS jobs from π.
5: $\pi' \leftarrow$ performLocalSearch (π').
6: **for** $k = \mathsf{TDS}$ down to 1 **do**
7: Find the sequence π'' with the smallest makespan by inserting the kth job from $\widetilde{\pi}$ in all possible positions of π'.
8: Let $\pi' = \pi''$.
9: **return:** π' as the output solution.

Lemma 3.5. *If makespans are computed from scratch, Algorithm 14 runs in $\mathcal{O}(n^2m)$ time using $\mathcal{O}(n)$ space. If makespans are computed incrementally, Algorithm 14 runs in $\mathcal{O}(nm)$ time.*

Proof. Calculation of the blocking created by the jobs needs $\mathcal{O}(n)$ time and $O(n)$ space and then sorting needs $\mathcal{O}(n \lg n)$ time. Then, for a given constant TDS times, selected jobs are reinserted in the partial sequences to obtain $O(n)$ partial sequences. Consequently, this step runs in $\mathcal{O}(n^2m)$ time if makespans are computed from scratch. When a job is reinserted in all possible positions, partials solutions obtained by inserting the job in the later positions could reuse computations done when inserting the job in the earlier positions. Therefore, for incremental computation, the time complexity becomes $\mathcal{O}(nm)$. □

Acceptance Criterion

Among many acceptance criterion procedure used in the literature, the SA-based one proposed by [104] and used by [51] is the most common one. In this function, acceptance probability $P = \min(e^{-\Delta/T}, 1)$ where $\Delta = C(\pi') - C(\pi)$, π is the current solution before local search, π' is the candidate solution to be accepted or not, $T = \frac{\sum_{j=1}^{n}\sum_{i=1}^{n} p_{ij}}{10 \cdot m \cdot n} \times T_0$ be the temperature, and T_0 is a given parameter.

However, the above acceptance criterion has some drawbacks. If P is very small later in the search, then most non-improving solution will be rejected. This will essentially turn into a hill climbing search than remaining as a simulated annealing algorithm. For example, with typical values of $T_0 = 0.5$ and $\frac{\sum_{j=1}^{n}\sum_{i=1}^{n} p_{ij}}{10 \cdot m \cdot n} \approx 5$, the value T could come approximately 2.5. When $\Delta = C(\pi') - C(\pi) = 20$, P approximately becomes 0.03%, which could be considered very low. Moreover, in medium or large sized problems depending on the magnitude of $C(\pi)$, a solution π' with $\Delta = 20$ could be an acceptable non-improving solution.

In this work, we use a very simple but effective acceptance criterion that to a large extent takes the magnitude of $C(\pi)$ into account. For this, we consider an initial Δ_0 that is the difference in the makespan achieved by the insertion procedure of the INEHtie algorithm. The value of Δ_0 serves as a reference for all Δ that could be achieved later during the local search phase. In fact the following threshold based simple formula in (3.23) is used to determine acceptance probability P where TAR (Threshold Acceptance Ratio) is a parameter.

$$P = \begin{cases} 1 & \text{if } \Delta \leq \mathsf{TAR} \times \Delta_0, \\ 0 & \text{otehrwise }, \end{cases} \tag{3.23}$$

Based on this threshold if the difference of the solution after local search and the current solution in terms of the objective is less than $\Delta_0 \times TAR$, the newly obtained solution will be the current solution. Note that each problem instance with different size or even the same size will have specific character. So, using the same acceptance criterion is not much reasonable. This newly proposed acceptance criterion gives a chance to the algorithm to obtain threshold specificity for each problem size and even problem instances. Although threshold acceptance criterion has been used before [10], the problem-dependent way of calculating the threshold is newly proposed here. Moreover, the use of makespan difference obtained by the INEHtie initialisation algorithm in determining the threshold is also new.

3.1.8 Computational Results

We perform experiments to test the effectiveness of the proposed INEH and INEHtie initialisation heuristics, to evaluate the components of the proposed SS and CGLS algorithms, and to compare with various existing algorithms. We use the three following benchmark sets in our experiments:

1. **Taillard Instances.** The first benchmark problem set is the 120 Taillard instances [1]. This benchmark set is made up of 12 combinations of numbers of jobs n and numbers of machines m. The $n \times m$ combinations are $\{20, 50, 100\} \times \{5, 10, 20\}$, $\{200\} \times \{10, 20\}$, and $\{500\} \times \{20\}$, each with 10 different instances, in total of 120 instances. As can be seen, three combinations are missing: $\{200\} \times \{5\}$, and $\{500\} \times \{5, 10\}$. [38] added these 30 instances. Including these new 30 instances, the Taillard benchmark would have 150 instances in total. In this paper, all 150 instances are used. Taillard instances are our main benchmark set. Because, firstly, this benchmark set is the most well-known benchmark set in PFSP literature, and also, secondly, these instances include various numbers of machines and jobs from small, medium, and large instance sized.

2. **VRF Instances.** The second benchmark set has 480 new hard VRF instances [3]. It includes 240 small and 240 large instances. In this paper, only 240 large instances are used. These 240 large instances are made up of 24 $n \times m$ combinations as $\{100, 200, 300, 400, 500, 600, 700, 800\} \times \{20, 40, 60\}$. Ten instances of each combination make a total of 240 instances.

3. **LY Instances.** The third benchmark set has 30 very large LY instances [4] using the same experiment design as Taillard instances [1]. These instances comprise three combinations of $n = \{1000, 1500, 2000\}$ with 20 machines.

We compare our INEH, SS and CGLS algorithms with the MBPFSP algorithms listed below. The first two are for MBPFSP, but the rest are for related problems and are adapted to MBPFSP.

1. **BCO:** a bee colony optimisation algorithm [16]
2. **IGA-RS:** an iterated greedy algorithm [51]

3. **HBS:** a hybrid backtracking search algorithm [74]
4. **TPA:** a three-phase algorithm [14]
5. **IGA-R:** an iterated greedy algorithm [76]
6. **ABC-DE:** a hybrid of artificial bee colony algorithm and differential evolution [77]
7. **IG_IJ:** an iterated greedy algorithm [39]

All methods have been reconstructed in programming language C and are run on the same high performance computing cluster Gowonda at *Griffith University*. Each node of the cluster is equipped with Intel Xeon CPU E5-2670 processors @2.60 GHz, FDR 4x InfiniBand Interconnect, having system peak performance 18,949.2 Gflops. Moreover, for a fair comparison, the proposed acceleration method is also used in all compared algorithms.

To analyse the experimental results, we have used the relative percentage deviation (RPD) calculated as

$$RPD = \frac{C - C_{\text{ref}}}{C_{\text{ref}}} \times 100 \tag{3.24}$$

where C is the makespan value generated by a given run for a given instance, and C_{ref} is the reference makespan. For constructive heuristics, C_{ref} is the minimum makespan obtained by any of the heuristics for the given problem instance. For various variants of the SS/CGLS to test the effectiveness of each of the components, C_{ref} is the best solution found by any of the variants. For metaheuristic algorithms compared with proposed algorithms, C_{ref} is the best known solution for the instance over any algorithm. For heuristics, each algorithm is run on each instance only once. For metaheuristics, we run each algorithm on each instance 5 times and compute average RPD (ARPD) over the 5 runs. We also compute a further average of RPDs or ARPDs over all 10 instances in each group or even over all instances in a benchmark set.

Analysis of Constructive Heuristics

We get the following 5 NEH-based constructive heuristics from the initialisation methods of the 8 metaheuristic algorithms that we mentioned before. We compare the proposed INEHtie heuristic with these.

1. **NEH:** originally proposed by [65] for PFSP, and later used in BCO [16] for MBPFSP and in IGA [51] for PFSP
2. **mNEH:** proposed in TPA [69] for BPFSP with RSb constraints
3. **MME:** used in IGA-R [76] and in ABC-DE [77], both for BPFSP with RSb constraints
4. **PFT-NEH:** proposed in IG_IJ [39] for BPFSP with RSb constraints
5. **NNEH:** proposed in SS [7] for MBPFSP

The comparison is done using the 150 Taillard instances [1]. Table 3.3 (bottom) shows that computation of the heuristics often take negligible times. Table 3.3 (top) shows the ARPD values obtained for all the heuristics compared. In problem instances where $n = 20$, we see that PFT_NEH obtains significantly better results over other heuristics. The reason is that PFT_NEH generates k ($k = n$ if $n \leq 200$; else $k = 20$) different solutions, and each of the solutions is then improved by insertion phase of NEH applied only on the last 20 jobs. This means for $n = 20$, PFT_NEH will have 20 complete NEH algorithms. Thus this is not surprising that PFT_NEH finds better results in this case. Nevertheless, we see that in 12 out of 15 instance groups the INEHtie wins. Especially those with large problem instances, INEHtie hugely improves over the other heuristics. However, to see the performance difference of the heuristics from statistical point of view, 95% confidence intervals with Tukey's Honest Significant Difference (HSD) is shown in Figure 3.9 Left. First, note that Tukey's HSD is a single-step multiple comparison procedure and statistical test based on raw data or in conjunction with an ANOVA to find means that are significantly different from each other [105]. Clearly, our heuristic significantly outperforms the other heuristics. We also compared all constructive heuristics on the very hard VRF instances and LY instances. For the VRF instances, we show the 95% confidence intervals with Tukey's HSD in Figure 3.9 Right. The conclusions remain the same both on the VFF and the LY instances.

One could be curious to know how other tie-breaking heuristics perform compared to our proposed tie-breaking heuristic. To this end, we compare the proposed TWT based tie-breaking with a number of other common tie-breaking methods listed below:

1. **FT:** chooses the first one over any subsequent ones [65]

TABLE 3.3: Comparison of constructive heuristic algorithms.

ARPD						
Instance	NEH	mNEH	MME	PFT_NEH	NNEH	INEHtie
20×5	2.918	3.165	2.938	**0.319**	2.891	2.555
20×10	2.151	2.123	2.415	**0.354**	2.299	1.460
20×20	1.984	1.864	1.901	**0.423**	2.744	1.416
50×5	0.908	1.020	0.624	4.687	0.674	**0.031**
50×10	1.462	1.508	1.489	5.612	0.759	**0.251**
50×20	2.068	1.935	2.138	8.373	1.168	**0.237**
100×5	0.513	0.340	0.281	2.931	0.430	**0.117**
100×10	1.179	1.317	1.106	3.718	0.998	**0.052**
100×20	1.277	1.093	0.974	5.459	1.064	**0.126**
200×5	0.294	0.440	0.505	2.819	0.488	**0.037**
200×10	0.878	0.747	0.754	3.262	0.766	**0.107**
200×20	1.450	1.356	1.190	2.991	1.155	**0.148**
500×5	0.314	0.269	0.308	2.441	0.338	**0.037**
500×10	0.831	0.739	0.644	2.478	0.696	**0.139**
500×20	0.917	0.843	0.813	3.218	0.738	**0.194**
Average	1.276	1.250	1.205	3.272	1.147	**0.460**
CPU Time						
Instance	NEH	mNEH	MME	PFT_NEH	NNEH	INEHtie
20×5	0.000	0.000	0.000	0.008	0.000	0.001
20×10	0.001	0.000	0.001	0.015	0.000	0.001
20×20	0.001	0.001	0.001	0.025	0.000	0.001
50×5	0.001	0.001	0.001	0.045	0.000	0.002
50×10	0.003	0.003	0.004	0.059	0.001	0.004
50×20	0.004	0.004	0.005	0.086	0.001	0.004
100×5	0.005	0.005	0.005	0.157	0.001	0.019
100×10	0.011	0.010	0.010	0.265	0.002	0.012
100×20	0.019	0.014	0.008	0.493	0.005	0.016
200×5	0.020	0.014	0.008	1.068	0.005	0.041
200×10	0.026	0.012	0.013	1.908	0.010	0.058
200×20	0.044	0.022	0.024	3.645	0.020	0.073
500×5	0.058	0.035	0.040	0.663	0.035	0.159
500×10	0.090	0.066	0.077	1.142	0.067	0.401
500×20	0.158	0.131	0.155	2.170	0.132	0.503
Average	0.030	0.021	0.023	0.783	0.019	0.086

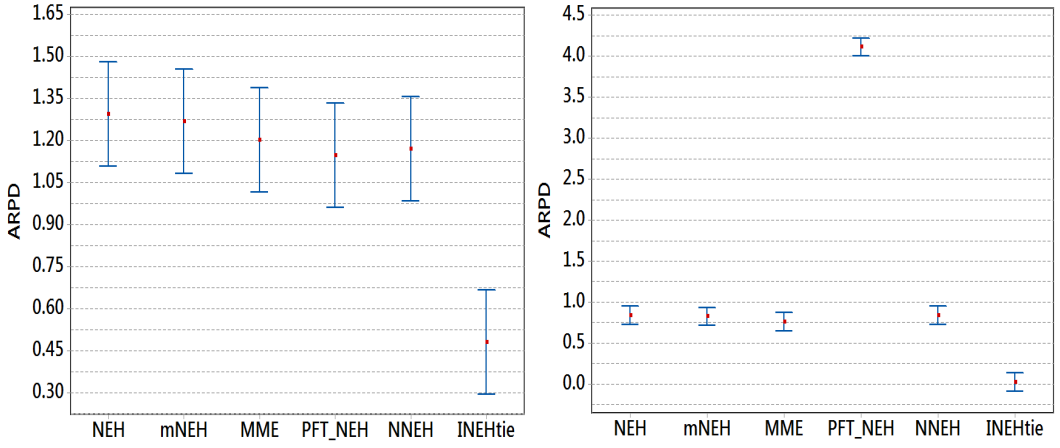


FIGURE 3.9: ARPD and 95% confidence intervals with Tukey's HSD for different heuristics: Left) Taillard's instances Right) VRF instances.

2. **LT:** chooses last one over any preceding ones
3. **TWT:** the exact tie-breaking rule proposed in this paper
4. **TWT-NF:** same as TWT but excludes the front delays (idle times created by the job at the first position)
5. **TC:** selects the one with less total completion time: $CT = \sum_{j=1}^{k} C_{m,[k]}$
6. **IT1:** selects the one with less $IT1 = \sum_{i=1}^{m}(C_{i,[n]}) - \sum_{j=1}^{n} P_j$[83]
7. **IT2:** same as the IT1 but just excludes the front delays

All these tie-breaking rules are used on top of proposed INEH. The results for the Taillard's instances are given in Table 3.4. From this table, LT obtained the worst results because its selection based on the inverse of the initial ordering in INEH is in conflict with blocking constraints. Other tie-breaking rules achieved better results than FT where FT can be seen as absence of any explicit tie breaking method. The proposed tie-breaking obtained better results in average ARPDs compared to the other tie-breaking rules although not statistically significantly. One might comment that TWT and IT1 (and so TWT-NF and IT2) are similar to each other. However, TWT uses the leave time of the last job while IT1 uses its completion time in the respective calculations. Moreover, TWT selects in favour of better resource utilisation while TC selects in favour customer satisfaction. Similar conclusions can be drawn from the results for the VRF and LY instances and we do not show them. It is worth noting that our finding of insignificant differences between the performances of various tie-breaking techniques for MBPFSP might appear to be somewhat contradictory to the findings reported by an extensive study performed by [106] on handling ties for PFSP. First, MBPFSPs have significantly different characteristics than PFSPs. Second, our findings are based on the comparison of only the tie-breaking methods listed above, on the benchmark instances that we use for MBPFSP with a uniform distribution of blocking constraints, and on our implementation of all the methods on top of our proposed INEH. We emphasise that the main objective of this paper is not to perform an exhaustive study on various tie-breaking methods for MBPFSP; which itself requires significant research effort and is clearly out of scope of this work.

TABLE 3.4: Comparison of proposed heuristic with different tie-breaking rules.

Instance	FT	LT	TWT	TWT-NF	TC	IT1	IT2
20×5	0.155	0.990	0.093	0.174	0.069	0.058	0.139
20×10	0.615	2.125	0.235	0.235	0.619	0.294	0.294
20×20	0.652	2.606	0.726	0.726	0.205	0.626	0.626
50×5	0.147	1.821	0.129	0.129	0.160	0.091	0.091
50×10	0.277	2.224	0.353	0.353	0.217	0.273	0.273
50×20	0.355	2.708	0.291	0.291	0.170	0.291	0.291
100×5	0.094	1.429	0.081	0.081	0.023	0.031	0.031
100×10	0.431	2.199	0.130	0.130	0.491	0.291	0.291
100×20	0.183	1.952	0.192	0.192	0.258	0.174	0.174
200×5	0.140	1.771	0.113	0.134	0.086	0.151	0.082
200×10	0.223	2.079	0.096	0.096	0.256	0.154	0.154
200×20	0.203	2.217	0.151	0.151	0.443	0.160	0.160
500×5	0.063	1.760	0.067	0.065	0.063	0.077	0.084
500×10	0.112	2.333	0.175	0.175	0.145	0.211	0.211
500×20	0.130	1.995	0.187	0.187	0.219	0.195	0.195
Average	0.251	2.012	0.200	0.207	0.228	0.204	0.205

Overall, unless mentioned otherwise, in the rest of the experiments, we will use INEHtie as the initialisation method in the final version of SS and CGLS. In addition, for a fair comparison, INEHtie method is also used in all compared metaheuristic algorithms.

SS Parameter Tuning

Owing to the fact that the proficiency of an algorithm depends on the parameters, initial experiments were conducted with some of potential parameter values to find the best combinations. In our SS algorithm, there are four parameters: the size of the population (Size of Pop), the size of R_1 and that of R_2, and the maximum number of iterations maxIter used in the ILS algorithm. In this paper, we used the Design of Experiments method [107] to examine the effect of each factors. Here, we considered four values $(25, 35, 45, 55)$ for Size of Pop and five values $(4, 7, 10, 13, 16)$ for maxIter. For the sizes of R_1 and R_2, due to the fact that these sizes have a direct impact on each other, we consider them as a pair. So, pairs $(5, 10), (3, 7), (7, 3), (10, 5)$ are considered for (R_1, R_2). Note that all selected values are fixed after an extensive preliminary test. Thus, combining different values of the parameters, we obtain a $4 \times 5 \times 4 = 80$ full factorial experiments. For all 80 different configurations, 60 instances with different combinations of n and m ($n \in \{50, 150, 250, 350\}$, and $m \in \{10, 30, 50\}$), are randomly generated with the common problem generation method of [1] . Note that we use a different benchmark set for parameter calibration compared to ones used to compare methods owing to the fact that testing the same benchmark for both calibration and

compared methods would lead to biased results [108]. Each problem is tested 5 times with specific parameters. So, there were totally $80 \times 50 \times 5 = 20000$ runs in the initial experiments. It is also worth pointing out that we use a termination criterion set to a maximum elapsed CPU time $60nm$ milliseconds to have a fair comparison.

Figure 3.10 shows the average RPD over all instances all runs for each value mentioned above for each of the four parameters varying the remaining other three parameters. Based on the results shown in the figure, the selected value of Size of Pop is 45, that of (R_1, R_2) is $(7, 3)$, and that of maxIter is 13. For further details, increasing the number of population improves the performance of our algorithm until size 45. When the number of R_1 is more than that of R_2, our algorithm has better results. Moreover, our algorithm has gradually better performance as we increase maxIter from 4 to 13, but after that no significant difference is observed. Also, using 3 more iterations for maxIter needs more run time however no better results are provided.

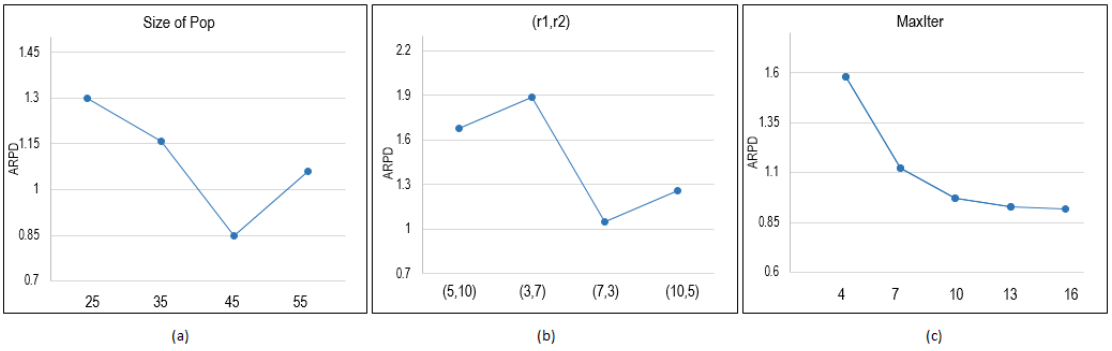


FIGURE 3.10: Effects of different parameter values on our SS.

CGLS Parameter Tuning

We use the design of experiments (DoE) method [109] for parameter tuning of the proposed CGLS algorithm, which has three parameters to be adjusted: TDS the total number of jobs to be removed from the current solution in the diversification procedure, GDS the number of jobs to be removed from the current solution based on the greedy criterion of wastage time, and TAR the threshold acceptance ratio. The parameters TDS and GDS are dependent on each other. For instance, when TDS is 3, GDS can not be 4 or more. So, we converted TDS and GDS into a single parameter DS that takes the values shown in Table 3.5. Note that TDS = GDS is not considered because this might lead to the repetition of producing the same partial solution in each iteration in the

diversification procedure. Besides the TDS, for TAR the following levels are also chosen: 0.01, 0.001, 0.0001.

TABLE 3.5: Combination of TDS and GDS values for each of DS levels.

DS	1	2	3	4	5	6	7	8	9	10	11	12
TDS	3	3	3	4	4	4	4	5	5	5	5	5
GDS	0	1	2	0	1	2	3	0	1	2	3	4

For this experiment, we have randomly generated 60 instances with $n \in \{50, 150, 250, 350\}$, and $m \in \{10, 30, 50\}$. We use a termination criterion set to a maximum elapsed CPU time $60nm$ milliseconds. Each combination was executed for 5 times. We have a full factorial design of experiments resulting in $12[\mathsf{DS}] \times 3[\mathsf{TAR}] \times 5[\text{replication}] \times 60[\text{instance}] = 10800$ runs.

TABLE 3.6: ANOVA table for the two parameters.

Source	DF	SS	MS	F	P-value
TAR	2	8.97	4.4863	48.14	**0.000**
DS	11	184.56	16.7778	180.05	**0.000**
TAR $*$ DS	22	0.53	0.0239	0.26	**1.000**
Error	10764	1003.0	0.0932		
Total	10799	11.97.08			

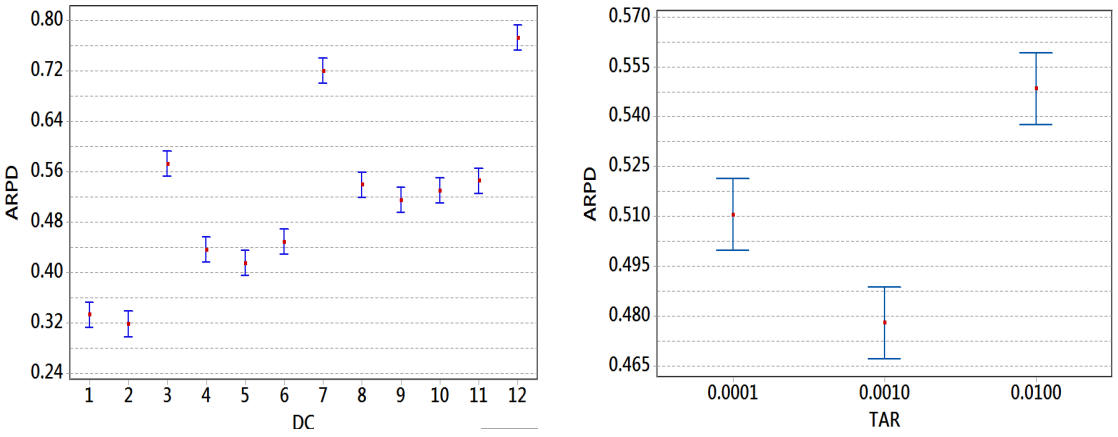


FIGURE 3.11: ARPD and 95% confidence intervals with Tukey's HSD for different values of parameters, DS: Left and TAR: Right.

The ANOVA results are given in Table 3.6. Note that three main ANOVA hypotheses i.e., normality, homoskedasticity of the different levels, and independence of the residuals are checked and no significant deviations were found that could raise the question of validity. From the table, the DS is the most significant factor due to highest F-ratio.

The interval plots of these factors considering 95% with Tukey's HSD are shown in Figure 3.11. From Figure 3.11-left, for the parameter DS, levels 1 and 2 obtain statistically better results compared to other selected levels. Level 2 also yield a lower average over level 1 but statistically equivalent. In addition, interestingly, cases 2,5, and 9 find better results than cases 1,4, and 8 respectively. This means using the proposed greedy job selection (based on *turn-wastage time*) improves the algorithm's performance. However, as a possible explanation of the observed result, using the greedy selection more than random selection decreases the quality because of the lack of diversity. In terms of parameter TAR, level 0.001 is statistically better than other levels. Level 0.01 is the worst case because it is kind of weak acceptance i.e. accepting mostly the worse solutions. However, level 0.0001 is the strong acceptance which accept worse solution rarely i.e. with a very small probability. From Figure 3.11-right, we can conclude that for this problem and for the proposed algorithm, accepting worse solutions with very low probability is better than accepting any worse solutions. In addition, level 0.001 keeps balance between these two scenarios. Overall, we fix TAR to 0.001 and DS to 2, TDS = 3, and GDS = 1.

Parameter Tuning of Other Methods Compared

In this paper, the proposed algorithm is compared against 8 algorithms borrowed from the literature. From the 8 algorithms, two are proposed exactly for MBPFSP with makespan objective. For these two algorithms, at the time of reimplementation, their parameters are exactly taken from the respective paper. The other six algorithms are not proposed for the MBPFSP, rather adapted for MBPFSP. For IGA-RS, [51] showed that the temperature factor is very robust. The same is in IG_IJ [39] as well. So, the temperature factor for these two algorithms are not tested here and their values are taken from the original paper directly.

For other parameters of the six algorithms, we have performed parameter tuning. We have used the same experimental design as is used in the parameter tuning of CGLS, using the same 60 instances generated, $60nm$ milliseconds as timeout, and executing each instance 5 times. We only show the selected values of the parameters in Table 3.26. Moreover, ARPD and 95% confidence intervals with HSD for different values of the algorithm parameters are provided as online material.

TABLE 3.7: The selected parameter values of the algorithms compared.

Algorithm	Parameter	Selected value
BCO [16]	P_{size}	10
	p_{LS}	0.05
	MaxIter	$P_{size} \times 2$
	MaxLoop	$P_{size} \times 0.5$
IG_IJ [39]	dS	5
	tP	0.5
	jP	0.01
IGA-RS [51]	d	3
	T	0.4
IGA-R [76]	d	4
	γ	7
TPA [14]	N_{iter}	200000
HBS [74]	P	30
	T_0	30
	λ	0.90
	Mu	0.2
	CR	0.9
ABC-DE [77]	PS	15
	pmu	0.9
	pc	0.3
	pls	0.2

Analysis of Proposed Acceleration Method

In this section, we would like to show the performance of the proposed acceleration method. To do that, two different tests are employed. In the first test, we show the CPU times of INEH with and without the proposed speed up method, denoted as INEH and INEH_{NS} respectively. This test is done on the 60 generated instances explained in Section 3.1.8 and shown in Figure 3.12. Based on the results, the INEH is obviously faster than INEH_{NS}.

As the second test, we run the proposed CGLS with and without speedup, denoted as CGLS and CGLS_{NS} respectively. This test is run on the 60 generated instances explained in Section 3.1.8. These two algorithms are run on each instance 5 times and with the same stopping criterion of $90nm$ milliseconds. The ARPDs (with respect to the minimum makespan obtained by any of variants for the given problem instance) and

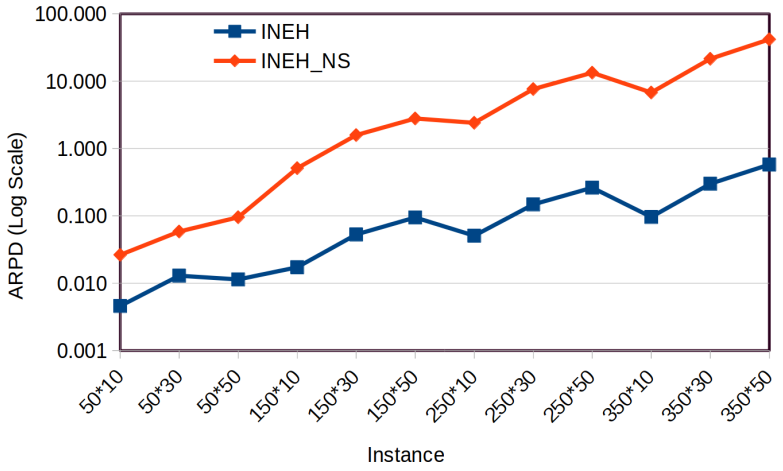


FIGURE 3.12: Effect of the acceleration method on INEH

average iterations of each instance group is shown in Table 3.8. From this table, we can see that the proposed acceleration method helps run hugely more iterations (about 97.559%) than when not used. Note that for large problem instances with $n \geq 350$, CGLS$_{NS}$ cannot finish even one iteration within the time limit in most of the cases. We also show the interval plot of the algorithms based on ARPDs in Figure 3.13. As can be seen from the figure, there is a statistically very significant difference between CGLS and CGLS$_{NS}$.

TABLE 3.8: ARPD and number of iterations of CGLS and CGLS$_{NS}$.

Instance	ARPD		Iteration		Iteration
n×m	CGLS	CGLS$_{NS}$	CGLS	CGLS$_{NS}$	diff (%)
50×10	**0.116**	0.723	**2856.72**	177.04	93.803
50×30	**0.179**	0.990	**2151.96**	149.24	93.065
50×50	**0.261**	0.878	**1766.76**	131.88	92.535
150×10	**0.189**	0.679	**1029.48**	17.24	98.325
150×30	**0.244**	1.298	**536.84**	9.72	98.189
150×50	**0.197**	1.126	**507.2**	9.28	98.170
250×10	**0.172**	1.117	**315.72**	2.44	99.227
250×30	**0.156**	1.388	**316.68**	2.44	99.230
250×50	**0.150**	1.214	**255**	1.92	99.247
350×10	**0.147**	0.912	**224.36**	1	99.554
350×30	**0.209**	1.411	**176.56**	0.44	99.751
350×50	**0.206**	1.282	**154.1**	0.6	99.611
average	**0.186**	1.085	**857.615**	41.937	97.559

Effectiveness of SS Components

Evaluating ILS Algorithm: In the proposed SS algorithm, we use an ILS algorithm as an improvement method. Our ILS uses three new elements: insertR, swapR, and

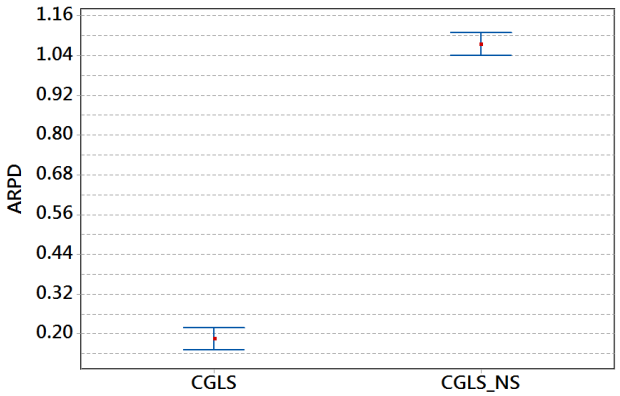


FIGURE 3.13: 95% confidence intervals with Tukey's HSD for CGLS and CGLS_{NS}.

NEH-based perturbation. Moreover, a **greedy** selection is used instead of a **random** selection in insertR and swapR. To evaluate the use of these elements, we consider the following seven versions of ILS, where the PnSg is actually our final algorithm.

1. PnSg: NEH-based perturbation, greedy selection, both insertR, swapR
2. PnSr: NEH-based perturbation, random selection, both insertR, swapR
3. PnSgI: NEH-based perturbation, greedy selection, only insertR
4. PnSrI: NEH-based perturbation, random selection, only insertR
5. PnSgS: NEH-based perturbation, greedy selection, only swapR
6. PnSrS: NEH-based perturbation, random selection, only swapR
7. PiSg: Insert-based perturbation, greedy selection, both insertR, swapR

To show the efficiency of the proposed local search, we also consider the local search used by [12] and denote it by TLS. In Table 3.9, we show the performance of all these algorithms in terms of the average of the RPDs over all runs all instances of each size. For this experiment, we use $90nm$ milliseconds as the stopping criterion and aforementioned 120 benchmark instances of [1]. Each instance was run five ($R = 5$) times with each version of the ILS.

TABLE 3.9: Comparison of our ILS variants in terms of ARPD.

instance	PnSg	PnSr	PnSgI	PnSrI	PnSgS	PnSrS	PiSg	TLS

Continued on next page

Table 3.9 – *Continued*

20×5	**0.399**	2.264	0.499	2.758	1.426	3.265	3.582	2.245
20×10	**0.843**	2.114	1.434	2.967	1.524	3.056	3.386	2.117
20×20	**1.047**	3.040	1.947	5.326	2.364	5.425	3.988	2.719
50×5	**0.419**	1.650	0.685	2.620	1.359	3.265	2.687	2.879
50×10	**0.903**	1.966	1.603	3.956	1.564	4.582	2.722	3.252
50×20	**1.071**	3.966	1.681	4.986	4.562	5.124	3.150	2.124
100×5	**0.367**	1.985	0.497	2.234	2.606	2.986	1.587	1.680
100×10	**0.831**	1.816	0.931	2.301	1.040	3.451	1.690	2.783
100×20	**2.069**	3.039	2.469	5.127	4.285	5.985	3.775	3.357
200×10	**1.476**	2.051	1.976	2.614	2.958	3.452	2.237	2.465
200×20	**1.040**	3.240	1.240	4.582	2.234	6.320	2.518	3.194
500×20	**0.143**	2.129	0.443	2.469	1.231	2.851	1.641	1.129
Average	**0.884**	2.438	1.284	3.495	2.263	4.147	2.747	2.495

Based on the results in Table 3.9, we conclude that each of the new elements of ILS algorithm significantly improves its performance. Comparison of PnSg and PiSg shows the efficiency of using NEH-based perturbation over insertion-based perturbation. Comparison of PnSgI and PnSgS and similarly comparison of PnSrI and PnSrS indicate the effectiveness of insertR over swapR. Comparison of PnSg with PnSgI and PnSgS shows the combined use of insertR and swapR is better than the use of insertR on its own which is better than swapR on its own. Comparison of PnSg and PnSr shows the efficiency of greedy selection over random selection. Finally, comparison of PnSg and TLS shows that the presented local search is better than Trabelsi's local search [12].

Evaluating SS Algorithm Components : We evaluate the effectiveness of using INEH heuristic in generating one initial solution, applying the path relinking as our combination method, accepting worse solutions with some conditions in the reference set update phase, and our modified distance measurement. For these reasons, we create four versions of the SS algorithm that are described below. These versions along with our final SS algorithm are run on the 120 benchmark instances of [1] with five independent replications.

1. SS: our final SS algorithm described in Section 3.1.6.

2. SS-R: all initial solutions are randomly generated while in SS one solution is generated using INEH heuristic and the rest randomly.

3. SS-TC: using a typical combination method such as the crossover used by [12]. Note SS uses the path relinking.

4. SS-NWS: changing the reference set updating method in SS with accepting only solutions better than the worst solution of R_1.

5. SS-D: using the same distance measure used by [95] instead of the measure used in SS.

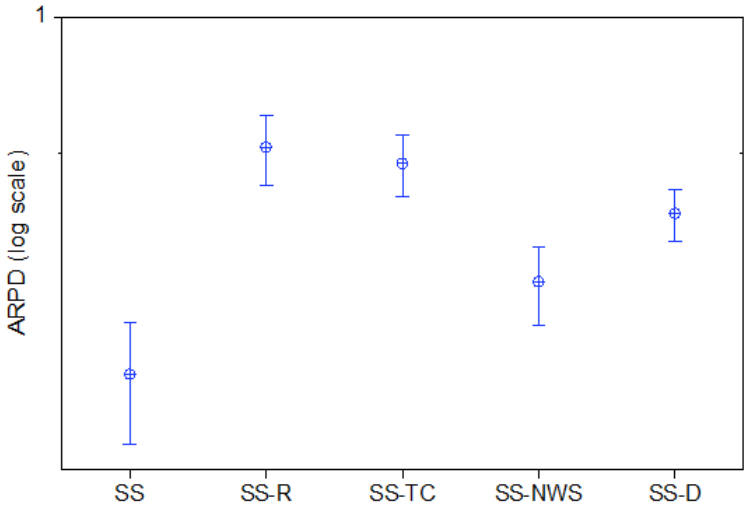


FIGURE 3.14: 95% Confidence interval for SS, SS-R, SS-TC, SS-NWS, and SS-D.

In Figure 3.14, we present a 95% confidence interval plot to show the performances of the SS versions. It is clear from the plot that the SS algorithm performs better when any of the component is used than when the same component is not used.

Effectiveness of CGLS Components

The proposed CGLS has a new blocking constraint aware greedy neighbourhood generation approach (GN) in the intensification procedure, a blocking constraint aware diversification procedure (DP), and a new problem dependant threshold based acceptance criterion (AC). To test the effectiveness of each of these components, we create the following five versions.

1. **CGLS:** Proposed final algorithm that includes the GN, DP, and AC components.

2. **SAA:** CGLS but AC is replaced by an *SA-based acceptance criterion* of [51].

3. **NIPS:** CGLS but no intensification of the partial solution in the diversification phase.

4. **RN**: CGLS but GN is replaced by a random neighbourhood generation approach.

5. **DBT:** CGLS but GN is replaced by a greedy neighbourhood generation approach based an idea named *differential blocking time* [7]. In essence, a different job sorting order is used.

In these experiments, the same stopping criterion of $90nm$ milliseconds is used for all algorithms listed above. Also, the benchmark problem instances are the 60 generated problem instances. All variants are run 5 times for each instance. The ARPDs (with respect to the minimum makespan obtained by any of the variants for the given problem instance) are given in Table 3.10.

Table 3.10: Comparison of CGLS variants

Instance	CGLS	SAA	NIPS	RN	DBT
50×10	**0.271**	0.282	0.376	0.354	0.383
50×30	**0.311**	0.344	0.421	0.350	0.357
50×50	**0.272**	0.333	0.561	0.356	0.311
150×10	**0.165**	0.267	0.439	0.278	0.198
150×30	**0.227**	0.440	0.687	0.448	0.365
150×50	**0.273**	0.414	0.817	0.404	0.365
250×10	**0.184**	0.390	0.626	0.348	0.277
250×30	**0.205**	0.434	0.722	0.398	0.279
250×50	**0.219**	0.482	0.724	0.491	0.221
350×10	0.196	0.314	0.496	0.281	**0.151**
350×30	**0.264**	0.586	0.838	0.620	0.333
350×50	**0.169**	0.408	0.700	0.387	0.178
average	**0.230**	0.391	0.617	0.393	0.285

As can be seen from Table 3.10, the CGLS version with all new components clearly outperform the other variants and lack of each new element makes the algorithm weaker. In other words, each of the new ideas used in the proposed CGLS is important and has a great impact in the performance. However, to ensure the statistical significance of the performance difference, a 95% confidence intervals with Tukey's HSD with a significance level of 0.05 is performed.

According to Figure 3.15, as expected, CGLS is statistically better than DBT, RN, SAA, and NIPS. This test reveals that applying intensification on the partial solution in the diversification phase is the most crucial part of the algorithm as the lack of it in the algorithm hugely decrease CGLS's performance. Overall, these results confirm the

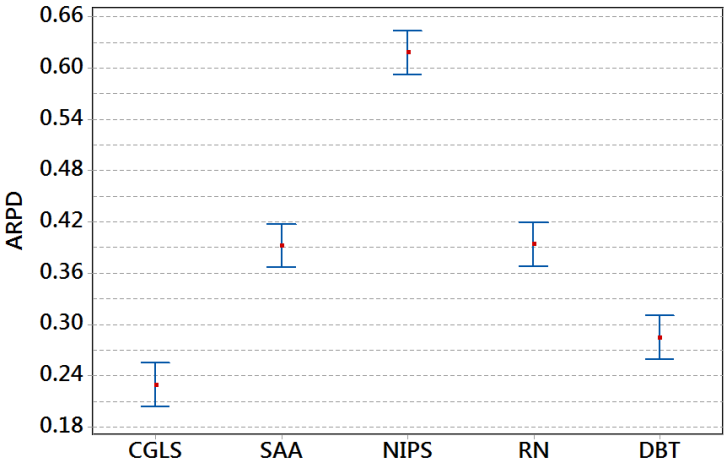


FIGURE 3.15: 95% confidence intervals with Tukey's HSD for CGLS variants.

effectiveness of the constraint aware neighbourhood generation, employing intensification of partial solution, and the new problem dependent threshold based acceptance criterion.

Comparison With Existing Algorithms

To ascertain the SS/CGLS's overall performance, we compare it with the 7 algorithms that we mentioned before. We run each algorithm with timeout ρnm milliseconds where $\rho \in \{30, 60, 90\}$. This timeout gives more time when the number of jobs and that of machines increase and thus allows us to analyse the performance of all the methods over short to very long CPU times. Nevertheless, all algorithms are tested on three benchmark sets already mentioned in Section 3.1.8 and are executed 5 times on each instance. Then, the ARPD value for each algorithm is computed using the best known solution known for that instance.

In the first experiment, the algorithms are run on the first benchmark set ([1]). The ARPDs of the algorithms for each instance group are shown in Tables 3.11-3.13. Besides, to have a better view about the ranking of the algorithms in each instance group, ranks of the algorithms based on their ARPDs are also shown in the tables; the less the ARPDs, the smaller the rank. From these tables, we can see that the proposed CGLS algorithm is much better than the other algorithms for all three different timeouts. Furthermore, to see the difference of the algorithms statistically, we also carry out an ANOVA considering algorithms as the factors. The ANOVA tables are given in Table 3.14. As shown, at a confidence level of $\alpha = 0.05$, the ARPDs of the competing

algorithms are statistically different because of p-value < 0.05. The mean plots with 95% Tukey's HSD of the compared algorithms are also reported in Figure 3.16. As we observed from this figure, the proposed CGLS algorithm statistically outperforms other competing algorithms. In addition, to see the behaviour of the algorithms for different numbers of jobs and machines, the interactions between m and n with the algorithms are shown in Figure 3.17. As can be seen from this figure, for the numbers of jobs, the proposed CGLS algorithm and IG_IJ are less influenced when $n > 50$. On the other hand, for numbers of machines, the proposed algorithm interestingly performs better when the number of machines are increased.

TABLE 3.11: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 30$) for Taillard instances [1]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
20×5	0.04 (6)	0.07 (9)	0.04 (8)	0.00 (**1**)	0.04 (5)	0.03 (3)	0.03 (4)	0.00 (**1**)	0.04 (7)
20×10	0.08 (7)	0.05 (6)	0.08 (8)	0.03 (4)	0.00 (**1**)	0.04 (5)	0.10 (9)	0.01 (2)	0.02 (3)
20×20	0.01 (8)	0.00 (7)	0.02 (9)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.04 (9)	0.00 (**1**)	0.00 (**1**)
50×5	1.12 (8)	1.07 (7)	1.37 (9)	0.80 (4)	0.99 (6)	0.68 (3)	0.66 (2)	0.81 (5)	0.29 (**1**)
50×10	1.82 (9)	1.05 (6)	1.80 (8)	0.89 (4)	1.15 (7)	0.93 (5)	0.75 (2)	0.80 (3)	0.30 (**1**)
50×20	1.69 (8)	0.86 (4)	1.97 (9)	1.03 (5)	1.52 (7)	1.03 (6)	0.77 (3)	0.68 (2)	0.41 (**1**)
100×5	2.03 (7)	1.13 (5)	2.49 (9)	1.06 (4)	2.24 (8)	1.16 (6)	0.63 (2)	0.78 (3)	0.27 (**1**)
100×10	3.11 (7)	1.25 (5)	3.29 (9)	1.14 (4)	3.25 (8)	1.81 (6)	1.08 (3)	0.76 (2)	0.38 (**1**)
100×20	4.01 (8)	1.28 (4)	4.34 (9)	1.48 (5)	2.86 (7)	2.32 (6)	0.88 (2)	1.21 (3)	0.34 (**1**)
200×5	1.62 (7)	1.11 (4)	3.11 (9)	1.25 (5)	2.52 (8)	1.58 (6)	0.74 (2)	1.07 (3)	0.55 (**1**)
200×10	2.87 (8)	1.53 (3)	3.19 (9)	1.61 (4)	2.70 (7)	1.97 (6)	0.94 (2)	1.69 (5)	0.39 (**1**)
200×20	2.65 (7)	1.66 (5)	4.24 (9)	1.38 (4)	2.95 (7)	1.67 (6)	1.01 (2)	1.03 (5)	0.31 (**1**)
500×5	3.03 (9)	1.24 (3)	2.46 (7)	1.37 (5)	2.51 (8)	1.66 (6)	0.89 (2)	1.26 (4)	0.50 (**1**)
500×10	3.43 (9)	1.44 (4)	2.67 (7)	1.78 (6)	2.67 (8)	1.57 (5)	0.95 (2)	1.38 (3)	0.44 (**1**)
500×20	4.29 (9)	1.18 (3)	3.45 (8)	1.88 (5)	2.97 (7)	2.25 (6)	1.06 (2)	1.32 (4)	0.38 (**1**)
average	2.12 (8)	0.99 (4)	2.30 (9)	1.05 (5)	1.89 (7)	1.25 (6)	0.70 (2)	0.85 (3)	0.31 (**1**)

TABLE 3.12: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 60$) for Taillard instances [1]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
20×5	0.03 (8)	0.00 (**1**)	0.04 (9)	0.00 (**1**)	0.03 (7)	0.02 (6)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)
20×10	0.07 (9)	0.00 (**1**)	0.06 (8)	0.00 (**1**)	0.00 (**1**)	0.03 (7)	0.00 (**1**)	0.01 (6)	0.00 (**1**)
20×20	0.01 (8)	0.00 (**1**)	0.02 (9)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (1)
50×5	0.87 (8)	0.85 (7)	1.07 (9)	0.62 (4)	0.75 (6)	0.52 (3)	0.52 (2)	0.68 (5)	0.23 (**1**)
50×10	1.38 (8)	0.78 (6)	1.52 (9)	0.66 (5)	0.82 (7)	0.66 (4)	0.60 (2)	0.61 (3)	0.20 (**1**)
50×20	1.30 (8)	0.70 (4)	1.72 (9)	0.82 (6)	1.08 (7)	0.73 (5)	0.60 (3)	0.56 (2)	0.29 (**1**)
100×5	1.54 (7)	0.75 (4)	1.99 (9)	0.82 (5)	1.62 (8)	0.91 (6)	0.51 (2)	0.59 (3)	0.18 (**1**)
100×10	2.30 (7)	0.90 (4)	2.86 (9)	0.95 (5)	2.39 (8)	1.25 (6)	0.85 (3)	0.64 (2)	0.29 (**1**)
100×20	3.14 (8)	1.02 (4)	3.39 (9)	1.10 (5)	2.10 (7)	1.63 (6)	0.68 (2)	0.97 (3)	0.26 (**1**)
200×5	1.29 (7)	0.87 (4)	2.47 (9)	0.99 (5)	1.95 (8)	1.25 (6)	0.61 (2)	0.86 (3)	0.42 (**1**)
200×10	2.15 (8)	1.05 (3)	2.70 (9)	1.28 (5)	1.94 (7)	1.33 (6)	0.75 (2)	1.21 (4)	0.29 (**1**)
200×20	2.09 (7)	1.22 (6)	3.39 (9)	1.09 (4)	2.17 (8)	1.22 (5)	0.80 (3)	0.74 (2)	0.20 (**1**)
500×5	2.30 (9)	1.00 (4)	2.01 (8)	1.13 (6)	1.88 (7)	1.11 (5)	0.69 (2)	0.98 (3)	0.38 (**1**)
500×10	2.64 (9)	1.12 (4)	2.21 (8)	1.37 (6)	2.04 (7)	1.23 (5)	0.74 (2)	0.99 (3)	0.36 (**1**)
500×20	3.35 (9)	0.80 (2)	3.00 (8)	1.44 (5)	2.14 (7)	1.58 (6)	0.82 (3)	1.00 (4)	0.25 (**1**)
average	1.63 (8)	0.74(4)	1.90 (9)	0.82 (5)	1.39 (7)	0.90 (6)	0.55 (2)	0.66 (3)	0.22 (**1**)

As the second experiment, the algorithms compared are tested on the 240 large VRF hard benchmarks [3]. The ARPDs and the ranks of the algorithms for each instance group are shown in Tables 3.15-3.17. In all instance groups and for each timeout, the

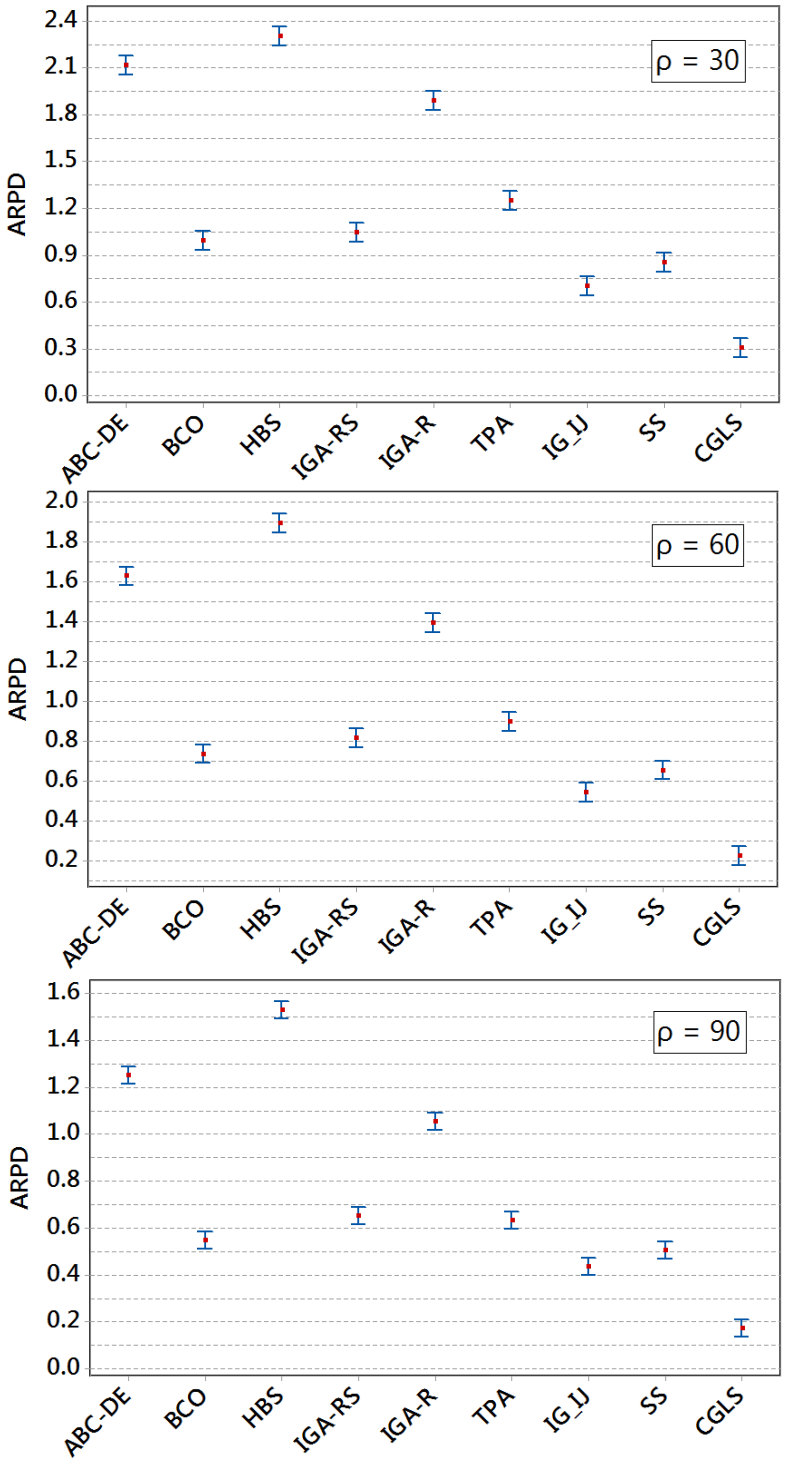


FIGURE 3.16: Means plots with 95% Tukey's HSD of the competing algorithms for the 150 Taillard instances [1] in three different timeout scenarios.

TABLE 3.13: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 90$) for Taillard instances [1]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
20×5	0.03 (8)	0.00 (**1**)	0.03 (8)	0.00 (**1**)	0.02 (7)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)
20×10	0.05 (8)	0.00 (**1**)	0.05 (8)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.01 (**1**)	0.00 (**1**)
20×20	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)	0.00 (**1**)
50×5	0.68 (8)	0.58 (6)	0.85 (9)	0.48 (4)	0.58 (6)	0.39 (2)	0.40 (3)	0.52 (5)	0.18 (**1**)
50×10	1.10 (8)	0.62 (6)	1.20 (9)	0.53 (5)	0.64 (7)	0.46 (2)	0.48 (3)	0.50 (4)	0.15 (**1**)
50×20	0.98 (8)	0.48 (3)	1.34 (9)	0.61 (6)	0.79 (7)	0.52 (5)	0.49 (4)	0.43 (2)	0.22 (**1**)
100×5	1.20 (7)	0.57 (4)	1.53 (9)	0.66 (6)	1.20 (7)	0.62 (5)	0.41 (2)	0.42 (3)	0.14 (**1**)
100×10	1.84 (7)	0.64 (3)	2.25 (9)	0.77 (5)	1.88 (8)	0.85 (6)	0.68 (4)	0.53 (2)	0.20 (**1**)
100×20	2.34 (8)	0.72 (4)	2.85 (9)	0.91 (5)	1.63 (7)	1.26 (6)	0.57 (2)	0.70 (3)	0.21 (**1**)
200×5	0.98 (7)	0.68 (4)	2.11 (9)	0.82 (5)	1.55 (8)	0.87 (6)	0.51 (2)	0.64 (3)	0.31 (**1**)
200×10	1.67 (8)	0.80 (3)	2.16(9)	1.02 (6)	1.46 (7)	0.96 (5)	0.58 (2)	0.92 (4)	0.24 (**1**)
200×20	1.58 (8)	0.92 (6)	2.85 (9)	0.89 (5)	1.55 (7)	0.82 (4)	0.63 (3)	0.61 (2)	0.14 (**1**)
500×5	1.74 (9)	0.74 (3)	1.56 (8)	0.92 (6)	1.34 (7)	0.79 (5)	0.55 (2)	0.76 (4)	0.32 (**1**)
500×10	2.11 (9)	0.86 (4)	1.87 (8)	1.02 (6)	1.62 (7)	0.86 (4)	0.58 (2)	0.72 (3)	0.28 (**1**)
500×20	2.48 (9)	0.62 (2)	2.31 (8)	1.13 (6)	1.56 (7)	1.11 (5)	0.63 (3)	0.82 (4)	0.20 (**1**)
average	1.25 (8)	0.55 (4)	1.53 (9)	0.65 (6)	1.05 (7)	0.63 (5)	0.43 (2)	0.51 (3)	0.17 (**1**)

TABLE 3.14: Statistical results of comparing algorithms

	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 30$	Algorithm	8	2787	348.425	476.87	**0.000**
	Error	6741	4925	0.731		
	Total	6749	7713			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 60$	Algorithm	8	1794	224.241	512.5	**0.000**
	Error	6741	2949	0.438		
	Total	6749	4743			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 90$	Algorithm	8	2338	292.203	762.03	**0.000**
	Error	6741	2585	0.383		
	Total	6749	4922			

proposed algorithm is much better than the other algorithms compared. It is interesting to mention that the proposed acceleration method for insert move improves those algorithms which uses that move exhaustively. For example, it has been showed that the SS algorithm is significantly better than BCO algorithm [7]. However, in this paper, we can see that for Taillard instance they are equivalent and in the VRF instances BCO statistically outperforms the SS algorithm. One of the possible reasons could be employing a different and stronger degree of local search than SS. However, note that, SS algorithm has the weakest local search among all compared algorithms although its results are still promising because of its well-designed structure.

To show the superiority of the proposed algorithm over compared ones from a statistical point of view, the ANOVA approach at confidence level of $\alpha = 0.05$ is applied where the algorithm type is considered as a factor. The results of ANOVA is given in Table 3.18. From this table, we can conclude that there is a significant difference between at least

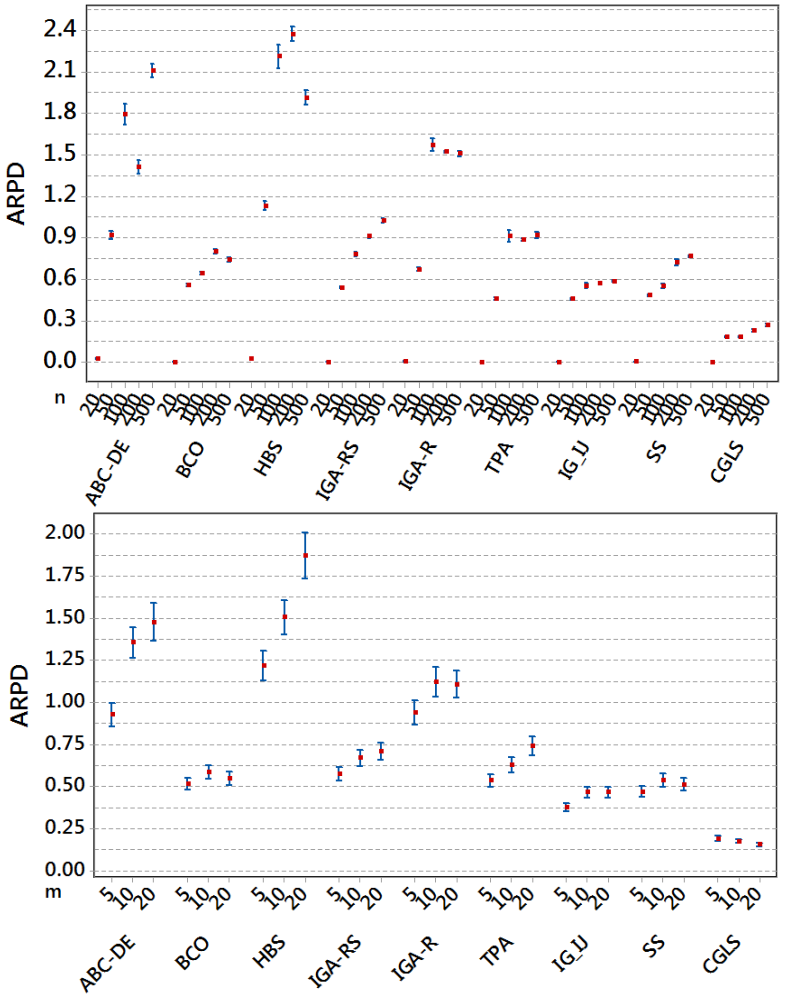


FIGURE 3.17: 95% confidence interval of the competing algorithms for the 150 Taillard instances [1] in different number of jobs and machines.

two algorithms as p-value < 0.05. Consequently, mean plots with 95% HSD interval plot of algorithms for each timeout are shown in Figure 3.18 to find out the statistically significant differences of all compared algorithms. As can be seen, the proposed CGLS algorithm significantly outperforms other algorithms in all three timeout scenarios.

As the third testbed, the algorithms are tested on the 30 very large LY instances [4]. The results are summarised in Table 3.19. As we observe, this experiment confirms the same conclusions discussed above. An ANOVA test and 95% confidence intervals are also shown in Table 3.20 and Figure 3.19 respectively. The ANOVA table says that there is a statistically significance among compared algorithms, and Figure 3.19 make it clear that this significant difference is between which algorithms. The proposed CGLS algorithm statistically outperforms other algorithms.

TABLE 3.15: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 30$) for VRF instances [3]. Best ranks in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
100×20	2.35 (7)	1.00 (3)	3.64 (9)	1.06 (4)	2.40 (8)	1.76 (6)	0.84 (2)	1.17 (5)	0.57 (**1**)
100×40	2.82 (7)	0.75 (2)	4.99 (9)	1.22 (4)	3.16 (8)	2.13 (6)	0.77 (3)	1.49 (5)	0.58 (**1**)
100×60	2.32 (7)	0.73 (2)	3.49 (8)	1.29 (4)	3.57 (9)	1.47 (5)	0.88 (3)	1.56 (6)	0.42 (**1**)
200×20	2.98 (8)	0.61 (2)	5.54 (9)	1.21 (5)	2.77 (7)	1.84 (6)	0.65 (3)	1.13 (4)	0.42 (**1**)
200×40	2.55 (6)	0.77 (2)	4.11 (9)	1.33 (4)	4.08 (8)	2.55 (7)	0.91 (3)	1.39 (5)	0.54 (**1**)
200×60	3.14 (7)	1.15 (3)	4.32 (9)	1.50 (4)	3.99 (8)	1.64 (5)	0.96 (2)	2.05 (6)	0.55 (**1**)
300×20	3.10 (8)	1.18 (3)	2.28 (7)	1.30 (4)	3.39 (9)	1.70 (6)	0.74 (2)	1.62 (5)	0.40 (**1**)
300×40	3.35 (9)	1.33 (4)	2.94 (8)	1.25 (3)	2.10 (7)	2.08 (6)	0.92 (2)	1.57 (5)	0.67 (**1**)
300×60	2.06 (6)	1.28 (3)	3.98 (9)	1.45 (4)	3.16 (8)	2.33 (7)	0.96 (2)	1.85 (5)	0.69 (**1**)
400×20	3.39 (9)	1.41 (3)	3.23 (8)	1.63 (4)	1.79 (5)	2.01 (7)	1.38 (2)	1.86 (6)	0.58 (**1**)
400×40	2.80 (8)	1.45 (3)	2.58 (6)	1.58 (4)	3.04 (9)	2.64 (7)	1.11 (2)	1.99 (5)	0.35 (**1**)
400×60	3.89 (9)	1.53 (4)	2.94 (8)	1.51 (3)	2.81 (7)	2.27 (6)	1.11 (2)	1.79 (5)	0.33 (**1**)
500×20	4.13 (8)	1.17 (3)	6.07 (9)	1.37 (4)	2.37 (7)	1.90 (6)	0.74 (2)	1.54 (5)	0.53 (**1**)
500×40	4.89 (8)	1.46 (3)	6.60 (9)	1.73 (4)	2.74 (7)	2.42 (6)	0.89 (2)	1.99 (5)	0.44 (**1**)
500×60	3.24 (7)	1.62 (3)	4.17 (9)	1.78 (4)	3.56 (8)	2.50 (6)	1.32 (2)	2.01 (5)	0.36 (**1**)
600×20	3.72 (8)	1.57 (5)	4.77 (9)	1.17 (2)	2.84 (7)	1.48 (4)	1.37 (3)	1.67 (6)	0.79 (**1**)
600×40	4.44 (8)	1.55 (4)	5.69 (9)	1.30 (3)	4.04 (7)	2.08 (5)	1.21 (2)	2.32 (6)	0.78 (**1**)
600×60	4.47 (8)	1.68 (4)	5.92 (9)	1.30 (3)	3.78 (7)	2.78 (6)	1.14 (2)	1.83 (5)	0.62 (**1**)
700×20	3.26 (8)	1.77 (3)	5.31 (9)	2.21 (4)	3.21 (7)	2.94 (6)	1.36 (2)	2.62 (5)	0.65 (**1**)
700×40	4.87 (8)	2.06 (5)	5.92 (9)	1.74 (3)	3.88 (7)	3.15 (6)	1.43 (2)	2.04 (4)	0.78 (**1**)
700×60	4.33 (8)	2.23 (4)	4.77 (9)	1.73 (3)	4.11 (7)	3.52 (6)	1.42 (2)	2.26 (5)	0.80 (**1**)
800×20	3.77 (8)	1.54 (4)	4.25 (9)	1.25 (2)	3.15 (7)	2.81 (6)	1.57 (5)	1.42 (3)	1.07 (**1**)
800×40	4.53 (7)	1.83 (5)	7.46 (9)	1.22 (2)	4.73 (8)	2.38 (6)	1.54 (3)	1.58 (4)	0.64 (**1**)
800×60	4.63 (7)	1.87 (4)	7.77 (9)	2.23 (5)	4.68 (8)	2.42 (6)	1.69 (2)	1.81 (3)	1.02 (**1**)
average	3.58 (8)	1.16 (3)	4.66 (9)	1.96 (5)	3.26 (7)	2.27 (6)	1.10 (2)	1.63 (4)	0.60 (**1**)

TABLE 3.16: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 60$) for VRF instances [3]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
100×20	1.85 (8)	0.70 (3)	2.82 (9)	0.89 (4)	1.85 (7)	1.33 (6)	0.62 (2)	0.94 (5)	0.37 (**1**)
100×40	2.09 (7)	0.58 (3)	3.59 (9)	1.03 (4)	2.24 (8)	1.64 (6)	0.57 (2)	1.15 (5)	0.37 (**1**)
100×60	1.77 (7)	0.55 (2)	2.55 (8)	1.10 (5)	2.55 (9)	1.15 (6)	0.61 (3)	1.09 (4)	0.27 (**1**)
200×20	2.23 (8)	0.43 (2)	4.07 (9)	0.96 (5)	2.08 (7)	1.43 (6)	0.47 (3)	0.84 (4)	0.28 (**1**)
200×40	1.99 (7)	0.61 (2)	2.94 (8)	1.08 (5)	3.03 (9)	1.78 (6)	0.72 (3)	0.97 (4)	0.36 (**1**)
200×60	2.43 (7)	0.80 (3)	3.22 (9)	1.31 (5)	2.89 (8)	1.15 (4)	0.75 (2)	1.43 (6)	0.37 (**1**)
300×20	2.42 (9)	0.86 (3)	1.67 (7)	1.09 (4)	2.40 (8)	1.29 (6)	0.53 (2)	1.11 (5)	0.27 (**1**)
300×40	2.55 (9)	0.95 (3)	2.13 (8)	1.02 (4)	1.65 (7)	1.45 (6)	0.66 (2)	1.17 (5)	0.46 (**1**)
300×60	1.63 (6)	0.96 (3)	2.91 (9)	1.25 (5)	2.49 (8)	1.84 (7)	0.69 (2)	1.20 (4)	0.44 (**1**)
400×20	2.51 (9)	1.10 (3)	2.51 (8)	1.31 (4)	1.33 (5)	1.53 (7)	0.98 (2)	1.40 (6)	0.40 (**1**)
400×40	2.12 (8)	1.16 (3)	2.06 (7)	1.21 (4)	2.19 (9)	1.89 (6)	0.85 (2)	1.54 (5)	0.23 (**1**)
400×60	3.07 (9)	1.18 (3)	2.11 (8)	1.29 (4)	2.06 (7)	1.61 (6)	0.81 (2)	1.40 (5)	0.24 (**1**)
500×20	3.28 (8)	0.86 (3)	4.74 (9)	1.20 (4)	1.68 (7)	1.47 (6)	0.53 (2)	1.22 (5)	0.33 (**1**)
500×40	3.71 (8)	1.07 (3)	4.75 (9)	1.40 (4)	1.96 (7)	1.74 (6)	0.62 (2)	1.41 (5)	0.28 (**1**)
500×60	2.57 (7)	1.30 (3)	3.33 (9)	1.39 (4)	2.64 (8)	1.85 (6)	0.99 (2)	1.41 (5)	0.24 (**1**)
600×20	2.88 (8)	1.14 (4)	3.79 (9)	0.98 (3)	2.13 (7)	1.16 (5)	0.97 (2)	1.32 (6)	0.51 (**1**)
600×40	3.39 (8)	1.17 (4)	4.21 (9)	1.12 (3)	2.79 (7)	1.62 (5)	0.86 (2)	1.69 (6)	0.49 (**1**)
600×60	3.34 (8)	1.21 (4)	4.73 (9)	1.10 (3)	2.61 (7)	2.00 (6)	0.81 (2)	1.25 (5)	0.39 (**1**)
700×20	2.53 (8)	1.32 (3)	3.96 (9)	1.75 (5)	2.31 (7)	2.24 (6)	0.98 (2)	1.74 (4)	0.42 (**1**)
700×40	3.61 (8)	1.56 (5)	4.55 (9)	1.48 (4)	2.87 (7)	2.20 (6)	1.07 (2)	1.41 (3)	0.50 (**1**)
700×60	3.38 (8)	1.54 (5)	3.56 (9)	1.33 (3)	3.16 (7)	2.81 (6)	1.03 (2)	1.47 (4)	0.54 (**1**)
800×20	2.93 (8)	1.16 (4)	3.10 (9)	1.04 (2)	2.48 (7)	2.13 (6)	1.21 (5)	1.11 (3)	0.70 (**1**)
800×40	3.48 (8)	1.36 (5)	5.33 (9)	0.97 (2)	3.40 (7)	1.80 (6)	1.12 (3)	1.24 (4)	0.45 (**1**)
800×60	3.56 (8)	1.46 (4)	5.59 (9)	1.73 (5)	3.39 (7)	1.78 (6)	1.17 (2)	1.24 (3)	0.70 (**1**)
average	2.75 (8)	0.95 (3)	3.53 (9)	1.39 (5)	2.41 (7)	1.67 (6)	0.82 (2)	1.22 (4)	0.40 (**1**)

TABLE 3.17: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 90$) for VRF instances [3]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
100×20	1.46 (8)	0.52 (3)	2.14 (9)	0.76 (5)	1.32 (7)	0.96 (6)	0.48 (2)	0.69 (4)	0.26 (**1**)
100×40	1.63 (8)	0.42 (3)	2.64 (9)	0.85 (5)	1.58 (7)	1.23 (6)	0.39 (2)	0.79 (4)	0.23 (**1**)
100×60	1.33 (7)	0.41 (2)	2.04 (9)	0.93 (6)	1.76 (8)	0.89 (5)	0.43 (3)	0.80 (4)	0.17 (**1**)
200×20	1.78 (8)	0.32 (2)	3.11 (9)	0.79 (5)	1.52 (7)	1.10 (6)	0.36 (3)	0.64 (4)	0.19 (**1**)
200×40	1.58 (7)	0.45 (2)	2.16 (9)	0.83 (5)	2.10 (8)	1.32 (6)	0.51 (3)	0.77 (4)	0.24 (**1**)
200×60	1.90 (7)	0.61 (3)	2.48 (9)	1.03 (6)	2.11 (8)	0.79 (4)	0.58 (2)	0.96 (5)	0.26 (**1**)
300×20	1.85 (9)	0.66 (3)	1.24 (7)	0.86 (5)	1.68 (8)	0.96 (6)	0.42 (2)	0.79 (4)	0.18 (**1**)
300×40	1.95 (9)	0.71 (3)	1.64 (8)	0.89 (5)	1.29 (7)	1.04 (6)	0.51 (2)	0.83 (4)	0.31 (**1**)
300×60	1.21 (6)	0.72 (3)	2.17 (9)	0.96 (4)	1.79 (8)	1.36 (7)	0.49 (2)	0.96 (4)	0.30 (**1**)
400×20	2.01 (9)	0.82 (3)	1.83 (8)	1.07 (6)	0.99 (5)	1.11 (7)	0.72 (2)	0.96 (4)	0.27 (**1**)
400×40	1.67 (9)	0.89 (3)	1.55 (7)	1.02 (4)	1.62 (8)	1.32 (6)	0.61 (2)	1.02 (4)	0.16 (**1**)
400×60	2.34 (9)	0.91 (3)	1.60 (7)	1.11 (5)	1.65 (8)	1.25 (6)	0.58 (2)	0.96 (4)	0.17 (**1**)
500×20	2.43 (8)	0.60 (3)	3.62 (9)	0.98 (5)	1.22 (7)	1.17 (6)	0.42 (2)	0.85 (4)	0.22 (**1**)
500×40	2.85 (8)	0.75 (3)	3.49 (9)	1.10 (5)	1.42 (7)	1.26 (6)	0.48 (2)	0.93 (4)	0.19 (**1**)
500×60	1.95 (7)	0.96 (3)	2.45 (9)	1.21 (5)	2.11 (8)	1.34 (6)	0.72 (2)	1.04 (4)	0.15 (**1**)
600×20	2.27 (8)	0.79 (2)	2.89 (9)	0.79 (2)	1.64 (7)	0.89 (5)	0.82 (4)	0.89 (5)	0.34 (**1**)
600×40	2.55 (8)	0.84 (3)	3.12 (9)	0.92 (4)	2.11 (7)	1.22 (6)	0.62 (2)	1.12 (5)	0.33 (**1**)
600×60	2.67 (8)	0.93 (5)	3.64 (9)	0.85 (4)	1.85 (7)	1.55 (6)	0.58 (2)	0.83 (3)	0.25 (**1**)
700×20	1.96 (8)	1.05 (3)	2.85 (9)	1.38 (5)	1.79 (7)	1.58 (6)	0.78 (2)	1.14 (4)	0.27 (**1**)
700×40	2.82 (8)	1.13 (4)	3.64 (9)	1.21 (5)	2.21 (7)	1.68 (6)	0.84 (2)	1.02 (3)	0.33 (**1**)
700×60	2.64 (8)	1.22 (5)	2.78 (9)	1.13 (4)	2.31 (7)	2.01 (6)	0.79 (2)	0.96 (3)	0.35 (**1**)
800×20	2.34 (8)	0.86 (4)	2.48 (9)	0.82 (2)	1.95 (7)	1.48 (6)	0.91 (5)	0.82 (2)	0.49 (**1**)
800×40	2.64 (8)	0.97 (5)	3.86 (9)	0.84 (3)	2.43 (7)	1.25 (6)	0.77 (2)	0.88 (4)	0.31 (**1**)
800×60	2.68 (8)	1.14 (4)	4.11 (9)	1.34 (5)	2.65 (7)	1.37 (6)	0.82 (2)	0.94 (3)	0.44 (**1**)
average	2.10 (8)	0.78 (3)	2.65 (9)	0.99 (5)	1.80 (7)	1.26 (6)	0.61 (2)	0.90 (4)	0.27 (**1**)

TABLE 3.18: ANOVA results of the compared algorithms for VRF instances.

	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 30$	Algorithm	8	17340	2167.49	4749.90	**0.000**
	Error	10719	4924	0.46		
	Total	10799	22264			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 60$	Algorithm	8	10006	1250.75	5009.92	**0.000**
	Error	10719	2694	0.25		
	Total	10799	12700			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 90$	Algorithm	8	5724	715.545	5266.36	**0.000**
	Error	10719	1466	0.136		
	Total	10799	7191			

TABLE 3.19: ARPDs and the ranks (in brackets) of the algorithms for LY instances [4]. Best values in bold.

n×m	ABC-DE	BCO	HBS	IGA-RS	IGA-R	TPA	IG_IJ	SS	CGLS
					$\rho = 30$				
1000×20	3.33 (8)	1.44 (4)	3.45 (9)	1.34 (3)	2.61 (7)	1.77 (6)	1.24 (2)	1.60 (5)	0.83 (**1**)
1500×20	3.16 (9)	1.51 (4)	2.96 (8)	1.30 (3)	2.83 (7)	1.56 (6)	1.21 (2)	1.55 (5)	0.65 (**1**)
2000×20	3.73 (9)	1.52 (3)	3.66 (8)	1.73 (5)	2.89 (6)	1.67 (4)	1.49 (2)	2.06 (7)	0.96 (**1**)
					$\rho = 60$				
1000×20	3.02 (8)	1.05 (3)	3.28 (9)	1.11 (4)	2.36 (7)	1.46 (6)	0.97 (2)	1.27 (5)	0.53 (**1**)
1500×20	2.96 (9)	1.28 (4)	2.77 (8)	1.13 (3)	2.47 (7)	1.28 (4)	1.02 (2)	1.37 (6)	0.46 (**1**)
2000×20	3.42 (9)	1.39 (3)	3.39 (8)	1.57 (5)	2.72 (7)	1.41 (4)	1.30 (2)	1.64 (6)	0.73 (**1**)
					$\rho = 90$				
1000×20	2.34 (8)	0.86 (3)	2.93 (9)	0.96 (5)	2.11 (7)	1.18 (6)	0.78 (2)	0.93 (4)	0.39 (**1**)
1500×20	2.62 (9)	1.05 (4)	2.58 (8)	0.92 (3)	2.36 (7)	1.10 (5)	0.89 (2)	1.16 (6)	0.46 (**1**)
2000×20	2.87 (8)	1.27 (3)	3.11 (9)	1.30 (5)	2.58 (7)	1.29 (4)	1.11 (2)	1.37 (6)	0.62 (**1**)

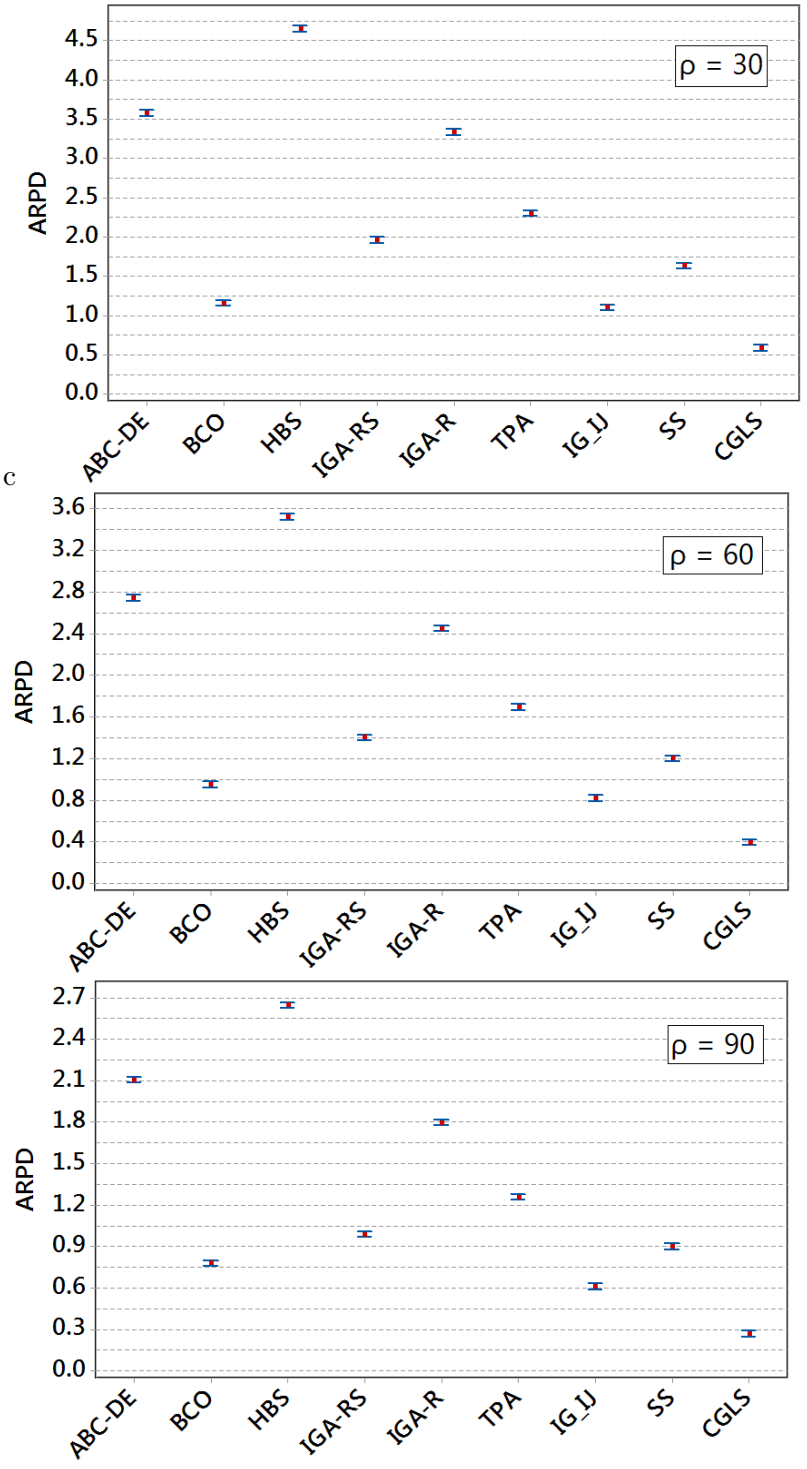
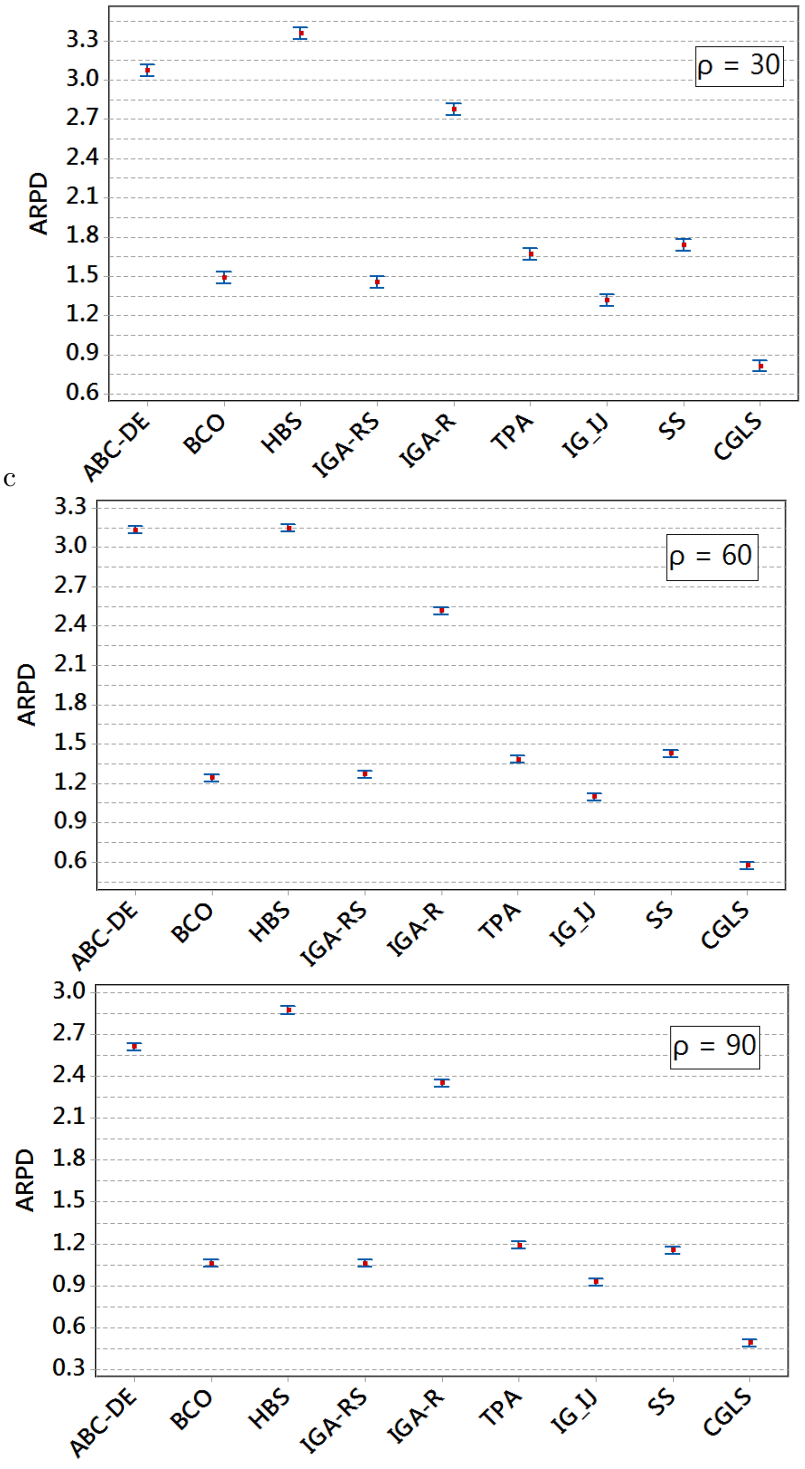


FIGURE 3.18: Means plots with 95% Tukey's HSD of the competing algorithms for the 240 VRF instances in three different timeout scenarios.



c

FIGURE 3.19: Means plots with 95% Tukey's HSD of the competing algorithms for the 30 LY instances in three different timeout scenarios.

TABLE 3.20: ANOVA results of the compared algorithms for LY instances.

	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 30$	Algorithm	8	930	116.252	1539.44	**0.000**
	Error	1341	101.3	0.076		
	Total	1349	1031.3			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 60$	Algorithm	8	1048.92	131.115	4393.06	**0.000**
	Error	1341	40.02	0.03		
	Total	1349	1088.94			
	Source	DF	Adj SS	Adj MS	F-Value	P-Value
$\rho = 90$	Algorithm	8	868.18	108.523	3793.77	**0.000**
	Error	1341	38.36	0.029		
	Total	1349	906.54			

3.1.9 Best Known Solutions

In this section, the best know solution found by the algorithms compared for all three testbeds, 150 Taillard instances [1], 240 large VRF instances [3], and 30 very large LY instances [4] are given in Tables 3.21-3.23. These tables are used in this paper to calculate the RPDs of the results, and can also be used by future researchers.

TABLE 3.21: Best known solutions of 150 Taillard instances.

Prob	Best	Prob	Best	Prob	Best	Prob	Best	Prob	Best
20×5		20×10		20×20		50×5		50×10	
1	2220	11	2690	21	2981	31	5186	41	5969
2	1394	12	2708	22	3125	32	3085	42	5816
3	1963	13	2319	23	3263	33	5297	43	5474
4	2378	14	2377	24	3232	34	5365	44	5798
5	2029	15	2440	25	3137	35	5267	45	5287
6	2247	16	2325	26	3161	36	5409	46	5341
7	2247	17	2438	27	3276	37	4807	47	4999
8	2114	18	2432	28	3051	38	5115	48	5421
9	2195	19	2475	29	3245	39	4759	49	5788
10	1851	20	2620	30	2852	40	5302	50	5928
50×20		100×5		100×10		100×20		200×10	
51	6846	61	10744	71	11067	81	11594	91	21202
52	6200	62	10613	72	10515	82	11725	92	20636
53	6425	63	10122	73	10599	83	12270	93	21489
54	6757	64	9943	74	12412	84	12238	94	21813
55	6244	65	10120	75	11081	85	11126	95	20802
56	6659	66	9664	76	11437	86	12128	96	20379
57	5910	67	10574	77	10341	87	12134	97	22174
58	6376	68	10033	78	11366	88	11982	98	22067
59	6463	69	10226	79	11415	89	11343	99	23041

60	6576	70	10512	80	11772	90	12207	100	21462
200×20		500×20		200×5		500×5		500×10	
101	24108	111	55906	121	21335	131	50942	141	52641
102	22197	112	55754	122	20372	132	51747	142	52706
103	24414	113	56527	123	20710	133	52824	143	53487
104	23078	114	54579	124	20481	134	51207	144	55117
105	23526	115	57648	125	20885	135	52001	145	53912
106	24916	116	61801	126	20125	136	52248	146	55560
107	22208	117	52627	127	20523	137	49534	147	50523
108	24411	118	53442	128	20788	138	52214	148	53115
109	24892	119	56367	129	19933	139	52345	149	52348
110	22674	120	54830	130	20116	140	50659	150	53696

TABLE 3.22: Best known solutions of 240 VRF instances.

Prob	Best	Prob	Best	Prob	Best	Prob	Best	Prob	Best	Prob	Best
100×20		100×40		100×60		200×20		200×40		200×60	
1	11532	11	14248	21	14931	31	23288	41	25595	51	28389
2	11880	12	13748	22	14753	32	21853	42	25999	52	27908
3	11598	13	13321	23	14053	33	23765	43	23764	53	28374
4	12710	14	12801	24	15104	34	23604	44	26002	54	27116
5	12090	15	13936	25	15177	35	24605	45	24479	55	26783
6	11540	16	13957	26	15842	36	21020	46	25768	56	28009
7	13134	17	13691	27	15086	37	23078	47	26247	57	28460
8	11914	18	14465	28	14789	38	22826	48	25796	58	26902
9	12502	19	13442	29	15155	39	22376	49	25031	59	27000
10	12354	20	14014	30	15462	40	24546	50	24379	60	28090
300×20		300×40		300×60		400×20		400×40		400×60	
61	34570	71	37951	81	40116	91	45686	101	51027	111	51707
62	35552	72	37925	82	40990	92	46466	102	50807	112	53904
63	35885	73	38118	83	40321	93	45835	103	51706	113	52668
64	33365	74	38830	84	39105	94	44684	104	51608	114	51202
65	34405	75	37802	85	40178	95	47090	105	47029	115	54010
66	35364	76	38209	86	39528	96	45989	106	50023	116	52020
67	31615	77	37506	87	41004	97	42711	107	50952	117	53011
68	33984	78	35774	88	39910	98	45719	108	46427	118	50619
69	31611	79	37052	89	38983	99	46916	109	49272	119	51693
70	35965	80	35790	90	40719	100	41348	110	49073	120	52850
500×20		500×40		500×60		600×20		600×40		600×60	
121	56889	131	64545	141	66504	151	64290	161	76707	171	76558
122	51362	132	62560	142	62845	152	63528	162	72401	172	76776
123	54150	133	59371	143	66911	153	68026	163	72005	173	76729
124	56694	134	63552	144	65295	154	72184	164	75669	174	76394
125	56691	135	57904	145	64130	155	68108	165	67496	175	77561

126	60221	136	60495	146	65140	156	67238	166	75392	176	79568
127	58926	137	61975	147	63615	157	66889	167	73573	177	75600
128	51967	138	63125	148	62369	158	65995	168	74642	178	75621
129	58353	139	63301	149	62717	159	70086	169	75436	179	74216
130	54166	140	62611	150	65999	160	64111	170	70475	180	76957
700×20		700×40		700×60		800×20		800×40		800×60	
181	80053	191	84209	201	89722	211	90309	221	95860	231	102538
182	84602	192	86729	202	85358	212	87859	222	98737	232	103878
183	80244	193	87724	203	89193	213	91516	223	97293	233	101955
184	77349	194	83504	204	92050	214	89044	224	99602	234	99191
185	79347	195	79359	205	85939	215	96025	225	99359	235	104496
186	71533	196	82342	206	86305	216	89004	226	98369	236	102625
187	78791	197	82459	207	89261	217	85480	227	91480	237	104190
188	79429	198	87583	208	89919	218	88461	228	95941	238	93776
189	81741	199	84930	209	91176	219	91556	229	98685	239	100628
190	74118	200	88198	210	92562	220	89228	230	99436	240	102425

TABLE 3.23: Best known solutions of 30 LY instances.

Prob	Best	Prob	Best	Prob	Best
1000×20		1500×20		2000×20	
1	108819	11	163861	21	234893
2	113089	12	172611	22	228469
3	122248	13	177254	23	216154
4	109186	14	167420	24	228953
5	116288	15	162364	25	210197
6	105020	16	174927	26	219247
7	111830	17	180641	27	218215
8	106382	18	172905	28	221777
9	121781	19	163867	29	237572
10	110031	20	163882	30	225511

3.2 Blocking Flowshops with Setup Times

In this section, we discuss about mixed blocking permutation flowshop scheduling problem (MBPFSP) with sequence-dependent setup times (SDST) constraints. We first formulate the problem and propose the methods in order to solve this problem. The formulation of problem and proposed algorithms are similar to that proposed in Section 3.1 except using SDST constraints besides MBPFSP.

A PFSP with *sequence dependent setup time* (SDST) is one of the realistic variants. Setup times are needed for those operations that must be performed on machines before starting processing of a job e.g., cleaning, fixing, and adjustments. Clearly, the machine

setup times here are not considered to be part of the job processing times. Moreover, the setup time of a machine depends not only on the job to be processed next but also does so on the job that is just processed by the same machine. Setup operations could be performed just before starting processing a job, but are typically performed immediately after the previous job is processed. Assume $[k]$ denotes the job at position k in the permutation π of n jobs. In a PFSP-SDST, $\sigma^i_{[k]}$ denotes the setup time of a machine i to process job $[k]$ after processing job $[k-1]$; there is no setup time for the first job. SDSTs are important in the context of flowshops since these are found in many industries, e.g., the paper cutting industry and the paint industry [110].

Despite existence of several PFSP variants, there are still significant modelling gaps when real-world applications are considered. In this research, we propose a new variant of PFSP named PFSP-BS that considers both blocking constraints and sequence-dependent setup times. In the proposed model, as in MBPFSP, we assume that there is a mix of blocking constraints (RSb, RCb, and RCb*) in the same problem instance. Also, as in PFSP-SDST, we assume SDSTs are required in the proposed model and are scheduled immediately after processing the previous jobs. However, when we consider both blocking constraints and setup times, two different situations could arise: when a machine is blocked by a job, it might or might not be possible to set up for the next job. We call *overlapable* setup operations when a machine can be set up for the next job even if it is blocked by a job. We call *non-overlapable* setup operations when a machine cannot be set up for the next job when it is blocked by a job. Examples of overlapable setup operations include entering some programs in the machine while that of non-overlapable setup operations include cleaning a machine which needs the machine to be free. In the proposed PFSP-BS, both overlapable and non-overlapable setup operations are allowed in the same problem instance in a mixed setting. Moreover, we consider makespan minimisation as our objective. Our motivation comes from real-life applications of this PFSP variant in the cider industry.

The cider production process comprises seven stages: stocking, washing, sorting, pressing, fermentation, filtration, and bottling [7]. First, apples of a job must stay in the stock until the washing machine is available (RSb). Then, apples also wait in sorting machine until apples of the previous job leave the pressing machine (RCb). Later, juice stays in the fermentation stage until filtration is finished (RCb*). Besides blocking constraints, in this production line, there are both overlapable and non-overlapable SDSTs as well. In

the pressing stage, apples go to the pressing machines only after machines are made free from the pressed apples of the previous jobs and then cleaned (non-overlapable SDST). However, the fermentation process needs yeast nutrients and sweeteners to be added to the juice. So, the required SDST can be started on the fermentation machine as soon as the processing of the previous job is finished regardless of being blocked or not. To summarise, the PFSP-BS is a realistic problem that, to the best of our knowledge, has not been studied before and hence draws our attention.

3.2.1 Related Work

PFSP-BS has not been studied yet, although it is realistic and it has applications in the industry. The closely related known problems are MBPFSP [12] and PFSP-SDST [40]. MBPFSP allows a mix of RSb, RCb, and RCb* constraints but no setup times. PFSP-SDST allows SDSTs but no blocking constraints. PFSP-BS generalises over both of these variants.

For MBPFSP, a genetic algorithm (GA) was proposed [12]. It was later outperformed by a scatter search (SS) algorithm [7]. The SS algorithm was further outperformed by a constraint-guided local search (CGLS) algorithm [33]. For BPFSP (RSb-PFSP), an iterated greedy (IG) algorithm, referred to as IG_R here, was presented [76]. Later, a modified fruit fly optimisation (MFFO) algorithm came for the same BPFSP [111] and lately also came another IG variant [39], referred to by IG_IJ here. For PFSP-SDST, an IG algorithm was presented [40] and then an artificial bee colony (ABC) algorithm [112].

A mixed integer linear programming model and a branch-bound method were applied on small instances of RSb-PFSP with non-overlapable SDSTs [113]. A water wave optimisation (WWO) algorithm also came for the same problem [17].

3.2.2 Preliminaries

At any time, each job in a PFSP variant can be processed by one machine at most and each machine can process one job at most. When a machine starts processing a job, it cannot be stopped.

PFSP Blocking Constraints

There is a buffer with unlimited capacity before each machine in a PFSP. This scenario is known as Without Blocking (Wb). However, in real-life factories, there may not be any intermediate buffer; which essentially causes a blocking constraint. When a blocking situation arises, a job, even after its processing is finished, stays at the current machine since the following machine is not available.

Three types of blocking has been reported in real-life situations. Under the classical Release when Starting Blocking (RSb) constraint, job $[k]$ blocks machine i which cannot start processing job $[k+1]$ because machine $i+1$ is free to process job $[k]$. Under the special Release when Completing Blocking (RCb) constraint, machine i waits before processing job $[k+1]$ since job $[k]$ is not finished by and has not left machine $i+1$. Under Release when Completing Blocking* (RCb*) constraint, machine i waits before processing job $[k+1]$ since job $[k]$ is not finished by machine $i+1$. Only one type of blocking constraint is allowed in a BPFSP while more than one type is allowed in a MBPFSP.

PFSP SDST Constraints

Setup operations (e.g. cleaning, fixing, and adjusting) are those that must be performed on machines before starting processing a job and setup time is time needed to perform those operations. In PFSP, all setup times are part of the processing times. However, in many scheduling applications, setup times must be considered separately to eliminate time wastage and to boost resource utilisation [110]. Various types of setup time are possible but sequence dependent setup time (SDST) is the most important one since it is found in 70% industries [114], including the paper cutting and the paint industry [110]. Under SDST constraints, each machine i after processing job $[k-1]=j$ and before processing job $[k]=\bar{j}$ requires sequence dependent setup time $\sigma^i_{[k]} = s^i_{j\bar{j}}$, where $s^i_{j\bar{j}}$ is given as input for each $j \neq \bar{j}$. Note that $\sigma^i_{[k]}$ is neither part of $p^i_{[k-1]}$ nor of $p^i_{[k]}$, since it depends on two successive jobs $[k-1]$ and $[k]$ in the permutation.

3.2.3 Proposed PFSP-BS

We describe the formulas needed to compute the makespan of a given PFSP-BS. Because of the reversibility property of flowshops [72], the makespan of a permutation π can be calculated passing the permutation from the first job through to the last job (forward calculation) or from the last job through to the first job (backward calculation).

Forward Makespan Calculation

Assume $S^i_{[k]}$ be the starting time and $C^i_{[k]}$ be the completion time of job $[k]$ at machine i. We use (3.25)–(3.36) to calculate the makespan $C(\pi)$ of a permutation π for an PFSP-BS. In (3.25), we define $S^i_{[k]} = 0$ and $C^i_{[k]} = 0$ whenever $i < 1$ or $i > m$ or $k < 1$ or $k > n$. Assume B_i is the blocking constraint after machines i and before machine $i+1$. Also, assume O_i denotes overlapable (O), or non-overlapable (N) SDST for a machine i.

$$S^i_{[k]} = 0, C^i_{[k]} = 0 \qquad k < 1 \vee k > n \vee i < 1 \vee i > m \tag{3.25}$$

$$C^i_{[k]} = S^i_{[k]} + p^i_{[k]} \qquad 1 \leq k \leq n,\ 1 \leq i \leq m \tag{3.26}$$

For $1 \leq k \leq n$ and $O_i = \mathsf{N}$,

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, S^{i+2}_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-2, B_i = \mathsf{RCb} \tag{3.27}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, C^{i+1}_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{RCb*} \tag{3.28}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, S^{i+1}_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{RSb} \tag{3.29}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, C^{i}_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{Wb} \tag{3.30}$$

For $1 \leq k \leq n$ and $O_i = \mathsf{O}$,

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, S^{i+2}_{[k-1]}, C^i_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-2, B_i = \mathsf{RCb} \tag{3.31}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, C^{i+1}_{[k-1]}, C^i_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{RCb*} \tag{3.32}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, S^{i+1}_{[k-1]}, C^i_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{RSb} \tag{3.33}$$

$$S^i_{[k]} = \max(C^{i-1}_{[k]}, C^i_{[k-1]} + \sigma^i_{[k]}) \qquad i \leq m-1, B_i = \mathsf{Wb} \tag{3.34}$$

$$\text{For } 1 \leq k \leq n, S^m_{[k]} = \max(C^{m-1}_{[k]}, C^m_{[k-1]} + \sigma^m_{[k]}) \tag{3.35}$$

$$\text{Finally to compute the makespan}, C(\pi) = C^m_{[n]} \tag{3.36}$$

Just like in PFSP, for a given permutation π in PFSP-BS, the computational complexity of calculating makespan $C(\pi)$ from (3.25)–(3.36) is $\mathcal{O}(mn)$. The aim of PFSP-BS with makespan criterion as shown in (3.37) is to find a permutation π_* with $C(\pi_*)$ being the smallest over each π.

$$C(\pi_*) \leq C(\pi) \qquad \forall \pi \tag{3.37}$$

Backward Makespan Calculation

$C(\pi)$ can be computed from (3.25)–(3.36) by iterating from 1 to n for position k and for each job $[k]$, iterating from 1 to m for machine i. Thus, each job at each machine is scheduled at the earliest time possible. We will now find the latest possible time each job can be scheduled at each machine, but the makespan of the whole schedule will remain the same. For this, we will calculate the schedule from job $[n]$ to [1], and for each job $[k]$

from machine m to 1. Assume $T^i_{[k]}$ denotes the time when job $[k]$ can be scheduled in the backward calculation; for convenience, time axis is assumed to be in the backward direction starting from $C(\pi)$. To deal with the case of $B_{i-2} = \mathsf{RCb}$, we use an auxiliary variable $R^i_{[k]}$ here.

$$T^i_{[k]} = 0, p^i_{[k]} = 0, \sigma^i_{[k+1]} = 0, R^i_{[k]} = 0 k < 1 \vee k > n \vee i < 1 \vee i > m \tag{3.38}$$

For $i \geq 2$, $n \geq k$, $O_{i-1} = \mathsf{N}$,

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, T^{i-1}_{[k+1]} + \sigma^{i-1}_{[k+1]}, R^i_{[k]}) B_{i-1} = \mathsf{RSb} \tag{3.39}$$

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, T^{i-1}_{[k+1]} + p^i_{[k]} + \sigma^{i-1}_{[k+1]}, R^i_{[k]}) \ B_{i-1} = \mathsf{RCb*} \tag{3.40}$$

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, R^i_{[k]}) B_{i-1} = \mathsf{RCb/Wb} \tag{3.41}$$

For $i \geq 2$, $n \geq k$, $O_{i-1} = \mathsf{O}$,

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, T^{i-1}_{[k+1]}, R^i_{[k]}) B_{i-1} = \mathsf{RSb} \tag{3.42}$$

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, T^{i-1}_{[k+1]} + p^i_{[k]}, R^i_{[k]}) B_{i-1} = \mathsf{RCb*} \tag{3.43}$$

$$T^i_{[k]} = \max(T^{i+1}_{[k]} + p^i_{[k]}, T^i_{[k+1]} + p^i_{[k]} + \sigma^i_{[k+1]}, R^i_{[k]}) B_{i-1} = \mathsf{RCb/Wb} \tag{3.44}$$

$$T^1_{[k]} = \max(T^2_{[k]}, T^1_{[k+1]} + \sigma^i_{[k+1]}) + p^1_{[k]} n \geq k \geq 1 \tag{3.45}$$

For $i \geq 2$, $n \geq k$, $B_{i-2} = \mathsf{RCb}$,

$$R^i_{[k]} = \begin{cases} T^{i-2}_{[k+1]} + \sigma^{i-2}_{[k+1]} & \text{if} O_{i-2} = \mathsf{N} \\ T^{i-2}_{[k+1]} & \text{if } O_{i-2} = \mathsf{O} \\ 0 & \text{otherwise} \end{cases} \tag{3.46}$$

Figure 3.20 shows forward and backward calculations of an example PFSP-BS schedule with 4 jobs and 5 machines. Notice that two calculation may produce different starting and completion times for some jobs.

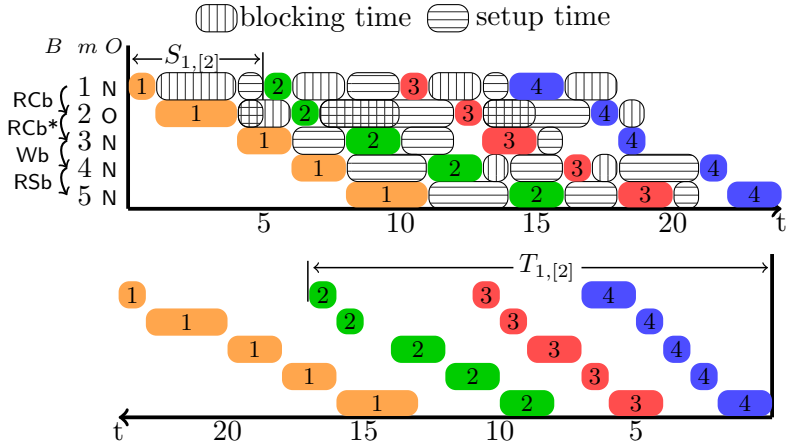


FIGURE 3.20: An example showing computation of a PFSP-BS makespan by (top) forward and (bottom) backward calculations

Lemma 3.6. *Makespan minimisation of PFSP-BS is NP-hard.*

Proof. Consider the particular instance where SDSTs are all set to zero. Thus, PFSP-BS is transformed to an MBPFSP that is known to be NP-hard [33]. □

3.2.4 Proposed Accelerated Makespan Computation

Neighbourhood operators using insertions are very widely used in PFSP and variants. As mentioned before, computing $C(\pi)$ involves $O(mn)$ operations. So insert-neighbourhood(π, k) has time complexity $\mathcal{O}(mn^2)$ and insert-neighbourhood(π) has $\mathcal{O}(mn^3)$.

Therefore, using these two neighbourhoods are very costly particularly in a large problem. Similar to the acceleration method for PFSP [71], we develop the same for PFSP-BS to compute $C(\underline{\pi})$ efficiently for each $\underline{\pi}$ in insert-neighbourhood(π, k) and insert-neighbourhood(π). The accelerated computation essentially utilises the reversibility property of the PFSP-BS.

Computing Makespan Incrementally

Given a permutation π, assume $S^i_{[k]}$, $C^i_{[k]}$, and $T^i_{[k]}$ are given for each machine i and for each job $[k]$. Assume $[\bar{k}] = j$ in π and an application of remove$(\pi, \bar{k})$ has produced a permutation $\bar{\pi}$ of $n-1$ jobs. Also, assume application of insert$(\bar{\pi}, j, \underline{k})$ has produced $\underline{\pi}$ of n jobs. We now show how to incrementally calculate $\bar{S}^i_{[k]}$, $\bar{C}^i_{[k]}$, and $\bar{T}^i_{[k]}$ for permutation $\bar{\pi}$, and $\underline{S}^i_{[k]}$, $\underline{C}^i_{[k]}$, $\underline{T}^i_{[k]}$, and hence $C(\underline{\pi})$ for permutation $\underline{\pi}$.

Computing $\bar{S}$, $\bar{C}$, and $\bar{T}$ for $\bar{\pi}$ incrementally: Notice that $\bar{S}^i_{[k]} = S^i_{[k]}$ and $\bar{C}^i_{[k]} = C^i_{[k]}$ for each i and for each $k < \bar{k}$. Compute $\bar{S}^i_{[k]}$ and $\bar{C}^i_{[k]}$ for $k \geq \bar{k}$ for each i using (3.25)–(3.36). Similarly $\bar{T}^i_{[k]} = T^i_{[k+1]}$ for each i and for each $k \geq \bar{k}$. Compute $\bar{T}^i_{[k]}$ for $k < \bar{k}$ for each i using (3.38)–(3.46).

Computing $\underline{S}$, $\underline{C}$, and $\underline{T}$ for $\underline{\pi}$ incrementally: Notice that $\underline{S}^i_{[k]} = \bar{S}^i_{[k]}$ and $\underline{C}^i_{[k]} = \bar{C}^i_{[k]}$ for each i and for each $k < \underline{k}$. Compute $\underline{S}^i_{[\underline{k}]}$ and $\underline{C}^i_{[\underline{k}]}$ for each i using (3.25)–(3.36), and $\underline{T}^i_{[k+1]} = \bar{T}^i_{[k]}$ for each i and each $k \geq \underline{k}$.

Computing makespan of $\underline{\pi}$ incrementally: To compute $C(\underline{\pi})$, we need the time point $\underline{L}^i_{[\underline{k}]}$ when job $[\underline{k}]$ in $C(\underline{\pi})$ will leave machine i and the time period $\underline{T}^i_{[\underline{k}]}$ needed to process all the jobs after position $\underline{k}$. We calculate $\underline{L}^i_{[\underline{k}]}$ in (3.47)–(3.54) below.

$$\underline{L}^i_{[\underline{k}]} = \underline{S}^{i+2}_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]} \qquad B_i = \mathsf{RCb}, O_i = \mathsf{N} \quad (3.47)$$

$$\underline{L}^i_{[\underline{k}]} = \underline{S}^{i+1}_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]} \qquad B_i = \mathsf{RSb}, O_i = \mathsf{N} \quad (3.48)$$

$$\underline{L}^i_{[\underline{k}]} = \underline{C}^{i+1}_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]} \qquad B_i = \mathsf{RCb*}, O_i = \mathsf{N} \quad (3.49)$$

$$\underline{L}^i_{[\underline{k}]} = \underline{C}^i_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]} \qquad B_i = \mathsf{Wb}, O_i = \mathsf{N} \quad (3.50)$$

$$\underline{L}^i_{[\underline{k}]} = \max(\underline{S}^{i+2}_{[\underline{k}]}, \underline{C}^i_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]}) \qquad B_i = \mathsf{RCb}, O_i = \mathsf{O} \quad (3.51)$$

$$\underline{L}^i_{[\underline{k}]} = \max(\underline{S}^{i+1}_{[\underline{k}]}, \underline{C}^i_{[\underline{k}]} + \sigma^i_{\underline{k}+1]}) \qquad B_i = \mathsf{RSb}, O_i = \mathsf{O} \quad (3.52)$$

$$\underline{L}^i_{[\underline{k}]} = \max(\underline{C}^{i+1}_{[\underline{k}]}, \underline{C}^i_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]}) \qquad B_i = \mathsf{RCb}^*, O_i = \mathsf{O} \quad (3.53)$$

$$\underline{L}^i_{[\underline{k}]} = \underline{C}^i_{[\underline{k}]} + \sigma^i_{[\underline{k}+1]} \qquad B_i = \mathsf{Wb}, O_i = \mathsf{O} \quad (3.54)$$

Given all $\underline{L}^i_{[\underline{k}]}$ and $\underline{T}^i_{[\underline{k}]}$, in (3.55), $C(\underline{\pi})$ can now be computed.

$$C(\underline{\pi}) = \max_{i=1}^{m}(\underline{L}^i_{[\underline{k}]} + \underline{T}^i_{[\underline{k}]}) \quad (3.55)$$

Evaluating Insertion Neighbourhood Incrementally

We need each permutation $\underline{\pi}$ in $\mathsf{insert\text{-}neighbourhood}(\pi, \bar{k})$. For this, we use $\mathsf{remove}(\pi, \bar{k})$ once to obtain $\bar{\pi}$ and then use $\mathsf{insert}(\bar{\pi}, j, \underline{k})$ for each $\underline{k}$ to obtain $\underline{\pi}$. Now for $\bar{\pi}$ calculate $\bar{S}^i_{[k]}$ and $\bar{C}^i_{[k]}$ where $k = 1, \ldots, n-1$ and $i = 1, \ldots, m$. Then, compute $\bar{T}^i_{[k]}$ for $i = m, \ldots, 1$ and $k = n-1, \ldots, 1$ for $\bar{\pi}$. Next, for each $\underline{\pi}$, compute $\underline{S}^i_{[\underline{k}]}$ and $\underline{C}^i_{[\underline{k}]}$ for $i = 1, \ldots, m$. Also, calculate $\underline{T}^i_{[\underline{k}]}$ for $i = m, \ldots, 1$ for $\underline{\pi}$. Then, calculate $\underline{L}^i_{[\underline{k}]}$ for $i = 1, \ldots, m$ for $\underline{\pi}$. Lastly, compute $C(\underline{\pi})$ for $\underline{\pi}$ using (3.55). For $\mathsf{insert\text{-}neighbourhood}(\pi)$, we do the same as we do for $\mathsf{insert\text{-}neighbourhood}(\pi, \bar{k})$ but for each $1 \leq \bar{k} \leq n$.

Lemma 3.7. *Evaluation of all permutations in* $\mathsf{insert\text{-}neighbourhood}(\pi, \bar{k})$ *and* $\mathsf{insert\text{-}neighbourhood}(\pi)$ *incrementally requires* $\mathcal{O}(mn)$ *and* $\mathcal{O}(mn^2)$ *time respectively.*

Proof. Computing $\bar{S}$, $\bar{C}$, $\bar{T}$ values for $\bar{\pi}$ needs $\mathcal{O}(mn)$ time. Then, computing $\underline{S}$, $\underline{C}$, $\underline{T}$, $\underline{L}$ and the makespan of each $\underline{\pi}$ needs $\mathcal{O}(m)$ time; there are n such $\underline{\pi}$. Thus, evaluation of all permutations generated by $\mathsf{insert\text{-}neighbourhood}(\pi, \bar{k})$ and $\mathsf{insert\text{-}neighbourhood}(\pi)$ requires $\mathcal{O}(mn)$ and $\mathcal{O}(mn^2)$ time respectively. □

3.2.5 Proposed Constructive Heuristic Algorithm

NEH [65] is a very well-known greedy constructive heuristic for PFSPs [67]. Because of its effectiveness, there exist a number of variants [115–117] for PFSPs and also for other PFSP variants [7, 118]. NEH has two main steps: ordering jobs initially and then inserting jobs, one by one, into their best positions in the already obtained partial sequences. While ordering, NEH assigns high priorities to the jobs with large sums of processing times over all machines. However, that ordering rule is not always the best for other PFSP variants. Based on the exact model of a PFSP variant, more efficient ordering rules may be obtainable [7]. In addition, prioritising jobs in general still remains a challenge in flowshop scheduling [117]. In this paper, we focus on the initial ordering of the jobs in NEH, and develop a new problem-dependent ordering rule to guide the insertion procedure for PFSP-BS.

The proposed ordering rule has two main differences with the original one. In the original one, assuming $w_j = \sum_{i=1}^{m} p_j^i$, NEH sorts jobs on the non-increasing order of w_js to obtain an initial order. In this paper, first, we modify w_j to get the SDSTs into the formula. For this, we define (sequence independent) average setup time $\hat{s}_j^i = \sum_{\bar{j}=1}^{n,\bar{j}\neq j}(s_{j\bar{j}}^i + s_{\bar{j}j}^i)/(2n-2)$ for each job j on each machine i. Then, we redefine $w_j = \sum_{i=1}^{m}[\beta \times \hat{s}_j^i + (1-\beta) \times p_j^i]$ where β will be empirically determined. Notice that β is to balance the relative weight of the setup time and the processing time associated with job j on machine i. The second modification to the ordering rule is that we choose the non-decreasing order of w_j instead of the non-increasing order. This is because scheduling jobs with higher w_j will tend to cause more delays for the following jobs, particularly for the blocking constraints. We call the proposed heuristic for PFSP-BS as NEH-BS. Using the proposed acceleration method, the time complexity of NEH-BS is $\mathcal{O}(n^2m)$.

3.2.6 Proposed Guided Local Search Algorithm

Algorithm 15 presents our proposed guided local search (GLS) framework. It starts from a greedily constructed initial solution. For this, we use our proposed NEH-BS constructive heuristic, which achieves problem-specific guidance from the use of setup times in job ordering. Then, the initial solution undergoes a guided intensification procedure to generate a current solution. Then, the current solution iteratively passes

through a guided diversification procedure followed by the same guided intensification procedure to produce a solution that is accepted or rejected using a given criterion. We use the simulated annealing (SA) based acceptance criterion using temperature schedule $e^{-\Delta/T}$ with T being replaced by γT in each iteration, where γ is less than but very close to 1. The main differences of GLS with other typical local search procedures (such as iterated local search (ILS) [119]) are the latter uses more of random approaches for next solution generation while the former takes guided approaches. Below we describe our guided intensification and diversification procedures.

Algorithm 15 Guided Local Search Framework

1: $\pi \leftarrow$ GuidedInitialisation().
2: $\pi \leftarrow$ GuidedIntensification(π).
3: **while** termination criteria not satisfied **do**
4: $\quad \pi' \leftarrow$ GuidedDiversification(π).
5: $\quad \pi' \leftarrow$ GuidedIntensification(π')
6: $\quad \pi \leftarrow$ AcceptanceFunction(π, π')
7: **return:** the global best solution found so far.

Constraint Guided Intensification

An intensification phase involves moving only to a better neighbouring solution from the current solution. Given a current solution, if there is no better neighbouring solution, then the current solution is returned as a local optimum solution. Typically the neighbouring solutions are generated from the current solution by using a random or an exhaustive approach. For example, in flowshops, the job to be removed first and then inserted back or jobs to be swapped are selected randomly, or alternatively all jobs are considered exhaustively. Intensification is only in the selection of the best solution from the neighbours. In this paper, we propose to select jobs greedily using constraint based problem specific knowledge as a guidance. We use insert-neighbourhood(π, k) since we also have proposed an acceleration method for that.

Our constraint-guided job selection is based on a measure of delays caused by each job. Blocking constraints cause a machine to be blocked with the current job until some latter machine can take it. We apportion this blocking period of a machine to the job causes blocking. In addition, because of the SDST constraints, machines must be set up after processing a job to become ready to process the following job. This setup time is associated with both of the jobs. Now setup times can be overlapping or non-overlapping

with the blocking times. In the former case, the sum of blocking and setup times is to be considered while in the latter case, the larger of the two. So the total delay for each job then can be used to identify the problematic jobs. Given a solution π, each job $[k]$ has the total delay time $\mathsf{TDT}(k) = \sum_{i=1}^{m} \mathsf{DT}^i_{[k]}$ where

$$\sigma^i_{[k]} = 0 \qquad i < 1 \vee i > m \vee k \leq 1 \vee k > n \quad (3.56)$$

$$\mathsf{DT}^i_{[k]} = \mathsf{BT}^i_{[k]} + \sigma^i_{[k+1]} + \sigma^i_{[k]} \qquad O_i = \mathsf{N} \quad (3.57)$$

$$\mathsf{DT}^i_{[k]} = \mathsf{BT}^i_{[k]} \qquad O_i = \mathsf{O}, \mathsf{BT}^i_{[k]} \geq \sigma^i_{[k+1]} \quad (3.58)$$

$$\mathsf{DT}^i_{[k]} = \sigma^i_{[k+1]} \qquad O_i = \mathsf{O}, \mathsf{BT}^i_{[k]} < \sigma^i_{[k+1]} \quad (3.59)$$

$$\mathsf{DT}^i_{[k]} = \sigma^i_{[k]} \qquad O_i = \mathsf{O}, \mathsf{BT}^i_{[k-1]} < \sigma^i_{[k]} \quad (3.60)$$

$$\mathsf{BT}_{i,[k]} = \begin{cases} S^{i+2}_{[k]} - C^i_{[k]} & B_i = \mathsf{RCb} \\ S^{i+1}_{[k]} - C^i_{[k]} & B_i = \mathsf{RSb} \\ C^{i+1}_{[k]} - C^i_{[k]} & B_i = \mathsf{RCb*} \\ 0 & B_i = \mathsf{Wb} \end{cases} \quad (3.61)$$

Algorithm 16 shows our constraint guided intensification procedure. First we create a sequence of jobs in the non-increasing order of their total delay times. The most problematic jobs are thus towards the beginning of the sequence and will be considered earlier than the jobs towards the end. We then consider each job from the sequence and use the selected job to generate the neighbouring solutions. The best solution is taken as the next solution and the process continues. If there is no better solution with a selected job, we consider the next job in the sequence created based on delay times.

Algorithm 16 GuidedIntensification(π)

1: Let π^{D} be the sequence of all jobs when arranged in the non-increasing order of the total delay times $\mathsf{TDT}(k)$ each job $[k]$ in the current solution π.
2: **for** For $k = 1$ to n // to iterate over π^{D} **do**
3: Let k' be such that $[k']$ in π is the same as $[k]$ in π^{D}.
4: Let π' be the $\underline{\pi} \in \mathsf{insert\text{-}neighbourhood}(\pi, k')$ with the lowest makespan but $\underline{\pi}$ is not the same as π.
5: If $C(\pi') < C(\pi)$ then $\pi = \pi'$ and go to Step 2.
6: **return:** π as the solution after intensification..

Random-Path Guided Diversification

A diversification phase takes the search away from a local optimum, even by moving to a worse solution. A very typical diversification approach is to start from a completely random solution. However, this approach results into very low quality solutions and the following intensification phase then needs much time to reach good solutions again. To remedy this issue, other approaches consider some perturbation procedures that make a small number of random moves from the current solution. In this paper, we propose a guided diversification method. Our method first generates a random target solution. It then explores a random path from the current solution to the target solution to obtain the best solution on the path. The random target solution perhaps allows the search to go to another part of the search space and disallows making aimless random moves. The random path allows exploration of many other intermediate solutions between the target and the current solutions. Moreover, taking the best solution from those explored on the path ensures the quality of the returned solution is not very bad. Algorithm 17 shows the proposed random-path guided diversification method.

Algorithm 17 GuidedDiversification(π)

1: $\pi_{\mathsf{s}} \leftarrow \pi$ // start exploration from the given solution.
2: $\pi_{\mathsf{t}} \leftarrow$ FullyRandomSolution(). // the target solution
3: $\pi_{\mathsf{o}} \leftarrow$ FullyRandomJobOrder(). // for a random path
4: $\pi_* \leftarrow \pi_{\mathsf{t}}$ // best solution, never return the given solution.
5: **for** For $k = 1$ to n **do**
6: $j \leftarrow$ the job at position k in π_{o}.
7: $k' \leftarrow$ position of job j in π_{s}.
8: $k'' \leftarrow$ position of job j in π_{t}.
9: if $k' \neq k''$ then swap jobs at k' and k'' in π_{s}.
10: if $C(\pi_{\mathsf{s}}) < C(\pi_*)$ then $\pi_* = \pi_{\mathsf{s}}$
11: **return:** π_* as the solution after diversification.

A path relinking procedure [62] has been used in Algorithm 17 to explore a random path from a given solution π to a random solution. We explain the path relinking procedure using an example. Assume the current solution is $\pi\langle 2,3,5,1,4\rangle$, the random target solution is $\pi_{\mathsf{t}} = \langle 1,4,2,3,5\rangle$, and a random path is a job order $\pi_{\mathsf{o}} = \langle 3,2,5,1,4\rangle$. We will start from π, so $\pi_{\mathsf{s}} = \pi$. Based on π_{o}, job 3 is at the first position of π_{o} and is considered first. Since job 3 is at position 2 in π_{s} and at position 4 in π_{t}, we swap the jobs at positions 2 and 4 in π_{s} to obtain a new $\pi_{\mathsf{s}} = \langle 2,1,5,3,4\rangle$. Then job 2 at the second position of π_{o} is considered. Gradually, all jobs in π_{o} are considered in the order they appear in π_{o}. The solution returned by Algorithm 17 is the best among all the new π_{s} generated and π_{t}. The given solution π_{s} is never returned so that the search can always move to a new solution.

3.2.7 Experimental Results

Taillard's 120 instances [1] are one well-known benchmark in PFSP literature. These instances include 12 groups each comprising 10 instances of the same problem size. The 12 groups have problem sizes $\{20,50,100\} \times \{5,10,20\}$, $\{200\} \times \{10,20\}$ and $\{500 \times 20\}$ in terms of $n \times m$ values. Moreover, in each group, the processing time of each job at each machine is uniformly distributed over $[1,100]$. In PFSP-SDST literature, setup times have been uniformly generated in the ranges of $[1,10)$, $[1,50)$, $[1,100)$ and $[1,125)$ to obtain respectively four scenarios SDST10, SDST50, SDST100 and SDST125, each having 120 Taillard's instances [120]. These scenarios allow to see the effect of having SDSTs larger and smaller than processing times. In the MBPFSP literature, blocking constraints RSb, RCb, RCb*, and Wb are uniformly distributed in each of the 120 Taillard's instances to obtain a benchmark set [7]. In this paper, for PFSP-BS, we uniformly distribute the blocking constraints in each of the $120 \times 4 = 480$ PFSP-SDST instances. Also, we uniformly distribute overlapable/non-overlapable setup time scenarios in each instance. Thus, the 480 instances (denoted by label $O_i = \mathsf{O/N}$) having overlapable/non-overlapable blocking and setup times are our benchmark. However, we will also use 480 instances where only overlapable cases are allowed (denoted by label $O_i = \mathsf{O}$) and in another 480 instances only non-overlapable cases (denoted by label $O_i = \mathsf{N}$).

Using the above mentioned PFSP-BS benchmark, we compare our proposed algorithms with existing state-of-the-art algorithms to be mentioned in appropriate sections. For this, we run each algorithm 5 times on each PFSP-BS instances. We use three different timouts of ρnm milliseconds where $\rho \in \{30, 60, 90\}$. These timeouts give more times to instances having larger n and m. All algorithms have been implemented in programming language C and run on the same high performance computing cluster *G*owonda at *G*riffith University. Each node of the cluster is equipped with Intel Xeon CPU E5-2670 processors @2.60 GHz, FDR 4x InfiniBand Interconnect, having system peak performance 18,949.2 Gflops.

To analyse the experimental results, we define relative percentage deviation RPD to be $\frac{\mathsf{a}-C_\mathsf{r}}{C_\mathsf{r}} \times 100$, where C_a is the makespan obtained for an instance in a given run of a given algorithm and C_r is a reference makespan, which will be clearly mentioned as needed. To summarise our results, we also define average relative percentage deviation (ARPD) using RPDs of the 5 runs of an instance, or of the instances in a group, or in a blocking plus setup time plus overlapping scenario, or even in the entire benchmark.

It is worth mentioning here that we show tables and charts for the performance of only some of the settings of the problems or the solvers. This is mainly to avoid showing similar things repeatedly where the actual conclusion do not vary. So unless it is mentioned otherwise the performance of the other settings are assumed to be similar; if there is any exceptions we will clearly mention as appropriate.

Performance of NEH-BS

We calibrate the parameter β of the proposed NEH-BS and found 0.65 to be the best value, although statistically not significantly better than the other values. Taking $\beta = 0.65$, we then compare our proposed NEH-BS constructive heuristic with PFT_NEH [17] and NNEH [7]. PFT_NEH was recently proposed for PFSP-SDST with only non-overlapping RSb constraints. NNEH was proposed for MBPFSP. There is no parameter for PFT_NEH and we calibrated the parameter of NNEH for PFSP-BS and found 0.1 to be the best value.

Table 3.24 shows the ARPDs obtained by the three algorithms in various settings of PFSP-BS. Here, C_r is the minimum makespan obtained by any of the three algorithms.

Among the 12 settings, NEH-BS is the best in 7 settings while NNEH is the best in 2 settings and PFT_NEH in 3 settings. NNEH does not take setup times into account while obtaining the initial job ordering. It is the best (in fact very slightly better than NEH-BS) when setup times are small and $O_i = \mathsf{N}$. PFT_NEH is the best when setup times are large or non-overlapable with blocking constraints. PFT_NEH performs a costlier $O(n^2)$ search to find an initial job ordering where time wastage due to blocking or being idle is taken into account. NEH-BS appears to be the best when overlapable setup times are present ($O_i = \mathsf{N/O}, \mathsf{O}$).

TABLE 3.24: ARPDs obtained by candidate algorithms

SDST	Algorithm	$O_i = \mathsf{N/O}$	$O_i = \mathsf{O}$	$O_i = \mathsf{N}$
10	NEH-BS	**0.27**	**0.27**	0.38
	FPT_NEH	1.84	1.84	1.99
	NNEH	0.45	0.39	**0.36**
50	NEH-BS	**0.29**	**0.25**	0.46
	FPT_NEH	1.64	1.75	1.53
	NNEH	0.47	0.47	**0.45**
100	NEH-BS	**0.54**	**0.40**	0.89
	FPT_NEH	1.14	1.35	**0.60**
	NNEH	0.67	0.50	1.10
125	NEH-BS	0.97	**0.65**	1.28
	FPT_NEH	**0.76**	0.71	**0.39**
	NNEH	1.12	0.82	1.43

Table 3.25 shows the ARPDs obtained by the three algorithms in each of the 12 instance groups in four SDST scenarios, only in the case where $O_i = \mathsf{N/O}$.

TABLE 3.25: ARPDs obtained by candidate algorithms on each instance group in $O_i = \mathsf{N/O}$ setting. In the table, BS, PFT, and NN respectively denote NEH-BS, PFT_NEH, and NNEH.

	SDST10			SDST50			SDST100			SDST125		
Ins.	BS	PFT	NN	BS	PFT	NN	BS	PFT	NN	BS	PFT	NN
20×5	**0.03**	3.63	0.44	**0.38**	1.77	0.69	**0.86**	0.89	0.93	**1.16**	1.80	1.51
20×10	**0.47**	2.46	0.83	**0.45**	1.09	0.87	1.23	1.66	**0.38**	**0.87**	1.53	1.62
20×20	0.69	1.62	**0.49**	**0.34**	2.42	0.53	**0.24**	1.29	1.62	0.99	**0.68**	1.41
50×5	**0.14**	1.13	0.38	**0.20**	1.09	0.70	**0.24**	0.81	0.72	1.52	**0.17**	1.40
50×10	**0.07**	1.86	0.85	**0.30**	1.64	0.33	**0.64**	1.11	0.73	**0.54**	0.55	0.89
50×20	**0.39**	1.54	0.68	**0.04**	1.80	0.89	**0.17**	1.87	0.58	**0.71**	0.74	0.72
100×5	**0.08**	1.69	0.09	0.41	1.43	**0.11**	0.77	0.62	**0.61**	1.60	**0.03**	1.47
100×10	**0.30**	1.80	0.50	**0.26**	1.25	0.45	0.60	**0.28**	0.69	1.15	**0.18**	1.02
100×20	**0.13**	1.98	0.33	0.39	1.88	**0.28**	0.44	1.40	**0.40**	**0.43**	0.82	0.62
200×10	0.37	1.36	**0.14**	0.37	1.73	**0.22**	**0.71**	1.15	0.73	1.33	**0.78**	1.43
200×20	**0.18**	1.87	0.24	**0.12**	2.14	0.22	0.26	1.32	**0.24**	**0.53**	0.99	**0.53**
500×20	**0.43**	1.18	0.51	**0.26**	1.40	0.35	**0.32**	1.22	0.42	0.84	**0.83**	0.84
Average	**0.27**	1.84	0.45	**0.29**	1.64	0.47	**0.54**	1.14	0.67	0.97	**0.76**	1.12

Calibrating GLS Parameters

We calibrate the parameters T and γ in the SA based acceptance criterion of the proposed GLS algorithm. For this and to avoid overfitting issues, we generate 44 new instances adapting instances from other PFSP variants to PFSP-BS. We employ the design of experiments method [107] to analyse the effects of these parameters on the performance. For T, values in $\{10, 50, 100\}$ are considered while for γ, values are in $\{0.99, 0.999, 0.9999\}$. For each setting, GLS is run 5 times on each instance. Then, ARPDs are computed using the minimum makespans obtained only in this experiment as C_{r} and are shown in Figure 3.21. We see that $T = 10$ and $\gamma = 0.9999$ are the best values although not significantly better than the closest values. We will use these values in our further experiments.

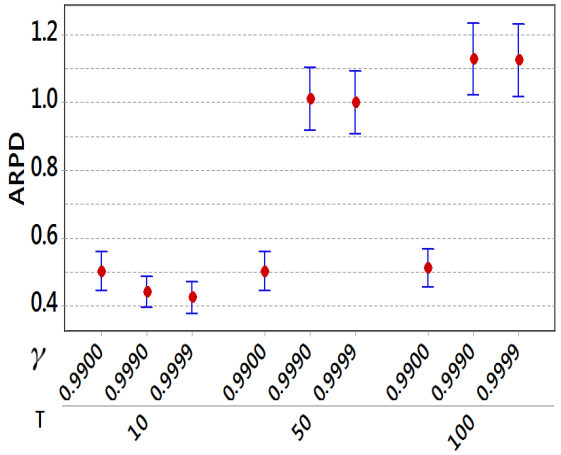


FIGURE 3.21: 95% confidence intervals for T and γ combinations in GLS.

Evaluating GLS Components

We evaluate our SA based acceptance criterion, guided intensification method, and guided diversification method using the 44 instances used in Section 3.2.7. For this, each setting of the GLS algorithm is run 5 times on each instance. Then, ARPDs are computed using the minimum makespans obtained only in the respective experiments as C_{r}.

Evaluation Acceptance Criterion: We compare our SA based acceptance criterion (along with $T = 10$ and $\gamma = 0.9999$) with a random acceptance (RA) criterion [76] with probability 50% , a threshold based acceptance criterion (TA) with threshold values that

depend on the solution quality as in [104], and also with a criterion of not accepting (NA) any worse solutions. Figure 3.22 shows 95% confidence intervals with Tukey's HSD and we see that the SA based criterion significantly outperforms the other three criteria.

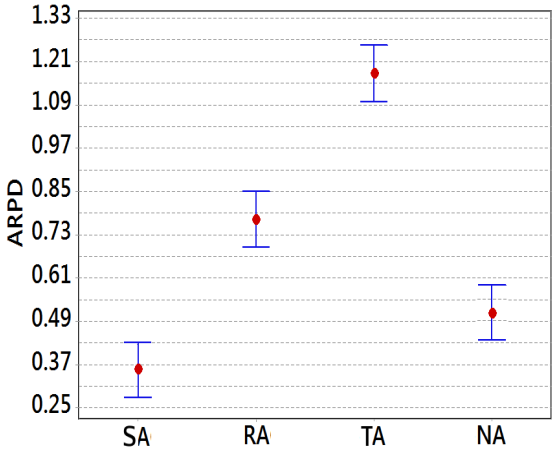


FIGURE 3.22: 95% confidence intervals for various acceptance criteria in GLS.

Evaluating Constraint Guided Intensification: Figure 3.23 shows the performance of our constraint-guided intensification method (denoted by G) against two other methods. These two methods are a random job selection based neighbour generation method (denoted by R) and a reference solution based neighbour generation method [121] (denoted by Ref). For method R, in Algorithm 16, π^D is constructed from a random permutation and for method Ref, π^D is the best solution found so far in the search. From the 95% confidence intervals with Tukey's HSD, we see that our guided intensification method significantly outperforms the other two methods.

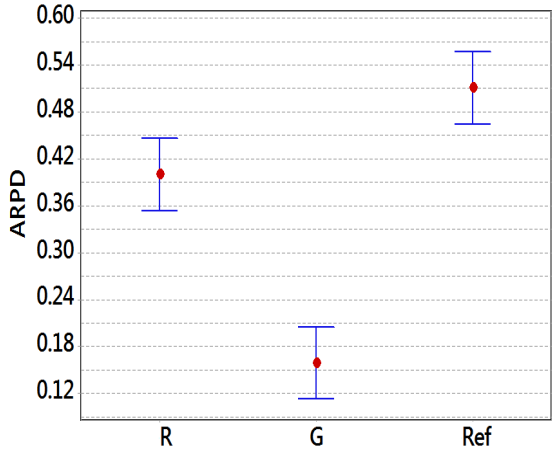


FIGURE 3.23: 95% confidence intervals for various intensification strategies.

Evaluating Random-Path Guided Diversification: Figure 3.24 shows the performance our random-path guided diversification method (SRPR) against two similar other typical methods denoted by SPR and IPR. All these three use a random permutation as π_{t} in Algorithm 17, but for π_{o}, SPRR uses a random order while SPR and IPR both uses a linear order from 1 to n. Moreover, SRPR and SPR use swapping in changing π_{s} while IPR uses insertion. Figure 3.24 also shows two other methods named SBOX and OX [120], which are typical crossover operators and are applied on π_{t} and π_{s}. From the 95% confidence intervals with Tukey's HSD, we see that our guided diversification method SPRR significantly outperforms the other four methods.

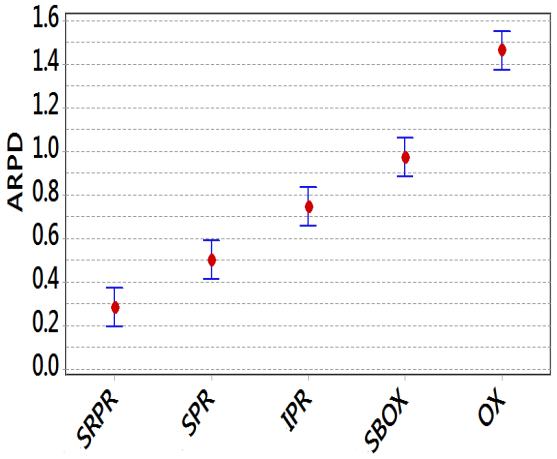


FIGURE 3.24: 95% confidence intervals for various diversification strategies.

Overall Performance Comparison

Our final GLS algorithm has NEH-BS with $\beta = 0.65$ in initialisation, method G in intensification, method SRPR in diversification, and SA based acceptance method with $T = 10$ and $\gamma = 0.9999$. We compare its overall performance with a number of solvers for PFSP-BS using aforementioned 120 instances for each of the 4 SDST cases and $O_i \in \{\mathsf{N/O}, \mathsf{O}, \mathsf{N}\}$. There exists no algorithm for PFSP-BS as this is just proposed in this paper. We therefore consider state-of-the-art algorithms of several PFSP variants and adapt them to PFSP-BS. After running all algorithms including our proposed GLS, in the ARPD computation for each instance, we use the minimum makespan obtained by any of the algorithm as C_{r}.

We adapted iterated greedy algorithm (IG_RS) [40] for PFSPs with SDSTs, iterated local search algorithm (ILS_PR) [38] for PFSPs, modified fruit fly optimisation (MFFO)

algorithm [111] for PFSPs with RSb, scatter search (SS) algorithm [7] for MBPFSP, a new iterated greedy algorithm (IG_IJ) [39] for PFSPs with RSb, and water wave optimization (WWO) algorithm [17] for PFSPs with RSb and non-overlapable SDSTs. Adaptation requires using the PFSP-BS model and the accelerated makespan computation while the key components of the search algorithms remain the same. Moreover, for effective adaptation, we also calibrated the parameters of those algorithms in the same way we have done for GLS using the same 44 generated instances. Table 3.26 shows the selected parameter values of the adapted algorithms.

TABLE 3.26: The selected parameter values of the adapted algorithms.

Algorithm	Selected Parameter Values
IG_RS [40]	$d = 5, T = 0.6$
ILS_PR [38]	$\lambda = 4, \gamma = 2$
MFFO [111]	PS = 10, pls = 0.7
SS [7]	SizeofPop = 30, r1 = 10, r2 = 10, MaxIter = 25
IG_IJ [39]	dS = 6, tP = 0.5, jP = 0.01
WWO [17]	PS = 5, $h_{\max} = 10$, $w = 20$, $\lambda_{\min} = 2$, $\lambda_{\max} = 3$

Table 3.27 shows the ARPDs obtained by the 7 algorithms (IG_RS, ILS-PR, MFFO, SS, IG_IJ, WWO, and GLS) with $\rho = 90$ in each of the 4 SDST scenarios and in each of the three cases $O_i =$ N/O, O, and N. As can be seen, in each case, proposed GLS algorithm outperforms all the other 6 algorithms with WWO and IG_IJ being the second and third best performing algorithms.

TABLE 3.27: ARPDs of obtained by the competing algorithms when $\rho = 90$

O_i	SDST	IG_RS	ILS-PR	MFFO	SS	IG_IJ	WWO	GLS
N/O	10	2.49	0.90	1.69	0.64	0.60	0.36	**0.13**
	50	3.13	1.19	2.26	0.86	0.81	0.55	**0.16**
	100	4.02	1.74	3.08	1.19	1.11	0.73	**0.28**
	125	4.54	2.04	3.52	1.38	1.28	0.85	**0.35**
O	10	2.47	0.92	1.63	0.64	0.59	0.40	**0.15**
	50	2.77	1.05	1.84	0.75	0.70	0.49	**0.17**
	100	3.55	1.52	2.58	1.09	1.00	0.72	**0.23**
	125	4.04	1.85	3.10	1.32	1.22	0.84	**0.30**
N	10	2.53	1.00	1.72	0.67	0.64	0.42	**0.17**
	50	3.24	1.40	2.37	0.88	0.84	0.64	**0.20**
	100	4.19	1.96	3.27	1.18	1.13	0.83	**0.32**
	125	4.56	2.17	3.65	1.28	1.23	0.93	**0.35**

We perform an ANOVA test to check that the performance differences are statistically significant. In all cases, the p value was 0.00, which is less than $\alpha = 0.05$. It means that there is a significant difference at least between two of the algorithms. Therefore, a post-hoc test is needed to find those methods. So for this, a 95% confidence interval

test is also done and the intervals for only $O_i = \mathsf{N/O}$ are shown in Figure 3.25. From the figure, we can clearly see that the proposed GLS algorithm significantly outperforms the other six algorithms in all four SDST scenarios. Confidence intervals for $O_i = \mathsf{O}$ and N are very similar to Figure 3.25.

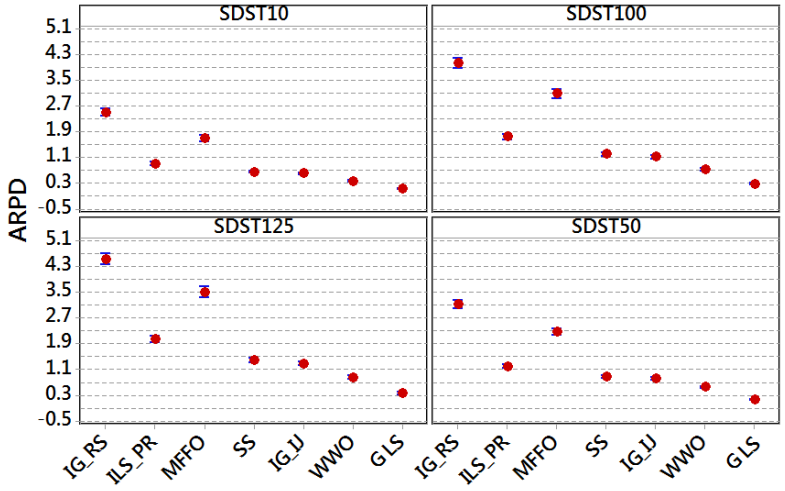


FIGURE 3.25: 95% confidence intervals for algorithms when $\rho = 90$ and $O_i = \mathsf{N/O}$.

For further detailed results, on each of the 12 groups of 10 instances, we show the ARPDS in Table 3.28 for the best performing three algorithms and when $\rho = 90$ and $O_i = \mathsf{N/O}$. The differences in the performance levels are very clear from this table as well.

TABLE 3.28: ARPDs obtained by the three best performing algorithms on each of the 12 groups of 10 instances when $\rho = 90$ and $O_i = \mathsf{N/O}$

	SDST10			SDST50			SDST100			SDST125		
Ins.	IG_IJ	WWO	GLS	IG_IJ	WWO	GLS	IG_IJ	WWO	GLS	IG_IJ	WWO	GLS
20×5	**0.01**	0.03	**0.01**	0.05	0.04	**0.02**	0.04	0.07	**0.02**	0.05	0.12	**0.06**
20×10	0.01	0.07	**0.00**	**0.02**	0.05	**0.02**	**0.01**	0.06	**0.01**	0.05	0.11	**0.04**
20×20	0.03	0.07	**0.00**	**0.02**	0.06	**0.02**	0.06	0.07	**0.05**	**0.03**	0.07	**0.03**
50×5	0.55	0.37	**0.09**	0.89	0.68	**0.27**	1.22	0.96	**0.40**	1.47	1.10	**0.59**
50×10	0.59	0.38	**0.12**	0.89	0.60	**0.24**	1.25	0.91	**0.47**	1.52	1.01	**0.48**
50×20	0.73	0.50	**0.22**	0.88	0.62	**0.25**	1.17	0.78	**0.43**	1.42	0.86	**0.43**
100×5	0.61	0.30	**0.11**	1.05	0.67	**0.16**	1.69	1.15	**0.38**	2.14	1.44	**0.53**
100×10	0.97	0.46	**0.18**	1.32	0.76	**0.22**	1.75	1.00	**0.37**	2.03	1.30	**0.54**
100×20	1.05	0.55	**0.20**	1.31	0.69	**0.28**	1.69	0.86	**0.29**	1.80	0.94	**0.50**
200×10	0.86	0.32	**0.20**	1.17	0.67	**0.16**	1.62	0.90	**0.26**	1.85	1.11	**0.41**
200×20	1.00	0.41	**0.18**	1.24	0.71	**0.16**	1.65	0.88	**0.33**	1.80	1.04	**0.37**
500×20	0.77	0.88	**0.22**	0.83	1.02	**0.18**	1.20	1.11	**0.29**	1.20	1.14	**0.27**
Average	0.60	0.36	**0.13**	0.81	0.55	**0.16**	1.11	0.73	**0.28**	1.28	0.85	**0.35**

For yet further detailed results on each of the 120 instances of SDST125 and when $\rho = 90$, we show the ARPDs obtained by the best performing three algorithms in Figure 3.26. Our proposed GLS outperforms the other two algorithms with wide margins, particularly in the large instances. Performances of the algorithms are similar in other SDST settings.

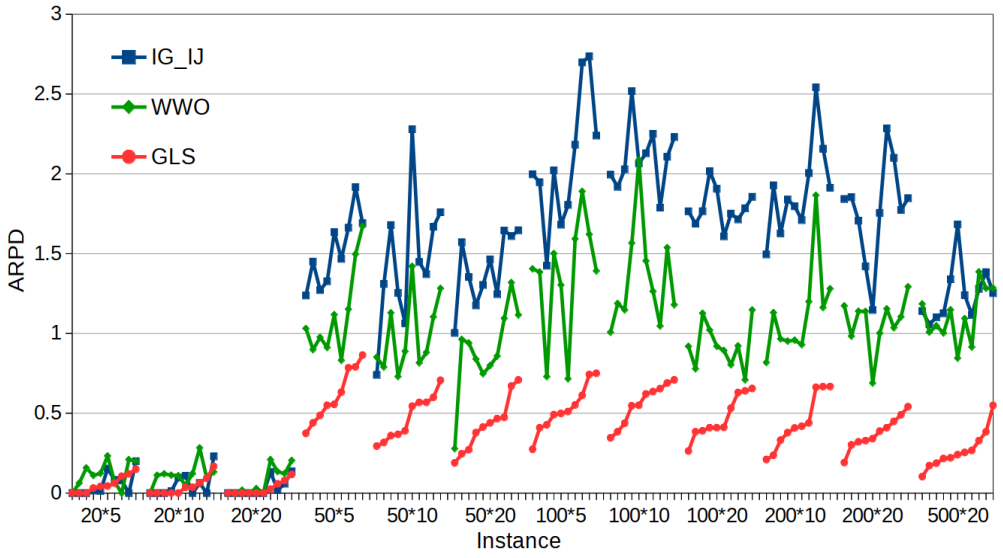


FIGURE 3.26: 95% confidence intervals for the best performing three algorithms on 120 instances of SDST125 scenario and when $\rho = 90$ and $O_i = \mathsf{N/O}$.

To see the performance of the 7 algorithms over various timeouts, we did the whole experiment also with $\rho = 30$ and 60. Since the results are very similar to those in Tables 3.27 and 3.28, and Figures 3.25 and 3.26, we do not present them.

Nevertheless, to obtain a summary, Table 3.29 shows the numbers of best solutions found by the competing algorithms in our entire experiment. Clearly, our proposed GLS outperforms the other six algorithms by a very wide margin.

TABLE 3.29: Numbers of best solutions found by the competing algorithms

O_i	SDST	IG_RS	ILS-PR	MFFO	SS	IG_IJ	WWO	GLS
N/O	10	12	30	26	26	30	35	**116**
	50	5	30	24	22	30	30	**120**
	100	4	30	21	30	29	35	**115**
	125	6	30	22	26	29	31	**119**
O	10	7	30	24	30	31	37	**109**
	50	5	30	19	24	30	32	**113**
	100	2	28	16	28	31	34	**110**
	125	0	24	11	20	27	26	**113**
N	10	5	31	18	30	29	34	**111**
	50	6	31	22	23	29	35	**112**
	100	3	29	20	24	30	35	**109**
	125	3	30	20	28	30	28	**108**

Effect of Numbers of Jobs Figure 3.27 shows confidence intervals of the ARPDs of the competing algorithms over the number of jobs n for SDST125 and when $O_i = \mathsf{N/O}$. We see from the interval plots that the performance of GLS is better than that of the other algorithms and almost remains consistent (although slightly becoming better) with the increase of n, while for other algorithms performance levels vary significantly.

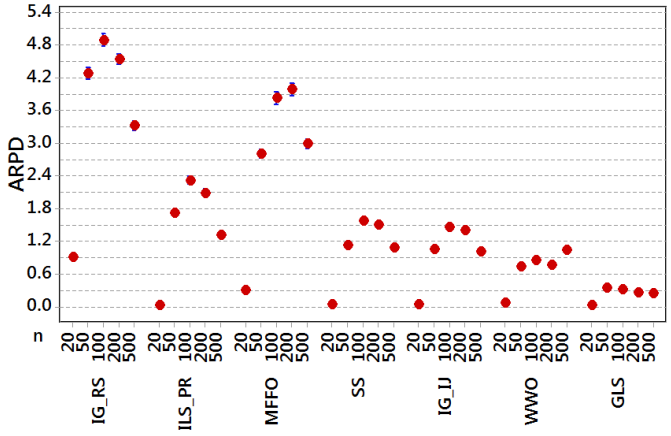


FIGURE 3.27: 95% confidence intervals for the competing algorithms over numbers of jobs n only for SDST125 and only when $O_i = \mathsf{N/O}$.

Effect of Numbers of Machines

Figure 3.28 shows confidence intervals of the ARPDs of the competing algorithms over the number of machines m for SDST125 and when $O_i = \mathsf{N/O}$. We see from the interval plots that the performance of GLS is significantly better than that of the other algorithms and almost remains consistent with the increase of m, while for some of the other algorithms performance levels vary significantly.

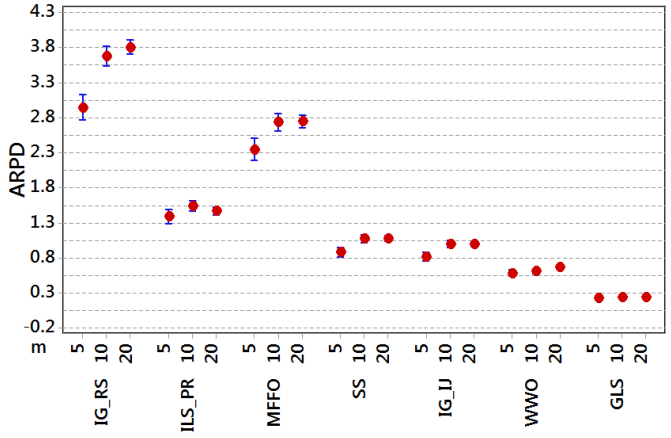


FIGURE 3.28: 95% confidence intervals for the competing algorithms over numbers of machines m only for SDST125 and only when $O_i = \mathsf{N/O}$.

Effect of Timeout Figure 3.29 shows confidence intervals of the ARPDs of the competing algorithms over the timeout parameter ρ values when $O_i = \mathsf{N/O}$. We see from the interval plots that the performance of GLS is significantly better than that of the other algorithms when ρ values are the same or even not. Expectedly, performance of each algorithm improves with the increase of ρ.

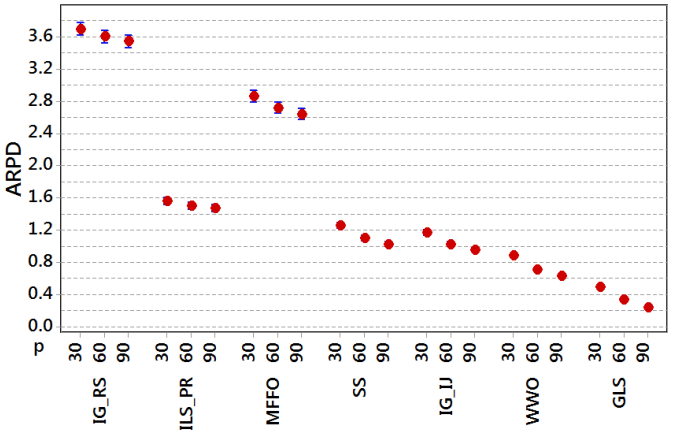


FIGURE 3.29: 95% confidence intervals for the competing algorithms over timeout parameter values ρ only when $O_i = \mathsf{N/O}$.

Effect of SDST Scenarios Figure 3.30 shows confidence intervals of the ARPDs of three best performing algorithms over various SDST scenarios when $O_i = \mathsf{N/O}$. We see from the interval plots that the performance of GLS is significantly better than that of the other algorithms. Moreover, performance of each algorithm degrades when SDST values increase. However, differences between the performances of pairs of algorithms increase with the increase of SDST values.

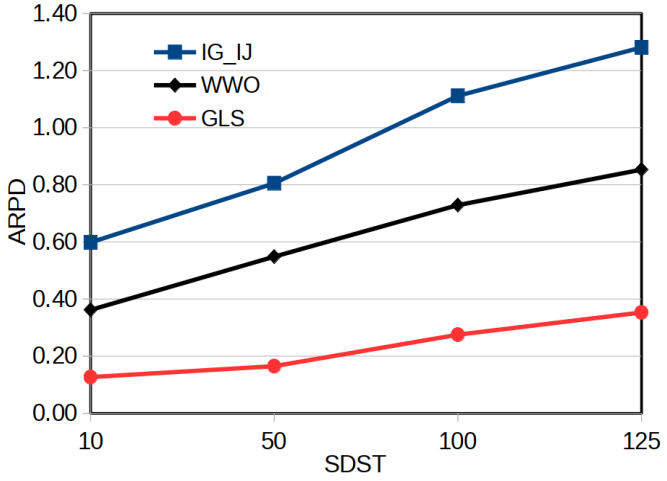


FIGURE 3.30: 95% confidence intervals for three best performing algorithms over various SDST scenarios only when $O_i = \mathsf{N/O}$.

Comparing with Lower Bounds

Since there is no known lower bound of makespan for PFSP-BS, we compare our results with some lower bounds of makespan obtained by using related problems. When optimal values are not known, lower or upper bounds are often obtained by relaxing some

constraints and solving the relaxed problems. Assuming no blocking constraints, we can transform PFSP-BS to PFSP-SDST. Then, making further assumption that SDSTs are all zero, we can obtain typical PFSPs. Alternatively, assuming SDSTs are all zero, we can transform PFSP-BS to MBPFSP and then further assuming no blocking constraints, we can obtain PFSPs. Given the 120 Taillard's instances [1], unfortunately, no optimal solution is known either for PFSPs or for PFSP-SDSTs or for MBPFSP. Nevertheless, it is obvious that given a problem instance, optimal makespan for the typical PFSP version will be smaller than the optimal makespan for the PFSP-SDST or MBPFSP, which will be smaller than the optimal makespan for the PFSP-BS. In the absence of optimal values for PFSP-BS, we compare our results with the best known makespan values of the typical PFSPs [122] and PFSP-SDSTs [40] and MBPFSPs. However, we note that these comparisons are just indicative and not definitive. Figure 3.31 shows these results. Notice that the more the SDST periods the wider the gap between the best known makespans for typical PFSPs and that for PFSP-SDSTs; which is expected. Similar observations can be made for the gap between the makespans for MBPFSP and PFSP-BS.

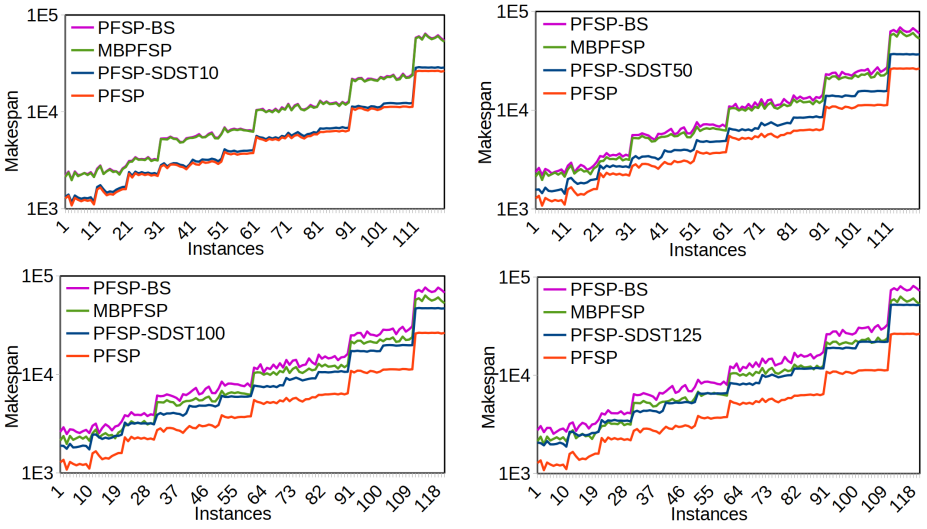


FIGURE 3.31: Comparison against lower bounds of makespan obtained from related problems.

Further Results on Blocking Variants

So far in all our experiments, we have used problem instances with a uniform distribution of blocking constraints RSb, RCb, RCb*, and Wb. It is possible to generate and use other problem instances with various types of blocking constraint distributions. However, we

do not show all those detailed results. In Table 3.30, we just show summarised results on all SDST scenarios for two configurations: RSb-N having only RSb constraints with non-overlapable SDSTs and RSb-RCb-O having a mix of RSb and RCb constraints with overlapable SDSTs. We use these two configurations because there exist studies on these [17, 30]. From the table, it is clear that the proposed GLS outperforms the other six algorithms significantly, which was also verified by ANOVA and Tukey's HSD tests with $\alpha = 0.5$.

TABLE 3.30: Performance comparison on blocking variants

B_i-O_i	SDST	IG_RS	ILS-PR	MFFO	SS	IG_IJ	WWO	GLS
RSb-N	10	4.34	1.75	3.14	1.08	0.99	0.75	**0.28**
	50	4.27	1.87	3.19	1.17	1.08	0.88	**0.30**
	100	5.08	2.41	3.94	1.49	1.42	1.10	**0.38**
	125	5.63	2.66	4.32	1.63	1.56	1.20	**0.47**
RSb-RCb-O	10	2.61	1.05	1.78	0.62	0.64	0.44	**0.18**
	50	2.81	1.10	1.97	0.74	0.70	0.50	**0.17**
	100	3.40	1.50	2.54	0.91	0.96	0.70	**0.24**
	125	3.86	1.77	2.94	1.03	1.13	0.82	**0.29**

3.2.8 Overall Discussion

We have first proposed PFSP-BS, which is a generalisation over PFSP-SDST and MBPFSP. We then have developed formulas to calculate makespan of a PFSP-BS and show that makespan minimisation is NP-hard. We then have developed a constructive heuristic algorithm NEH-BS and a guided local search algorithm GLS for PFSP-BS. Both algorithms have been calibrated to find optimal parameter values. To compare our algorithms, we have used benchmark instances that are generated based on 480 PFSP-SDST instances [120].

NEH-BS has been empirically compared with PFT_NEH [17] and NNEH [7], both adapted to PFSP-BS and parameters are calibrated. NEH-BS is found to be better than both PFT_NEH and NNEH when overlapable SDSTs are considered either solely or in a mixed fashion.

GLS have been empirically compared with IG_RS [40], ILS_PR [38], MFFO [111], SS [7], IG_IJ [39], and WWO [17]. These algorithms are originally for various PFSP variants and are adapted for PFSP-BS along with parameter calibration. GLS significantly outperforms all the six algorithms in many combinations of blocking constraints, setup times, and overlapping conditions. GLS obtains the best found solutions in almost

all instances in all settings. We have shown that GLS performs consistently with the increase of number of jobs and number of machines. We also have shown that GLS with the shortest timeout ($\rho = 30$) performs better than any timeout setting ($\rho \in \{30, 60, 90\}$) of the other algorithms. GLS performance slightly degrades with the increase of setup time duration, but the performance gap with the second and third best performing algorithms increases.

3.3 Conclusions

Mixed blocking permutation flowshop scheduling problem (MBPFSP) with the objective of makespan minimisation is NP-hard. It has important industrial applications that include the cider production industry. MBPFSP has made some progress in recent years. However, within practical time limits, existing incomplete algorithms still either find low quality solutions or struggle with large problems, mainly because of using generic heuristics or metaheuristics that usually lack problem specific structural knowledge. In MBPFSP, a machine could be blocked with the currently finished job until the subsequent machine is available to process the same job. These blocking constraints affect the makespan. So MBPFSP search should naturally take explicit steps to take the blocking constraints into account. Unfortunately, existing research on MBPFSP just uses only the makespan to compare generated solutions, but the search otherwise is not aware of the blocking constraints. Moreover, existing such methods use either an exhaustive or a random neighbourhood generation strategy. In this work, we aim to advance MBPFSP search by better exploiting the problem specific structural knowledge. We use the constraint and the objective functions to obtain such problem specific knowledge and we exploit such knowledge both in a constructive search method and in a local search method. We also present an acceleration method to efficiently evaluate insertion-based neighbourhoods of MBPFSP.

In order to solve this problem, we first present a very effective scatter search (SS) algorithm for this. At the initialisation phase of SS, we use a modified version of the well-known Nawaz, Enscore and Ham (NEH) heuristic. For the improvement method in SS, we use an Iterated Local Search (ILS) algorithm that adopts a greedy job selection and a powerful NEH-based perturbation procedure. Moreover, in the reference

set update phase of SS, with small probabilities, we accept worse solutions so as to increase the search diversity. Next, we have presented techniques to incorporate constraint awareness in different phases of an local search algorithm for mixed blocking permutation flowshop problems (MBPFSP). The blocking constraints are a key characteristic of MBPFSP, but existing algorithms use such constraints only in the makespan calculation. We have shown that further consideration of blocking constraints in constructive heuristics, in neighbourhood generation in the intensification of a local search phase, and in job selection in the diversification phase improves the search performance significantly. Moreover, we have presented a problem-dependent threshold-based simple acceptance criterion. Our experimental results on three standard testbeds demonstrate that our proposed algorithms significantly improve over existing best-performing algorithms.

After studying the MBPFSP, Inspired by the cider industry, we propose a new PFSP variant that generalises over simultaneous use of several types of blocking constraint and various settings of sequence-dependent setup time. We also present a computational model for makespan minimisation of the new variant and show that solving this variant remains NP-hard. We then present an acceleration method to compute makespan efficiently and thus evaluate the neighbourhoods generated by insertion operators. We develop a constructive heuristic taking both blocking constraints and setup times into account. We also develop a new local search algorithm that uses a constraint guided intensification method and a random-path guided diversification method. Our comprehensive experimental results on a set of benchmark instances demonstrate that our proposed algorithms significantly outperform several state-of-the-art adapted algorithms.

Chapter 4

Aircraft Scheduling

In this chapter, we focus on aircraft scheduling problem which is an NP-hard problem. It involves allocation of aircraft to runways for landing and takeoff flights, minimising total flight delays. ASP has made significant progress in recent years. However, within practical time limits, existing incomplete algorithms still struggle with large sized problems [13, 15]. One key reason behind this is the typical way of using generic heuristics or metaheuristics that usually lack problem specific structural knowledge. As a result, existing such methods use either an exhaustive or a random neighbourhood generation strategy. So their search guidance comes only from the evaluation function that is used mainly after the neighbourhood generation. In this work, we aim to advance ASP search by better exploiting the problem specific structural knowledge. We use the constraint and the objective functions to obtain such problem specific knowledge and we exploit such knowledge both in a constructive search method and in a local search method. Our motivation comes from the constraint optimisation paradigm in artificial intelligence, where instead of random decisions, constraint-guided more informed optimisation decisions are of particular interest. This chapter is based on following publications.

- Multi-Runway Aircraft Scheduling
  - Vahid Riahi, MA Hakim Newton, MMA Polash, Kaile Su, and Abdul Sattar. 'Constraint guided search for aircraft sequencing.' Expert Systems with Applications 118 (2019): 440-458.
- Single-Runway Aircraft Scheduling

– Vahid Riahi, MA Hakim Newton, and Abdul Sattar. 'Constraint-guided local search for single mixed-operation runway.' In Australasian Joint Conference on Artificial Intelligence, pp. 329-341. Springer, Cham, 2018.

4.1 Motivation

Airline industries today play a key role in transportation of people and freight. About 58.93 million passengers travelled in year 2016 in Australia, which was 2.5% more compared to that in year 2015 [19]. World wide more than 6 billion passengers travelled with domestic and international flights in 2012, which is anticipated to be more than double by 2025 [123]. Unfortunately, in Europe in March 2015, 7% more departing flights were delayed by about 4 more minutes compared to the 28% departing flights with an average delay of 24 minutes in March 2014. Moreover, 32% arrival flights in the same region were delayed with an average of 32 minutes in March 2015 compared to 25 minutes in March 2014 [124]. Although weather condition is a great factor for these delays, 32% of flight delays between 2010 and 2015 in the US were because of air traffic volume [125]. Flight delays cause additional fuel burning, disrupts flight connections, cause passenger dissatisfaction, and severely affect crew scheduling. According to Federal Aviation Administration (FAA), the total cost of all US air transportation delays in 2007 was estimated to be $31.2 billion [20]. Flight delays are thus a very important growing challenge that needs to be tackled with proper emphasis.

To keep pace with the increasing demand for air transportation, airports have been increasing capacities by building more runways or allowing more flights per runway per unit of time. However, building more runways needs huge investments and availability of space. For example, Brisbane airport in Australia has been constructing a new parallel runway, which is expected to take 8 years with AU$1.35 billion investment [126]. On the other hand, allowing more flights arbitrarily is simply not possible because of the technical restrictions. Each aircraft creates wake turbulence that the subsequent aircraft must avoid, otherwise dangerous consequences might be imminent. Mandatory time separations between pairs of flights are therefore enforced by aviation authorities such as Federal Aviation Administration (FAA) in the United State or Civil Aviation Authority (CAA) in the United Kingdom. However, the separation times essentially restrict the available choices for aircraft sequencing and cause delays in landing and takeoff.

It has been showed that given a significant level of diversity of aircraft, operational delays at a major US airport can be reduced approximately up to four hours a day by optimising landing and takeoff sequences [127]. As a result, developing effective aircraft scheduling algorithms to produce optimal or near-optimal sequences of aircraft has appeared to be a promising technique. These scheduling algorithms will utilise the runways more efficiently to increase the overall capacity of the airports and to smoothen the flow of air traffic. In this work, we develop efficient constraint optimisation search algorithms for *aircraft sequencing problems* (ASP). For this, we use existing aircraft sequencing models from the literature.

Each ASP comprises a set of departing aircraft and a set of arriving aircraft. Each landing or takeoff operation must be performed within a given time window (known as the *time window constraint*). Each aircraft belongs to a class (e.g. heavy, large, and small). Each pair of flights depending on their aircraft classes must be separated by a given time period (known as the *separation time constraint*). Besides the aforementioned two hard constraints that must be satisfied, ASPs might have a number of soft constraints that should be satisfied as much as possible. The soft constraints include restricting the number of flights allocated to a particular runway, scheduling each landing or takeoff at the corresponding given ready time, consideration of safety issues and fuel burnt, prioritising all landing aircraft over departing ones and all heavier aircraft over lighter ones. Interestingly, each soft constraint can be considered as an objective function and vice versa. Among many possible alternatives, as an objective function in ASP and also in this chapter, *total weighted tardiness* of all flights is minimised. Obviously, the smaller the tardiness of a given schedule, the better the quality of the schedule. ASP has been classified as an NP-hard problem [128].

ASP has made significant progress in recent years. However, within practical time limits, existing incomplete algorithms still either find low quality solutions or struggle with large problems. One key reason behind this is the typical way of using generic heuristics or metaheuristics that usually lack problem specific structural knowledge available from the constraints or the objectives of the problem instance. Moreover, existing such methods, for example those by [13] and [21] among others, use either an exhaustive or a random neighbourhood generation strategy. So their search guidance comes only from the evaluation function that is used mainly after the neighbourhood generation. As a result, when such a method performs well, it is typically not known why the method

performs well and what problem specific aspects are behind the performance. To be more specific, in ASP, the time window or separation time constraints, or the total weighted tardiness objective should affect decision making during the search process including the neighbourhood generation; unfortunately, this is not the case with the existing approaches.

There are two different conditions in the ASP based on the number of runways: single runway ASP, and multiple runway ASP. In this chapter, we have studied both cases with the aforementioned constraint-based search approach. Nevertheless, our main focus is on the multiple-runway case.

4.2 Multi-Runway Aircraft Scheduling

In this section, we are to investigate the ASP with the multiple runway case.

4.2.1 Multi-Runway Aircraft Scheduling Problem

Assume an ASP with N aircraft $\{1, \ldots, N\}$ arriving or departing and M runways $\{1, \ldots, M\}$ to perform landing or takeoff operations on. At any time, each runway can be used by only one aircraft and each aircraft can operate on only one runway. There are W aircraft class operation weights $\{1, \ldots, W\}$ for prioritisation of one aircraft's operation over another's. The aircraft class operation weight w_j for an aircraft j depends both on the aircraft class (e.g. small, large, or heavy) and the type of operation (e.g. landing or takeoff). For each pair of aircraft operating on the same runway, there is a minimum separation time $s(w, w')$, where w and w' are the respective aircraft class operation weights. Table 4.1 summarises the separation times that we use in this work. Each aircraft j has a time window $[r_j, l_j]$ defined by ready time and latest time in which the respective landing or takeoff operation for the aircraft is to be performed. Because of separation times, an aircraft might not be scheduled at its ready time. If an aircraft j is scheduled to operate at time $t_j \in [r_j, l_j]$, its weighted tardiness (WT) in that case would be $w_j(t_j - r_j)$. To solve the ASP described above, for each aircraft j, we have to determine the runway $\rho_j \in [1, M]$ to perform the operation on and the start time t_j to perform the operation at. The objective in this case is to minimise the total weighted tardiness (TWT) over all aircraft to be scheduled.

TABLE 4.1: Aircraft classes (e.g. small, large, heavy), aircraft operations (e.g. takeoff, landing), aircraft class operation weights (1–6), and safety separation times (in seconds) between each pair of aircraft operating on the same runway based on the FAA standard.

Operation			takeoff			landing		
	Class		small	large	heavy	small	large	heavy
		Weight	1	2	3	4	5	6
takeoff	small	1	60	60	60	60	60	60
	large	2	60	60	60	60	60	60
	heavy	3	120	120	90	60	60	60
landing	small	4	75	75	75	82	69	60
	large	5	75	75	75	131	69	60
	heavy	6	75	75	75	196	157	96

rows: leading aircraft columns: following aircraft

For completeness of the solutions in terms of the optimal objective values, below we provide the constraint optimisation model.

$$\text{find } \forall_{j\in[1,N]} t_j \in [r_j, l_j] \qquad \text{ready time}$$

$$\text{and } \forall_{j\in[1,N]} \rho_j \in [1, M] \qquad \text{runway assigned}$$

$$\text{minimising TWT} = \sum_{j=1}^{N} w_j(t_j - r_j) \qquad \text{objective function}$$

$$\text{subject to } \forall_{j\in[1,N]}(t_j \in [r_j, l_j]) \qquad \text{time windows}$$

$$\forall_{i\in[1,M]}\forall_{j\neq j':\rho_j=\rho_{j'}\wedge t_j<t_{j'}}(t_{j'} - t_j > s(w_j, w_{j'})) \qquad \text{separation times}$$

With the complete model described above, in this work, we add runway capacity constraints as soft constraints. Arguably in theoretically optimal solutions, distributions of aircraft operations over runways will be closed to uniform rather than being very highly skewed. Moreover, for various practical reasons, a (nearly) uniform distribution of aircraft operation over the runways is desired. So the soft constraint in this case is to keep C_i the number of aircraft allocated to each runway i in the range $[\lfloor \frac{N}{M} \rfloor - \nu, \lceil \frac{N}{M} \rceil + \nu]$, where v is a given parameter. This soft constraint might help save considerable amount of search. Moreover, different combination of capacity limits on the runways could be tried in parallel and the best result obtained could be returned as the final result.

Time Window Constraints

When an aircraft j enters the radar range for landing or departure, the air traffic controller assigns a runway to it and a start time for landing/takeoff. That start time *must*

be within a specified *time window*, bounded by the *earliest* and *latest* landing/takeoff time e_j and l_j i.e. $t_j \in [e_j, l_j]$. The earliest time is calculated from the minimum time needed for the aircraft to approach the runway from its current position using its maximum speed. The latest time is based on the longest time the aircraft can keep flying depending on its operational and service level constraints. Also, each landing or takeoff aircraft has a *planned operating time*, which is also known as the *ready time* r_j. Usually, a penalty is incurred if the aircraft is scheduled to operate before or after its ready time. The $[e_j, l_j]$ time window has been used previously by [13, 21, 129, 130]. However, this time window is more theoretical since scheduling an aircraft before its ready time is usually considered to be not practical [131]. This is because operating an aircraft before its ready time might have security risks and for a landing aircraft, it might need more fuel to accelerate beyond its optimal cruise speed [132]. In other words, an aircraft is normally not allowed to be operated ahead of its ready time. Consequently, the practical time window $[r_j, l_j]$ is used in aircraft sequencing. In fact, this practical time window has recently attracted much more attention [132–135]. In this research, we use this practical time window.

Separation Time Constraints

Each aircraft performing landing or takeoff creates wake turbulence that subsequent aircraft on the same runway needs to avoid. For this, a certain minimum period of separation time is required between any two aircraft operating on the same runway. These separation times depend on the operation types and the aircraft classes (e.g. heavy, large, and small). However, to ensure certain standard safety levels, the exact separation times that are to be enforced are determined by appropriate aviation authorities such as Federal Aviation Administration (FAA) in the United States or Civil Aviation Authority (CAA) in the United Kingdom. Table 4.1 shows the separation times introduced by the FAA [136] and we use these in this work.

Assume two aircraft $j \neq j'$ operate on the same runway and aircraft j operates before aircraft j' i.e. $\rho_j = \rho_{j'} \wedge t_j < t_{j'}$. Therefore, we need to ensure $t_{j'} - t_j \geq s(w_j, w_{j'})$, where $s(w_j, w_{j'})$ is the minimum separation time for aircraft j' with weight $w_{j'}$ from aircraft j with weight w_j. Given the weights w_j and $w_{j'}$ of the aircraft j and j' respectively, the value of $s(w_j, w_{j'})$ could be found at row w_j and column $w_{j'}$ of Table 4.1.

Although the separation time constraints must hold between every two aircraft operating on the same runway, [137] showed that using the FAA standard depicted in Table 4.1, these constraints are satisfied automatically when there are at least three aircraft in between two given aircraft, all operating on the same runway. We demonstrate this in Figure 4.1. In our implementation, we therefore, use this finding.

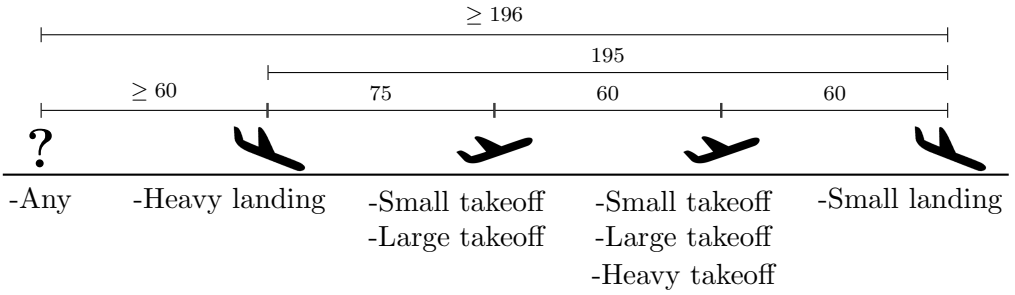


FIGURE 4.1: When separation times in FAA standards could be automatically satisfied

Minimising Total Tardiness

As described before, each aircraft has an optimal scheduled operation time, known as its ready time. This implies performing the operation at that time has no delay and no extra fuel burn. However, the time separation constraints could mean that some flights cannot operate at their ready times. Therefore, the scheduled times for some flights are different from their optimal times. Moreover, if an aircraft j operates after its ready time r_j, its tardiness is $(t_j - r_j)$ which becomes $w_j(t_j - r_j)$ if the aircraft class operation weight w_j is considered. The weight of an aircraft depends on the operation type (e.g. landing and takeoff) and the class of the aircraft (e.g. heavy, large, and small). A landing aircraft has greater priority than a departing aircraft due to higher average fuel burning and safety measures. Departing aircraft can wait at the ground with lower risk. On the other hand, heavier aircraft get more weights than the lighter ones, again based on higher average fuel burning and safety measures. In this chapter, we use the weights shown in Figure 4.1 as these appear in the benchmark problems [2] that we use in our experiments. Note that operation types are indirectly encoded in the weights (e.g. smaller weights for takeoff and larger weights for landing) and hence are not explicitly specified in the problem description.

Overall, the objective function of the ASP is to minimise the cost resulting from the deviation of the start times from the respective ready times. In this chapter, we use the total weighted tardiness of a schedule TWT $= \sum_{j=1}^{N} w_j(t_j - r_j)$. This objective allows not only reducing delays, but also maximising runway usage, thus reducing congestion at airports [131]. Also, this objective function can be referred as the total excess fuel cost, as if all start times t_j equal their associated ready times r_j in a given schedule, then no excess fuel cost is incurred [2].

Runway Capacity Constraints

[138] mentioned that runways are the main constrained resource and thus the source of delays in all airport systems. For airports with multiple runways, the distribution of flights over runways could be uniform. Given uniform distributions of types of operation, types of aircraft, and times of operation, an equal distribution of flights over runways ideally will achieve the optimum solutions in terms of performance. An equal distribution will also improve the longevity and the maintenance of the runways. However, there are plenty of other factors that need to be considered as well.

One such important factor is noise [139]. According to Single European Sky ATM Research (SESAR), the two main environmental issues associated with aviation are emissions and noise [140]. Each runway has specific paths for landing and takeoff. So using mostly one runway in an airport will cause human health problems such as hearing loss, communication interference, sleep interference, anxiety and stress, for those who live in the vicinity of that runway [141–143]. Noise pollution these days become more crucial with the rise of populations in cities and their territorial expansion, particularly when residential areas become closer to airports. An equal distribution of flights over runways thus also addresses this noise issue. Now the question is how to achieve the equal distribution.

The Heathrow Airport, London, has two runways, that are used in a segregated fashion, i.e. one runway for landings and the other for takeoffs. The reason is that the number of arrival and departure flights are almost the same. However, they are using a Runway Alternation rule, which can be described as Half Day Noise Relief [144]. Although takeoffs need much higher engine thrust than landings, landing noise is frequently more annoying to the ear because of dominant fan noise [145]. So, to share the disturbance

and to give everyone periods of relief from aircraft noise, based on Runway Alternation mechanism, aircraft lands on one runway between 07.00-15.00, and on the other between 15.00-23.00. The morning/afternoon rota changes weekly, on Mondays [144]. So, as can be observed, the distribution and allocation of aircraft over runways are very important.

While obtaining an uniform distribution of flights over the runways is a very desired objective, for a given ASP instance, the optimal solution in terms of total tardiness may not have an equal number of flights allocated to each runway. To achieve a balance to a certain extent, in this thesis, we consider an upper bound U and a lower bound L on the numbers of flights that could be allocated to each runway. These bounds are computed from $\frac{N}{M} \pm \nu$, where ν is a given number. These bounds essentially also eliminates many solutions that are very suboptimal. It is worth noting that these runway capacity constraints are at the end soft constraints, whose violation could be tolerated. Note that for a given ν, there could be a large number of runway capacity combinations. For example with $N = 15, M = 3, \nu = 1$, possible combinations are $\langle 5, 5, 5 \rangle$ where all capacities are equal and $\langle 4, 5, 6 \rangle$ and its permutations where variations are allowed. Eliminating symmetries to improve search, we can however only consider $\langle 5, 5, 5 \rangle$ and $\langle 4, 5, 6 \rangle$. In this study, we propose to perform separate search with each combination, perhaps in parallel, and return the best solution found. We therefore do not strictly include this as a constraint in our model. For convenience, henceforth, we describe our algorithms and provide theoretical analysis using $\lceil \frac{N}{M} \rceil$ as the maximum number of flights per runway.

4.2.2 Related Works

Extensive overviews of aircraft scheduling are available in several articles [146–148]. [133] divided the air traffic flow management (ATFM) literature into three classifications. The first classification is based on the traffic control between airports [149, 150] and the traffic control in the terminal manoeuvring area (TMA) of an individual airport [133, 134, 151]. In this chapter, we focus on the traffic control in the TMA of an individual airport.

The second classification is based on the type of information: static [22, 23, 152] and dynamic [153–156]. In the static case, it is assumed that all data such as ready times, and time windows are known upfront and can be taken into account in the process. However, in the dynamic case, all these data become known only when an aircraft is

ready to land or depart. In this research, the static case of the ASP is considered. Note that although modelling approaches for dynamic case tend to be quite different from the ones for the static case, researchers working on these problems still are attracted to the static case. This is because a static approach can be inserted in a dynamic system that iteratively solves an ASP problem with static information available in each of a series of sliding time windows [133].

The third classification is based on the algorithmic approaches and is the main focus of this chapter. Since landing is more crucial than takeoff, some of the algorithms only consider landing operations and thus solve the aircraft landing problem (ALP). Comparatively, only a few algorithms consider aircraft takeoff problem (ATP) although ATP is regarded as more difficult to solve compare to the ALP [157]. Other algorithms however solve the aircraft sequencing problem (ASP) considering both landing and takeoff operations. Nevertheless, in the literature of these problems, several MIP formulations [129, 158–162] are proposed that are solved with exact solvers; branch-and-bound (B&B) algorithms [135, 152, 158, 163–165], and dynamic programming (DP) approaches [166–170]. However, as already mentioned, these problems are NP-hard. Thus, it is difficult to obtain optimal solutions for a problem in a reasonable time. Consequently, it becomes necessary to develop qualified approximate solutions. Below we provide a wide but non-exhaustive review of the recent literature focussing on static cases of ALP, ATP, and ASP separately.

Aircraft Landing Problem (ALP)

The single-runway ALP studied by [152] and pointed out that it could be extended to multiple-runway case. They proposed a genetic algorithm for these two cases considering the penalty costs for landing before and after ready times as the objective function. As another method, two local search-based algorithms are proposed [171]: hill climbing (HC), and simulated annealing (SA) for single-runway ALP. They compared these two algorithms with exact algorithms such as constraint programming (CP) approaches on benchmarks up to 123 aircraft. The results showed the efficiency of SA and HC algorithms compared to the exact methods. In addition, the SA method obtained better solutions although HC was faster.

The multiple-runway ALP studied by [22] and a population-based metaheuristic, scatter search (SS), is proposed, in order to achieve effective runway utilisation, where two different objective functions (a non-linear and a linear) were used during the experiments. As another algorithm, a new GA with a new representation designed by [172]. They showed that the new representation make it possible to easily adopt a uniform crossover operator. The experimental results indicated the efficiency of the new GA against common GAs. Also, a hybrid algorithm is proposed by [173] that integrated bee evolutionary genetic algorithm with modified clustering method (named BEGA-CM) for solving single-runway ALP. They compared the BEGA-CM algorithm with GA and showed that their proposed algorithm by far faster than GA.

[174] studied multiple-runway ALP and formulated it as a Job Shop Scheduling Problem (JSSP) based on a graphical representation. Then proposed a hybrid algorithm called ACOGA, combination of ant colony optimization (ACO) with GA, and showed the efficiency of proposed algorithm against GA. [21] studied multiple-runway ALP and presented two algorithms hybridising SA with Variable Neighbourhood Search (VNS) and Variable Neighbourhood Descent (VND), namely SA-VNS and SA-VND. They compared their algorithms with SS and BA [22] and the results indicated the effectiveness of SA-VNS and SA-VND against other algorithms compared.

Recently [13] proposed an Iterated Local Search (ILS) for single and multiple-runway ALP made up of an VND algorithm as local search, multiple operators for perturbation with time-varying perturbation strength. They compared their algorithm with SA-VNS [21] and SS [22] and their results showed the effectiveness of ILS. [23] investigated single and multiple runway cases of the ALP and proposed a hybrid particle swarm optimisation algorithm (PSO) in a rolling horizon approach. The objective function was the same as is used by [22]. In order to test their algorithm, they used a set of well-known benchmark instances containing up to 500 aircraft and 5 runways, and showed that their algorithm obtained better solution than SA-VNS and SA-VND [21] and SS and BA [22].

Aircraft Take-Off Problem (ATP)

This problem has attracted less attention compared to ALP. ATP not only consists of more operational constraints but also is heavily related to taxi-out scheduling problem, so that they need to be combined each other. The ATP at London Heathrow Airport

has been investigated [175] which has a single runway to be used for departures with the objective of maximising the runway throughput. They developed different metaheuristics such as steeper descent(SD), tabu search (TS), and SA to solve the problem. The results showed that the TS is better than the others with a small margin. A greedy algorithm and a 2-interchange heuristic algorithm for ATP is proposed [176]. After testing proposed methods on randomly generated datasets, it was reported that both methods are better than an FCFS method, and 2-interchange heuristic outperforms the greedy algorithm.

Aircraft Scheduling Problem (ASP)

This problem is also known as integrated landing and takeoff problem. Three greedy algorithms and two metaheuristics are proposed [15]: SA and Metaheuristic for Randomized Priority Search (Meta-RaPS) to solve multiple-runway ASP. They tested the algorithms on 55 instances based on Doha International Airport (DIA) generated by [2]. Then, It has been showed that ASP can be viewed as a job shop scheduling problem [177] with additional practical constraints such as holding circle constraints, and blocking constraints for runways. Holding circle constraint is defined and used by air traffic controllers to show the speed and path of landing flights needed to wait in air when terminal area is congested [177]. The runway blocking constraints is appears [178] when because of presence of one aircraft on a runway, other aircraft cannot use that runway. They proposed several heuristics and tested them on instances obtained from the Roma Fiumicino airport, Italy. The results showed the efficiency of the heuristics against the FCFS method. Next, a TS algorithm is proposed [179] for multiple-runway ASP that includes a hybrid neighbourhood structure. The results showed the ability of this algorithm to find good solutions in reasonable computing times although when compared with the SA [15], it had worse performance than SA.

As can be seen, although the number of articles on the static case of ASP is still small, most of them are focused on exact methods [133–135, 161, 180]. However, as mentioned before, ASP being NP-hard, exact methods are not the reasonable choices particularly when practical-sized problem instances are considered. So, we propose constraint optimisation based local search algorithms that are both effective and simple to implement. We then compare our algorithms with two best local search-based metaheuristic that

are proposed for the ASP and the related problems. The first one is an SA algorithm by [15] for ASP. The second one is adapted to ASP from the ILS-based best known ALP algorithm by [13]. It is noteworthy again that neither the SA nor the ILS algorithm uses problem specific structural knowledge from the constraint or the objective functions while our proposed search methods do. While local search algorithms share many aspects with each other, yet certain key aspects make them unique on their own right. Our proposed algorithms with the exploitation of problem specific structural knowledge is clearly and conceptually separable from the existing local search methods. Below we briefly describe the SA and the ILS algorithms.

The Simulated Annealing Approach

The SA algorithm proposed by [15] starts from an initial solution. Next, in a loop, it performs several steps that are typical in a simulated annealing algorithm. It first explores the neighbourhood of the current solution using swap operators. Then, it selects the best neighbouring solution as the new solution. If the new solution is better than the current solution, the new solution is accepted. If the new solution is worse than the current solution, it could still be accepted with some probability. This acceptance probability diminishes with the increase of loop iterations. For initialisation, the SA algorithm proposed three heuristics: Earliest Ready Time (ERT), Fast Priority Index (FPI), and Adapted Apparent Tardiness Cost with Separation and Ready Times (AATCSR). The experimental results showed that the AATCSR performed better than the other two. In AATCSR, starting from an empty solution, every time an unscheduled aircraft with the largest index obtained from a complex formula is appended.

The Iterated Local Search Approach

The ILS algorithm proposed by [13] starts with an initial feasible solution followed by local search. Then, it iteratively invokes the perturbation phase, the local search phase, and the acceptance phase. To generate an initial solution, it uses a randomised greedy heuristic. For perturbation, it randomly performs either swap or insertion for a number of times, but the number of times the operators are used decreases with the number of ILS iteration completed. For local search, it uses a variable neighbourhood descent (VND) algorithm [44] with a number of swap and insertion operators. For acceptance

criteria, if the solution returned by the local search is better than the current solution, it is always accepted; otherwise, it is accepted with a small probability.

4.2.3 Our Solution Representation

A typical ASP solution representation could consider directly assigning a specific time to each aircraft. As in many other optimisation search problems, the ASP decision variables could thus be the time variables taking values from the time horizon. However, in ASP, the weighted tardiness of an aircraft specifically depends on how delayed its operation is from its ready time. Considering the minimisation of the weighted tardiness, each aircraft's starting time therefore can be determined from the preceding aircraft's starting times and the separation times between the aircraft and those aircraft. These specific ASP facts that are directly encoded in the constraints and the objective functions have an influence on the way ASP solutions could be represented. For instance, one can represent ASP solutions by using sequences of aircraft assigned to the runways. The search in this case will be on the permutations of the given aircraft with a view to minimising the total weighted tardiness.

Assume $\pi = \langle \vec{1}, \vec{2}, \ldots, \vec{M} \rangle$ be an ASP solution where $\vec{i}$ denotes the sequence of aircraft that will operate on runway i. Further, assume $|\vec{i}|$ denotes the number of such aircraft, and $i[k]$ denotes such an aircraft at position k. The lower the position k, the earlier the start time $t_{i[k]}$. Below is an example solution for an ASP with 7 aircraft and 3 runways. Aircraft 3, 7 are scheduled to runway 1 in the order. Similarly, aircraft 4, 1, 5 are scheduled to runway 2 and aircraft 6, 2 to runway 3.

$$\begin{cases} 3, 7 \\ 4, 1, 5 \\ 6, 2 \end{cases}$$

For convenience of theoretical analysis and algorithm description, we assume each runway has a capacity of $\lceil N/M \rceil$ aircraft. However, as mentioned before, in our implementation and experiments, we use upper and lower bounds on the runway capacities to produce a number of combinations and run our algorithm for each combination.

Calculating Total Tardiness

Assume a runway i with the sequence $\vec{i}$ of aircraft scheduled to operate on. As is mentioned before, since the tardiness $t_{i[k]} - r_{i[k]}$ of a aircraft $i[k]$ in the sequence $\vec{i}$ is calculated using its ready time $r_{i[k]}$ as the reference, the earlier we can schedule the aircraft $i[k]$'s start time, the lesser is the cost. We therefore can directly calculate the start time $t_{i[k]}$ of the aircraft $i[k]$ from the start times of the previous aircraft in the sequence $\vec{i}$ of the same runway i. As is already shown in Figure 4.1 and by [137], we just have to use the start times of at most the previous 3 aircraft.

$$t_{i[1]} = r_{i[1]} \tag{4.1}$$

$$t_{i[2]} = \max\{r_{i[2]}, t_{i[1]} + s(w_{i[1]}, w_{i[2]})\} \tag{4.2}$$

$$t_{i[3]} = \max\{r_{i[3]}, t_{i[2]} + s(w_{i[2]}, w_{i[3]}), t_{i[1]} + s(w_{i[1]}, w_{i[3]})\} \tag{4.3}$$

$$\begin{aligned} t_{i[k]} = \max\{&r_{i[k]}, t_{i[k-3]} + s(w_{i[k-3]}, w_{i[k]}), t_{i[k-2]} + s(w_{i[k-2]}, w_{i[k]}), \\ &t_{i[k-1]} + s(w_{i[k-1]}, w_{i[k]})\} \qquad \forall k \in [4, |\vec{i}|] \end{aligned} \tag{4.4}$$

Using the above formulas, once we obtain the start times of each aircraft in each runway, we can calculate the total weighted tardiness of a given schedule from the formula shown below.

$$\text{TWT} = \sum_{i=1}^{M} \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]}) \tag{4.5}$$

For convenience, we also define $\text{TWT}(i) = \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]})$ to be the total weighted tardiness for runway i and $\text{WT}(j) = w_j(t_j - r_j)$ to be the weighted tardiness for aircraft j.

An Example

Consider an example with 8 aircraft and 2 runways. The required data are given in Table 4.2. One possible solution is shown in Figure 4.2 where aircraft 2, 4, 1, 7 in the order are assigned to runway 1. Also, aircraft 5, 8, 3, 6 in the order are assigned to

runway 2. Using equations (4.1)–(4.8), the start times of the aircraft are obtained as follows:

TABLE 4.2: An example problem instance with 8 aircraft and 2 runways

j	1	2	3	4	5	6	7	8
r_j	46	93	117	135	229	250	256	308
d_j	646	693	717	735	829	850	856	908
w_j	1	2	4	6	4	1	1	4

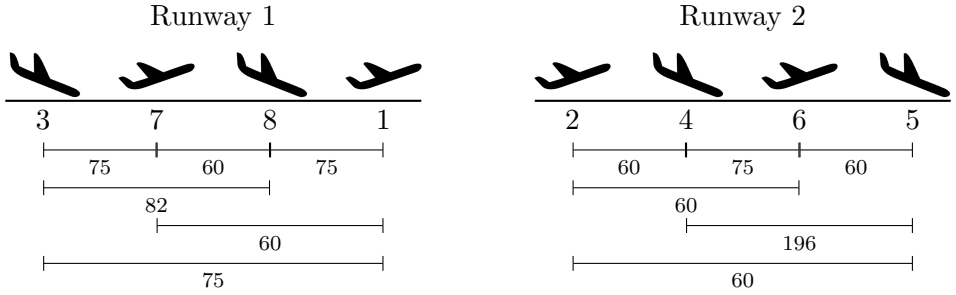


FIGURE 4.2: An example schedule for 8 aircraft and 2 runways

- Runway 1:

$$t_{1[1]} = r_{1[1]} = r_3 = 117$$

$$\begin{aligned} t_{1[2]} &= \max\{r_{1[2]}, t_{1[1]} + s(w_{1[1]}, w_{1[2]})\} \\ &= \max\{r_7, t_3 + s(w_3, w_7)\} \\ &= \max\{256, 117 + 75\} = 256 \end{aligned}$$

$$\begin{aligned} t_{1[3]} &= \max\{r_{1[3]}, t_{1[2]} + s(w_{1[2]}, w_{1[3]}), t_{1[1]} + s(w_{1[1]}, w_{1[3]})\} \\ &= \max\{r_8, t_7 + s(w_7, w_8), t_3 + s(w_3, w_8)\} \\ &= \max\{308, 256 + 60, 117 + 82\} = 316 \end{aligned}$$

$$\begin{aligned} t_{1[4]} &= \max\{r_{1[4]}, t_{1[3]} + s(w_{1[3]}, w_{1[4]}), t_{1[2]} + s(w_{1[2]}, w_{1[4]}), t_{1[1]} + s(w_{1[1]}, w_{1[4]})\} \\ &= \max\{r_1, t_8 + s(w_8, w_1), t_7 + s(w_7, w_1), t_3 + s(w_3, w_1)\} \\ &= \max\{46, 316 + 75, 256 + 60, 117 + 75\} = 391 \end{aligned}$$

- Runway 2:

$$t_{2[1]} = r_{2[1]} = r_2 = 93$$

$$\begin{aligned} t_{2[2]} &= \max\{r_{2[2]}, t_{2[1]} + s(w_{2[1]}, w_{2[2]})\} \\ &= \max\{r_4, t_2 + s(w_2, w_4)\} \\ &= \max\{135, 93 + 60\} = 153 \end{aligned}$$

$$
\begin{aligned}
t_{2[3]} &= \max\{r_{2[3]}, t_{2[2]} + s(w_{2[2]}, w_{2[3]}), t_{2[1]} + s(w_{2[1]}, w_{2[3]})\} \\
&= \max\{r_6, t_4 + s(w_4, w_6), t_2 + s(w_2, w_6)\} \\
&= \max\{250, 153 + 75, 93 + 60\} = 250
\end{aligned}
$$

$$
\begin{aligned}
t_{2[4]} &= \max\{r_{2[4]}, t_{2[3]} + s(w_{2[3]}, w_{2[4]}), t_{2[2]} + s(w_{2[2]}, w_{2[4]}), t_{2[1]} + s(w_{2[1]}, w_{2[4]})\} \\
&= \max\{r_5, t_6 + s(w_6, w_5), t_4 + s(w_4, w_5), t_2 + s(w_2, w_5)\} \\
&= \max\{229, 250 + 60, 153 + 196, 93 + 60\} = 349
\end{aligned}
$$

From the start times of the aircraft obtained above, we then calculate the total weighted tardiness in the following way:

$$
\begin{aligned}
\text{TWT} = \textstyle\sum_{i=1}^{M} \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]}) &= (4 \times (117 - 117)) + (1 \times (256 - 256)) + (4 \times \\
(316 - 308)) + (1 \times (391 - 46)) + (2 \times (93 - 93)) &+ (6 \times (153 - 135)) + (1 \times (250 - 250)) \\
+ (4 \times (349 - 229)) &= 965
\end{aligned}
$$

Lemma 4.1. *Computing TWT from scratch using Equations (4.1)–(4.5) has a time complexity of $O(N)$ and a space complexity of $O(N)$ as well.*

Proof. For convenience of the worst case analysis, we assume Equation 4.8 to be the generalised case for all aircraft. Clearly, computation of $t_{i[k]}$ from Equation 4.8 needs $O(1)$ time and $O(1)$ space. Thus, for N aircraft, computing the ready times needs $O(N)$ time and $O(N)$ space. In Equation 4.5, we compute weighted tardiness for each aircraft. So the total time is $O(N)$ and space is $O(1)$. Hence, computing TWT has a time complexity $O(N)$ and a space complexity $O(N)$. □

4.2.4 Proposed Constructive Search Method

Our objective is to minimise total weighted tardiness TWT of a schedule. For each given aircraft, its weighted tardiness depends on how early it can be scheduled and is zero if it is scheduled at its ready time. *In our constraint-guided constructive search, we exploit the weighted tardiness criterion explicitly by selecting the next aircraft based on its ready time and then scheduling it at its best position in the partial solution already constructed.* For this the next aircraft is scheduled at each possible position in a given partial solution with k aircraft and each potential partial solutions with $k + 1$ aircraft are evaluated. Our constructive method is thus search based and is guided by the objective function while existing constructive methods, e.g. in [15], are mere appending of the next aircraft

using a predefined criterion, which is not the objective function directly. Although our final constructive search method uses swap operators, we also implement an insertion operator based one. We call our approaches swap and insertion based heuristics (SBH and IBH) respectively and describe them below.

Swap Based Heuristic (SBH)

In our SBH method described in Algorithm 18, we first arrange all the aircraft in an increasing order of their ready times and get a list θ. Next, we take the first M aircraft from θ and assign each of them to a different runway to obtain an initial solution π. Then, we consider an iteration for each aircraft in θ. In each iteration, the next aircraft in θ is appended to the list of aircraft in all the runways. This produces M new solutions. Assume π_i is one such new solution and the aircraft was added to the ith runway. Now for each π_i, the last aircraft in the ith runway is swapped with all other aircraft in the same runway to obtain further newer solutions. Among all the new and newer solutions, the best one is selected for the next iteration.

Algorithm 18 SBH

1. Arrange all aircraft in the given problem in ascending order of their ready times and put them in a list θ. We use θ_k $(1 \leq k \leq N)$ to denote the aircraft at position k in θ.
2. Without loss of generality, let $\pi = \langle \vec{1}, \ldots \vec{M} \rangle$ be the initial solution where each $\vec{i}$ has one aircraft θ_i.
3. For each $k \in [M+1, N]$, take aircraft θ_k from θ
   (a) For each runway $i : |\vec{i}| < \lceil N/M \rceil$, append θ_k to the end of $\vec{i}$ of π to obtain a solution π_i. We have at most M new solutions.
   (b) For each solution π_i of the M solutions from the previous step, for each $1 \leq k' < |\vec{i}|$, swap $i[k']$ with the appended aircraft $i[|\vec{i}|]$ to obtain $|\vec{i}| - 1$. Thus, from this step, in total, we get $\sum_i (|\vec{i}| - 1) = k - 1$ new solutions
   (c) Let π be the best solution (in terms of weighted tardiness) from the $M + k - 1$ new solutions found in the previous two steps. Solution π is used in the next iteration of k
4. Return solution π.

For an example, assume an ASP in Table 4.3 with 5 aircraft and 2 runways. At first, based on the ready times, the initial sequence becomes $\theta = \langle 1, 2, 3, 4, 5 \rangle$. Aircraft 1 and 2 are first assigned to runway 1 and 2 respectively. Note that the runways are identical here.

TABLE 4.3: An example problem with 5 aircraft and 2 ruways

j	1	2	3	4	5
r_j	32	43	97	112	154
d_j	632	643	697	712	754
w_j	5	6	4	2	3

$$\begin{cases} 1 \\ 2 \end{cases}$$

Then, the aircraft at the third position of θ is 3. Appending 3 separately at the end of both runways and then performing swaps within the same runway, we obtain the following solutions:

$$\begin{cases} 1,3 \\ 2 \end{cases} \quad \begin{cases} 3,1 \\ 2 \end{cases} \quad \begin{cases} 1 \\ 2,3 \end{cases} \quad \begin{cases} 1 \\ 3,2 \end{cases}$$

The weighted tardiness for the above partial solutions are 0, 1110, 168 and 900 from left to right. Therefore, the first partial solution from above is selected. Next the subsequent aircraft 4 from the sequence θ needs to be considered for insertion into the selected partial solution.

$$\begin{cases} 1,3,4 \\ 2 \end{cases} \quad \begin{cases} 1,4,3 \\ 2 \end{cases} \quad \begin{cases} 4,3,1 \\ 2 \end{cases} \quad \begin{cases} 1,3 \\ 2,4 \end{cases} \quad \begin{cases} 1,3 \\ 4,2 \end{cases}$$

The weighted tardiness for the above 5 partial solutions are 120, 150, 1345, 12 and 774 from left to right. Therefore, the fourth partial sequence is selected for the next and final step. The fifth aircraft in θ is aircraft 5 which can be inserted into the selected partial solution in the following ways:

$$\begin{cases} 1,3,5 \\ 2,4 \end{cases} \quad \begin{cases} 1,5,3 \\ 2,4 \end{cases} \quad \begin{cases} 5,3,1 \\ 2,4 \end{cases} \quad \begin{cases} 1,3 \\ 2,4,5 \end{cases} \quad \begin{cases} 1,3 \\ 2,5,4 \end{cases} \quad \begin{cases} 1,3 \\ 5,4,2 \end{cases}$$

The weighted tardiness for the above partial solutions are from left to right 66, 480, 1735, 84, 324 and 2310. As a result, the first solution becomes the final solution; which will be used as the initial solution in the EPS.

Lemma 4.2. *The SBH algorithm shown in Algorithm 18 has a time complexity* $O(N^3)$ *and a space complexity* $O(N)$.

Proof. The first step in Algorithm 18 takes $O(N \lg N)$ time and $O(N)$ space to sort the aircraft. The second step needs $O(M)$ time and $O(M)$ space. For the worst case analysis, let us ignore the condition in $i : |\vec{i}| < \lceil N/M \rceil$ and try all runways. The worst case appears when each subsequent aircraft is placed in the runway that is the next one in a round robin fashion. Therefore, for a given $k \in [M+1, N]$, we generate $M + k - 1$ new partial solutions each having k aircraft. If we compute the TWT from scratch for every partial solution, it will take $O(k)$ time and $O(k)$ space. Since in each swap, aircraft only in one runway get affected, we can however compute TWT incrementally in $O(\lceil N/M \rceil)$ time and $O(\lceil N/M \rceil)$ space. So the overall time complexity becomes $O(N \lg N + M + \sum_{k=M+1}^{N}(M + k - 1)k)$ when TWT is computed form scratch and $O(N \lg N + M + \sum_{k=M+1}^{N}(M + k - 1)\lceil N/M \rceil)$ when TWT is computed incrementally. Simplifying, the time complexity comes to $O(N^3)$ assuming $N \gg M$. The overall space complexity however remains $O(N)$. □

Insertion Based Heuristic (IBH)

In our IBH method described in Algorithm 19, we first arrange all the aircraft in an increasing order of their ready times and get a list θ. Next, we take the first M aircraft from θ and assign each of them to a different runway to obtain an initial solution π. Then, we consider an iteration for each aircraft in θ. In each iteration, the next aircraft in θ is inserted in all possible positions in all the runways π and from the new solutions, the best one (in terms of weighted tardiness) is selected for the next iteration.

Lemma 4.3. *The IBH algorithm shown in Algorithm 19 has a time complexity* $O(N^3)$ *and a space complexity* $O(N)$.

Proof. The proof is very similar to that of Lemma 4.2. □

4.2.5 Proposed Local Search Method

Local search involves generation of neighbouring solutions around the current solution and evaluation of the neighbouring solutions. From the current solution, local search

Algorithm 19 IBH

1. Arrange all aircraft in the given problem in ascending order of their ready times and put them in a list θ. We use θ_k $(1 \leq k \leq N)$ to denote the aircraft at position k in θ.
2. Without loss of generality, let $\pi = \langle \vec{1}, \ldots \vec{M} \rangle$ be the initial solution where each $\vec{i}$ has one aircraft θ_i.
3. For each $k \in [M+1, N]$, take aircraft θ_k from θ
   (a) For each runway $i : |\vec{i}| < \lceil N/M \rceil$, insert θ_k in all possible positions k' of $\vec{i}$ of π to obtain solutions $\pi_{i,k'}$, where $1 \leq k' \leq \hat{n}+1$ and $\hat{n} = |\vec{i}|$ before inserting θ_k. Thus we have $\sum_i (\hat{n}+1)$ new solutions
   (b) Let π be the best solution (in terms of weighted tardiness of each $\pi_{i,k'}$) from the new solutions found above (if any). Solution π is used in the next iteration of k
4. Return solution π.

then moves to one of the neighbouring solution, preferably to the best one as per given criteria. A typical local search algorithm would consider the neighbour generation and evaluation procedures to be independent of each other. As such, it would generate the neighbouring solutions randomly or exhaustively, so not much purposefully, although using predefined neighbourhood generation operators.

A more effective approach would be to be selective in neighbourhood generation either to save the effort required to explore the unpromising areas of the search space or to quickly take the search to the promising areas. In the constraint optimisation paradigm in artificial intelligence, neighbourhood generation and evaluation are considered to be part of an integrated process. So the evaluation functions i.e. the constraints and the objective functions are used more purposefully in the generation of the neighbourhoods. The current solution is analysed to identify the problematic part in terms of the constraint violation or the objective value and neighbourhood solutions are generated to improve that part of that solution. In our proposed local search method for ASP, we will design neighbourhood operators, even if those are swap and inserts they will be customised, such that unpromising neighbouring solutions are generated as few as possible, and we will apply the neighbourhood operators on runways that have the most weighted tardiness i.e. part of the problematic part of the solution. To test the effectiveness of our constraint guided strategies, we implement them on top of a generic explorative perturbative search (EPS) method shown in Algorithm 20. Note that our contribution

is not in the EPS algorithm itself, rather in the difference that we make in terms of its performance when we use our constraint guided strategies within the EPS framework.

Algorithm 20 Explorative Pertubative search (EPS)

1: $\pi \leftarrow$ generateInitialSolution() // IBH or SBH.
2: **while** termination criteria not satisfied **do**
3: $\quad \pi' \leftarrow$ performExploration(π).
4: $\quad \pi \leftarrow$ acceptAndUpdateBest(π', π).
5: $\quad \pi \leftarrow$ performPerturbation(π).
6: **return:** the global best solution found so far.

At first Algorithm 20 obtains a heuristically constructed solution by using our proposed constructive method. Within a loop, it then performs explorative local search to find a new improving solution. With certain acceptance criteria, the new solution is then accepted as the current solution or is discarded. Inside the loop, the current solution is then perturbed, meaning partially destroyed and then the destroyed part is constructed again, to obtain a new solution. The new solution becomes the current solution for the next iteration. The perturbation in effect serves the purpose of a partial restart to take the search out of a local optimum or a plateau. It is worth mentioning that in all phases of our algorithms, feasibility of a solution is checked as soon as the solution is generated and any infeasible solution is immediately discarded.

Performing Exploration

The kinds of move used in a local exploration play a significant role in its effectiveness. Given the permutation type representation of the ASP solutions, the typical search operators are swap and insert moves e.g. in travelling salesman problem [181] and in vehicle routing problem [182]. In general, the kinds of insert and swap moves used in the literature are inter-insert, intra-insert, inter-swap, and intra-swap. The inter-insert operator removes one aircraft from a runway and inserts it into another runway. The intra-insert operator removes one aircraft from a runway and inserts it into the same runway. The inter-swap performs swaps of aircraft pairs where two aircraft are in two different runways. The intra-swap performs swaps of aircraft pairs where two aircraft are from the same runway.

In this thesis, instead of the intra-swap, we use another type of swap called adjacent swap that performs swapping of two adjacent aircraft in the same runway; in most cases

the ready times of the two aircraft do not change by a very large margin making it preferable to intra-swap. With intra-swap operators, most of the new solutions become infeasible when the two aircraft to be swapped are far apart. Adjacent swap will generate fewer new solutions than intra-swap but most of the new solutions will be feasible. For the same reason, intra-insert operator is not preferable either. However, adjacent insert is the same as the adjacent swap. *Note that the consideration of the adjacent swap or adjacent insert is based on the analysis of the constraints here and is part of our constraint guided strategies.*

As we will discuss later in Section 4.2.5, we perform ASP search on a given combination of runway capacities. Hence, we do not use inter-insert operator in this work. Below we describe the exact implementation of all the insert and swap operators.

- **Inter-Swap Operator:** Given a solution π, one runway i with the most weighted tardiness and another random runway $i' \neq i$ are selected and then each aircraft $i[k]$ in π is then swapped with each aircraft $i'[k']$ in π to obtain $|\vec{i}| \times |\vec{i'}|$ new solutions. Only the best new solution is however returned. Algorithm 21 provides the pseudo-code of the inter-swap operator and Figure 4.3 (a) shows an example. Clearly, Algorithm 21 runs in $O(\lceil N/M \rceil^3)$ time and $O(N)$ space when TWT is computed incrementally.

Algorithm 21 Inter-swap operator

1. Let π be the current solution given as a parameter
2. Select a runway $i = \text{argmax}_i \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]})$ and another random runway $i' \neq i$ randomly.
3. For each $k \in [1, |\vec{i}|]$ for each $k' \in [1, |\vec{i'}|]$, every time in π, swap aircraft $i[k]$ and $i'[k']$ to obtain a new solution.
4. From the $|\vec{i}| \times |\vec{i'}|$ new solutions, return the best one (in terms of weighted tardiness)

- **Adjacent-Swap Operator:** Given a solution π, two runways $i' \neq i''$ are first chosen uniformly randomly. Then for each runway $i \in \{i', i''\}$, for each two adjacent aircraft $i[k]$ and $i[(k+1)]$ in $\vec{i}$ of π are swapped with each other to obtain $|\vec{i}|-1$ new solutions. Only the best new solution is however returned. Algorithm 22 provides the pseudo-code of the adjacent-swap operator and Figure 4.3 (b) shows an example. Note that we use two runways here although the swapping of aircraft are done

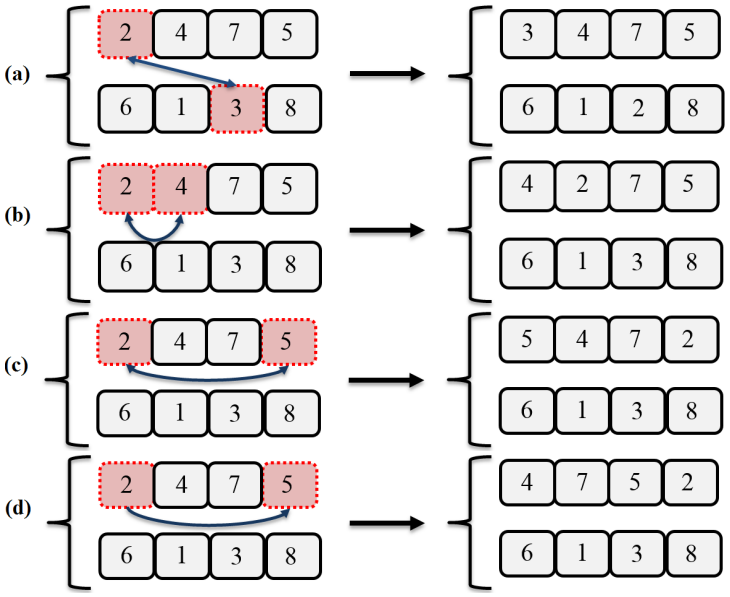


FIGURE 4.3: Operators: (a) inter-swap, (b) adjacent-swap, (c) intra-swap, (d) intra-insert

within a single runway. This is just to match with the inter-swap operator that involves two runways every time. Also, note that we select both runways randomly because the runway with the highest weighted tardiness is already considered in the perturbation phase. Clearly, Algorithm 22 runs in $O(\lceil N/M \rceil^2)$ time and $O(N)$ space when TWT is computed incrementally.

Algorithm 22 Adjacent-swap operator

1. Let π be the current solution given as a parameter
2. Randomly choose two runways i' and i'' from all runways
3. For each $i \in \{i', i''\}$, for each $k \in [1, |\vec{i}|)$, every time in π, swap aircraft $i[k]$ and $i[k+1]$ to obtain a new solution.
4. From the $|\vec{i'}|+|\vec{i''}|-2$ new solutions, return the best one (in terms of weighted tardiness)

- **Intra-Swap Operator:** Given a solution π, two runways $i' \neq i''$ are first chosen uniformly randomly. Then for each runway $i \in \{i', i''\}$, each pair of aircraft $i[k]$ and $i[k']$ in $\vec{i}$ of π are swapped with each other to obtain $|\vec{i}| \times (|\vec{i}|-1)$ new solutions. Only the best new solution is however returned. Algorithm 23 provides the pseudo-code of the intra-swap operator and Figure 4.3 (c) shows an example. Note that we use two runways here although the swapping of aircraft are done within a single runway. This is just to match with the inter-swap operator that involves two runways every

time. Also, note that we select both runways randomly because the runway with the highest weighted tardiness is already considered in the perturbation phase. Clearly, Algorithm 23 runs in $O(\lceil N/M \rceil^3)$ time and $O(N)$ space when TWT is computed incrementally.

Algorithm 23 Intra-swap operator

1. Let π be the current solution given as a parameter
2. Randomly choose two runways i' and i'' from all runways
3. For each $i \in \{i', i''\}$, for each $k, k' \in [1, |\vec{i}|)$, every time in π, swap aircraft $i[k]$ and $i[k']$ to obtain a new solution.
4. From the $|\vec{i'}| \times (|\vec{i''}| - 1)$ new solutions, return the best one (in terms of weighted tardiness)

- **Intra-Insert Operator:** Given a solution π, two runways $i' \neq i''$ are first chosen uniformly randomly. Then for each runway $i \in \{i', i''\}$, each $i[k]$ is removed and inserted at each position in $\vec{i}$ of π to obtain $(|\vec{i}| - 1)$ new solutions. Only the best new solution is however returned. Algorithm 24 provides the pseudo-code of the intra-insert operator and Figure 4.3 (d) shows an example. Note that we use two runways here although the removal and insertion of aircraft are done within a single runway. This is just to match with the inter-swap operator that involves two runways every time. Also, note that we select both runways randomly because the runway with the highest weighted tardiness is already considered in the perturbation phase. Clearly, Algorithm 24 runs in $O(\lceil N/M \rceil^3)$ time and $O(N)$ space when TWT is computed incrementally.

Algorithm 24 Intra-insert operator

1. Let π be the current solution given as a parameter
2. Randomly choose two runways i' and i'' from all runways
3. For each $i \in \{i', i''\}$, for each $k, k' \in [1, |\vec{i}|)$, every time in π, remove aircraft $i[k]$ and insert at position k' to obtain a new solution.
4. From the $|\vec{i'}| \times (|\vec{i''}| - 1)$ new solutions, return the best one (in terms of weighted tardiness)

We use at most two operators in the perform exploration procedure shown in Algorithm 25. We will later choose the operators through experimentation. Algorithm 25 is basically a variable neighbourhood descent (VND) algorithm [44]. The main advantage

of VND is due to the use of multiple neighbourhood operators in local search. The search might lead to a different local optimum when using each of these operators separately. However, using them within a combination has the advantage of escaping a local optimum associated with one operator through the use of another operator.

Algorithm 25 Perform exploration

1. Let π be the current solution given as a parameter
2. Let $\mathcal{N}_1$ and $\mathcal{N}_2$ are two given neighbourhood operators
3. Let $l = 1$ // to denote operator $\mathcal{N}_l$ will be used
4. While $l \leq 2$ do // since we are using two operators here
   (a) π' = the best solution (in terms of weighted tardiness) among all the solutions obtained by applying $\mathcal{N}_l$ on π
   (b) If π' is better than π then $\pi = \pi', l = 1$ else $l = l + 1$
5. Return solution π

Performing Perturbation

There could be various ways to perturb (destroy and reconstruct some parts of) an ASP solution. Existing methods mostly perform random swapping or random removal and reinsertion for a number of times. *In our constraint guided strategy, we make an attempt to fix the most problematic part in the current solution.* This could mean selecting the aircraft j having the most weighted tardiness $\text{WT}(j)$ and placing it to the best possible position. However, selecting such aircraft does not work well because mostly the aircraft that are towards the end of the schedule of the runways get selected. This is because the tardiness of the previous aircraft has a cascaded effect on the aircraft scheduled later. We therefore select a runway i having the maximum weighted tardiness $\text{WT}(i) = \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]})$ of the aircraft assigned to the runway. Then, we take c aircraft from c randomly selected positions in $\vec{i}$ and remove them from $\vec{i}$. Then we append each of the c aircraft to the end of $\vec{i}$ and consider swapping the aircraft with all other aircraft in the same runway. Among the solutions produced, the one with the best $\text{WT}(i)$ is selected to be used for the next d. We repeat the above steps for a given d times. Algorithm 26 shows the perturbation procedure.

Instead of swapping the selected aircraft with the other aircraft scheduled in the same runway, we also experiment with a removal and reinsertion based approach. In this

Algorithm 26 Swap based perturbation

1. Let π be the current solution given as input
2. For $d' = 1$ to d do // d is given as input
   (a) $i = \text{argmax}_i \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]})$
   (b) For each $c' = 1$ to c do // c is a given input,
   remove job $j_{c'}$ from $\vec{i}$ where $j_{c'} = i[k]$, k is randomly selected from $[1, |\vec{i}|]$, and at least one $j_{c'}$ for this d' should be different from all $j_{c'}$ selected for the immediate previous d'.
   (c) For each $c' = 1$ to c do, // add to $\vec{i}$ in SBH style
   append $j_{c'}$ at the end of $\vec{i}$ to get a new $\vec{i}$ and then for each $k' \in [1, |\vec{i}|]$ but $k' \neq k$, swap aircraft $j_{c'}$ with the aircraft at $i[k']$ to obtain a new $\vec{i}$. Let $\vec{i}$ be the best (in terms of weighted tardiness) among all the new $\vec{i}$.
3. Return solution π

case, as part of the destruction phase, we remove c aircraft from $\vec{i}$. Then, as part of the construction phase, each of the c aircraft is inserted into $\vec{i}$ in all possible positions—at the beginning, at the end, and in between two successive aircraft, but not at position k again—to obtain new solutions. The best one (in terms of weighted tardiness) among the new solutions is then selected on which similar perturbation steps are repeated. Details of this approach is shown in Algorithm 27.

Algorithm 27 Insertion based perturbation

1. Let π be the current solution given as input
2. For $k = 1$ to d do // d is given as input
   (a) $i = \text{argmax}_i \sum_{k=1}^{|\vec{i}|} w_{i[k]}(t_{i[k]} - r_{i[k]})$
   (b) For each $c' = 1$ to c do // c is a given input,
   remove job $j_{c'}$ from π where $j_{c'} = i[k]$, k is randomly selected from $[1, |\vec{i}|]$, and at least one $j_{c'}$ for this d' should be different from all $j_{c'}$ selected for the immediate previous d'.
   (c) For each $c' = 1$ to c do, // add to $\vec{i}$ in IBH style
   insert $j_{c'}$ at the beginning, at the end, or at any middle position of $\vec{i}$ to obtain new $\vec{i}$s. Let $\vec{i}$ be the best (in terms of weighted tardiness) among all the new $\vec{i}$.
3. Return solution π

In our experiments, we also use random runway selection in Algorithms 26 and 27 instead of the greedy selection keeping all other steps in those algorithms the same. This is to show the effectiveness of the greedy selection.

Lemma 4.4. *Algorithms 26 and 27 both runs in $O(dN^2)$ time and $O(N)$ space.*

Proof. In each of the d times, selection of the runway takes $O(M)$ time and $O(M)$ space, then $O(\lceil N/M \rceil)$ new solutions are created and computing the TWTs take $O(N)$ time if from scratch or $O(\lceil N/M \rceil)$ time if incrementally. Assuming $N \gg M$, the time complexity is thus $O(dN^2)$. The space complexity is certainly related to the number of aircraft. □

Acceptance criterion

The new solution π' returned by the exploration method is accepted in Algorithm 20 if it is better than the current solution π; otherwise it is accepted with a probability p where p is a parameter of our algorithm.

Handling Runway Capacity

Assume 15 aircraft, 3 runways and $\nu = 2$. So the maximum and minimum numbers of aircraft to be allocated to each runway are 7 and 3 respectively. Disregarding possible symmetries over permutations, we can distribute 15 aircraft over 3 runways in the following 4 ways $\langle 5, 5, 5 \rangle$, $\langle 4, 5, 6 \rangle$, $\langle 3, 6, 6 \rangle$, $\langle 3, 5, 7 \rangle$. These 4 combinations have standard deviations 0, 1, 1.732, and 2 respectively. We can use the standard deviation of a combination to denote the diversity level within that combination.

As could be easily understood, there are quite a large number of possible combinations for moderate sizes of N, M, ν. Finding an optimal solution for an ASP is very difficult even for a given combination, let alone trying all possible combinations. We therefore propose a practical approach. Since in theory, with uniform distributions of types of aircraft, types of operations, and times of operations, optimal solutions lie in the combinations that have standard deviations very close to zero, we should consider only such combinations. Moreover, separate runs could be made, preferably in parallel, for each combination than allowing change of combinations within the same run. This is to produce good solutions very quickly given the short time windows available in ASP instances and also considering the typical time performance of our algorithms. The length of typical time windows could be understood from the following information.

For a landing aircraft, the scheduling process of the aircraft starts once the aircraft enters the TMA, about 30-40 minutes before its planned landing time. The start time

(operation time) is finalised in advance of reaching the final approach path, about 20-30 minutes before landing [183]. For a departing aircraft, the scheduling process of each aircraft starts once the aircraft enters the holding area. The start time (operation time) is finalised in advance of entering the taxiway, since it is not possible to change the sequence during taxiing or at the holding area. Thus, departing operations are scheduled about 20 minutes before planned departure time [147]. The planning horizon for ASP is usually 20-30 minutes.

In this work, we propose very efficient local search algorithms that run on a given combination of runway capacities. These algorithms are fast enough to be used within a dynamic system that solves a series of static instances to deal with the sliding time windows in a dynamic system [133, 184]. In our experiments, we however evaluate our algorithms over various combinations of runway capacities having various standard deviations.

4.2.6 Experimental Results

We have implemented our algorithm in C++ language and on top of the constraint-based local search system, Kangaroo [185]. The functions and the constraints are defined using invariants in Kangaroo. Invariants are special constructs that are defined by using mathematical operators over the variables. While computation of constraint violations and objective functions, and temporary evaluation of neighbourhood solutions are performed incrementally by Kangaroo, we mainly focus on the search algorithms. It is worth noting that by using Kangaroo, we automatically get speed-up methods that achieve performances similar to that achieved by the acceleration methods proposed by [71] for flowshop scheduling. As mentioned in Section 4.2.2, we compare our algorithm with the SA algorithm [15] and the ILS algorithm for ASP [13]. These two algorithms have been reconstructed in C++ and on top of Kangaroo. All experiments reported in this work have been run on a MAC computer 2.70 GHz Intel Core i5-4570 processor and 8 GB RAM 1600MHz memory using the OS X operating system.

In this work, we test our algorithms on 12 real-world instances based on Milano Linate Airport [186]. These instances include 60 aircraft whereas weights are determined based on the number of seats and the fuel consumption of each aircraft. These instances are publicly available at http://www.or.deis.unibo.it/research.html. In this work, we also

have used 75 instances that were generated by [2] considering the realistic parameters of the Doha International Airport. Using real-world instances overs the crafted ones are often preferable but depending on exactly what scenarios are used the real-world instances might not have all the characteristics to test the strengths and weaknesses of a particular algorithm. Therefore, crafted instances particularly when generated using realistic parameters are good benchmarks for a thorough empirical evaluation of an algorithm.

This benchmark set used by [2] using Doha International Airport parameters is made up of problem instances with a number of aircraft $n = 15, 20, 25, 50$ and a number of runways $r = 2, 3, 4, 5$. These instances are available from the website http://ahmed.ghoniem.info/download/MASP-SET.txt. In this benchmark, the ready times of the aircraft are randomly generated using a discrete uniform distribution over the interval $(0, \gamma \times \frac{n}{m})$ where γ is a parameter selected in the interval (30, 100). In addition, operations (landing and takeoff) and aircraft classes (small, large and heavy) are selected randomly. The minimum separation times are taken from actual numbers based on FAA standard that can be seen in Table 4.1.

Before starting the comparison, it is worth noting that in this chapter, the main instances for evaluation are those proposed by [2] using Doha International Airport parameters. These benchmarks vary in sizes of the aircraft and runways, which allows us to have a comprehensive analyse and find the impact of each of the factors of the tested algorithms. These instances have been widely used in the literature e.g. in [135, 179, 180]. Aircraft scheduling becomes important because finding an optimal schedule is really hard when a large number of aircraft are to be operated within a small period of time. In real instances taken from Milan airport, 60 aircraft should be operated in almost 2.5 hours (two and half hours) while in Farhadi's benchmark 50 aircraft should be operated in almost 40 min in a case with two runways. Interestingly, it is clear that when more runways are added in an instance, solving the problem becomes easier. This has been considered in Farhadi's instances. For example, in instances with 50 aircraft and 3 runways, 50 aircraft should be operated in almost 25 minute. All of aforementioned reasons encourage us to use Farhadi's benchmark instead of real-world instances.

The performance of each algorithm is evaluated by means of Error (ERR) defined by the following formula where TWT is the average of the total weighted tardiness obtained

by the given algorithm over 5 runs and Best is the best weighted tardiness obtained by any of the tested algorithms.

$$\text{ERR} = \frac{\text{TWT} - \text{Best}}{\text{Best}} \times 100\% \tag{4.6}$$

As a termination criterion, all local search algorithms are run for $T_{\max} = N \times M \times k$ ms CPU time, where N and M are respectively the numbers of aircraft and runways while $k = 10$.

Proposed Constructive Search Method

We use two constructive heuristics IBH and SBH. For both, we first sort the aircraft on an increasing order of the ready times. As an alternative, we could arrange the aircraft on a decreasing order of the weights breaking ties randomly. Combining these alternatives, we have four constructive heuristic methods: IBHR, IBHW, SBHR, and SBHW. Among these, SBHW mostly leads to infeasible solutions, so we do not consider it for further evaluation. With the three remaining heuristics, we also compare the AATCSR heuristic proposed by [15], who showed that AATCSR is better than two other heuristics. AATCSR allocates aircraft to runways using a complex index formula.

Table 4.4 shows performance of the heuristics. Based on the results, we see that the proposed three heuristics are better than AATCSR and the ready time criterion is better than the weight criterion. Moreover, SBHR is better than IBHR implying swapping is better than insertion. However, to check the statistical significance of these conclusions, we also show a 95% confidence interval plot in Figure 4.4. From this plot, we see that the difference between AATCSR and IBHW is not statistically significant. The same is true for the difference between IBHR and SBHR. However, the difference between the performance of IBHR or SBHR is statistically significant from that of IBHW or AATCSR.

It is worth noting that the average CPU time taken by three proposed heuristics are around 0.03 second, which is larger than the CPU time 0.001 second taken by AATCSR. However, 0.03 second is practically a negligible time, particularly in the context of the aircraft sequencing problem. Overall, AATCSR heuristic allocates aircraft to runways

TABLE 4.4: Performance of the constructive heuristics.

Instance	**AATCSR**	**IBHW**	**IBHR**	**SBHR**
$N, M, \#$	ERR (%)	ERR (%)	ERR (%)	ERR (%)
$15, 2, 5$	40.9	11.2	10.7	**5.8**
$15, 3, 5$	35.2	**1.2**	15.9	12.8
$15, 4, 5$	15.8	**0.0**	15.7	10.8
$20, 2, 5$	58.0	25.8	9.0	**3.1**
$20, 3, 5$	30.9	17.6	**1.6**	5.0
$20, 4, 5$	40.9	11.2	10.7	**5.8**
$20, 5, 5$	21.1	**2.3**	9.0	9.0
$25, 2, 5$	61.2	39.7	**3.8**	**3.8**
$25, 3, 5$	28.8	18.3	14.3	**10.0**
$25, 4, 5$	44.1	19.1	19.9	**10.1**
$25, 5, 5$	20.5	21.3	03.4	**0.5**
$50, 2, 5$	189.7	44.9	**0.0**	**0.0**
$50, 3, 5$	151.5	78.9	**7.2**	**07.2**
$50, 4, 5$	11.9	183.1	164	**5.1**
$50, 5, 5$	112.8	213.9	2.2	**0.0**
average	58.9	45.4	8.9	**6.7**

$\#$ Number of instances for a given N, M pair Bold: minimum error

following a complex ranking, while proposed heuristics, although start with an initial ranking, perform all possible insertion or swapping of a given aircraft. The difference in the CPU time is therefore reasonable and pays off.

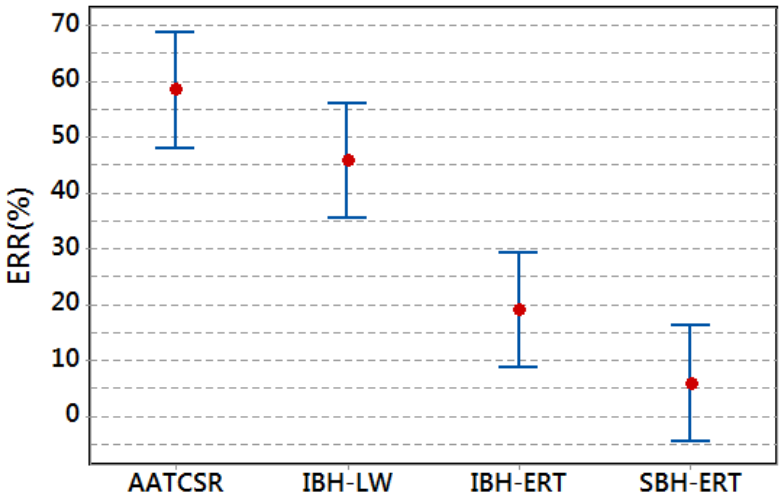


FIGURE 4.4: 95% confidence interval plot for four heuristic algorithms

Given the conclusions above and unless stated otherwise, for the rest of the experiments, we will use SBHR as our initialisation method for the explorative perturbative search and will also denote this simply by SBH.

Local Search Parameter Tuning

There are three parameters in our EPS algorithm: one is d the number of times to perform the perturbation steps, another is c the number of aircraft to be selected in each iteration of the perturbation steps, and the other is p the probability of accepting the solution returned by the exploration procedure. To determine the suitable values of these parameters, we follow the method of full factorial design. For each parameter, we consider 4 different values, which are listed below. For this experiment, we randomly select 15 instances from the 75 instances in our benchmark. For each of the $4 \times 4 \times 4$ settings and for each instance, we then run our algorithm 5 times with the same timeout as mentioned before. Note for this experiment, in the VND based perform exploration procedure, we use inter-swap as $\mathcal{N}_1$ and adjacent-swap as $\mathcal{N}_2$.

- $d \in \{1, 2, 3, 4\}$
- $c \in \{1, 2, 3, 4\}$
- $p \in \{0.3, 0.4, 0.5, 0.6\}$

The ANOVA results are given in Table 4.5. From this table, c is the most significant factor due to the highest F-ratio, followed by p and d factors. The interval plots of these factors considering 95% Tukey's Honest Significant Difference (HSD) confidence intervals are shown in Figure 4.5. As we see, among the values tested, our EPS statistically significantly performs the best when $d = 2$, $c = 3$, and $p = 0.5$. In the rest of the thesis, we will use these parameter values.

Table 4.5: ANOVA table for the three parameters.

Source	DF	SS	MS	F	P-value
d	3	3376.4	1125.46	4977.36	**0.000**
c	3	8758.4	2919.46	12911.32	**0.000**
p	3	3659.8	1219.93	5395.14	**0.000**
$d * c$	9	1649.4	183.27	810.51	**0.000**
$d * p$	9	954.1	106.01	468.83	**0.000**
$c * p$	9	349.3	38.81	171.65	**0.000**
Error	4763	1077.0	0.23		
Total	4799	19824.3			

As already mentioned, we compared the proposed EPS algorithm against the SA algorithm [15] and the ILS algorithm [13]. The SA algorithm was proposed for the same problem and was even evaluated on the same instances. So, its parameters are taken from the chapter. However, ILS is originally proposed for ALP. So, in this thesis, we

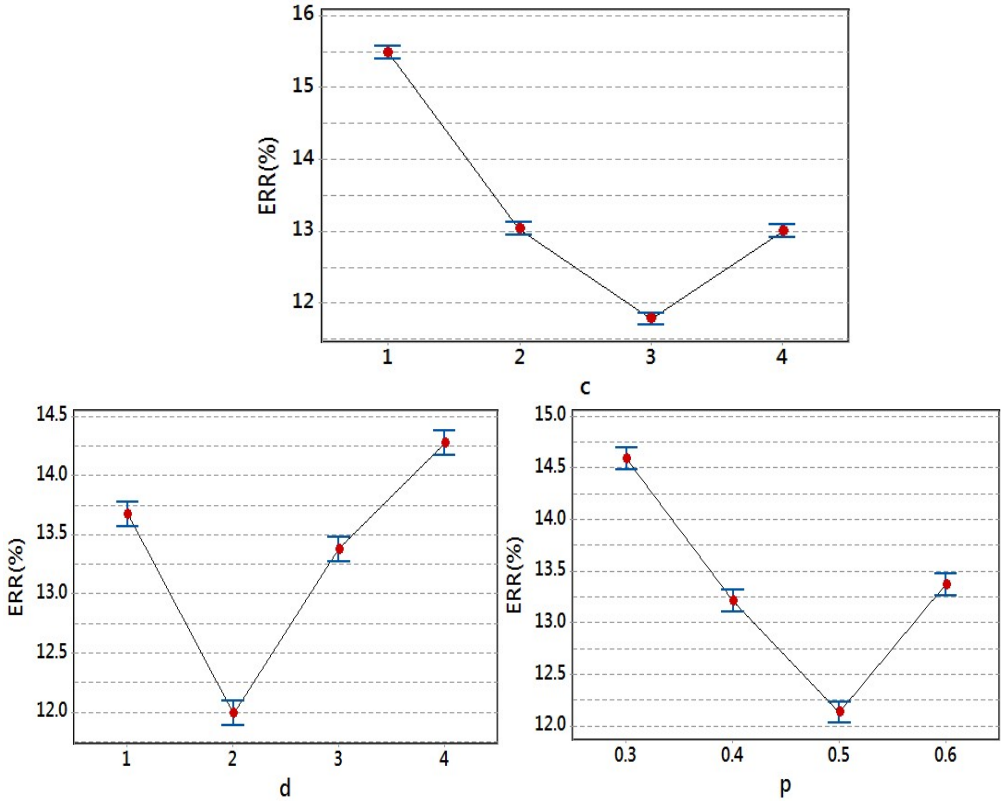


FIGURE 4.5: Interval plots with 95% Tukey's HSD confidence intervals for c, d and p

again did a parameter calibration for this algorithm. There are two parameters in this algorithm that need to be tuned, minimum time varying mintv, and maximum time varying maxtv. In the original chapter, the authors tested three values for mintv: 0.1-0.3, and two values for maxtv: 0.8 and 0.9. We also consider these values for testing. The ANOVA table and The interval plots of these factors considering 95% HSD confidence intervals are shown in Table 4.6 and Figure 4.6. As can be seen, ILS has its best performance with mintv = 0.1 and maxtv = 0.9, although there is no significant difference between maxtv values.

TABLE 4.6: ANOVA table for the three parameters.

Source	DF	SS	MS	F	P-value
mintv	2	1798.22	899.109	466.29	**0.000**
maxtv	1	15.92	15.919	8.26	**0.004**
mintv $*$ *maxtv*	2	4.10	2.052	1.06	**0.346**
Error	444	856.12	1.928		
Total	449	2674			

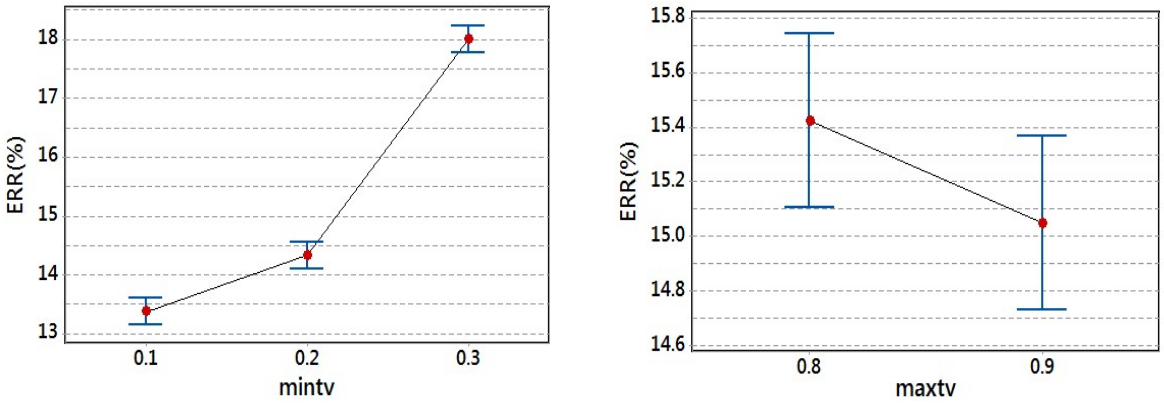


FIGURE 4.6: Interval plots with 95% Tukey's HSD confidence intervals for mintv and maxtv

Greediness in Perturbation

In our EPS, in the perturbation phase, we use a greedy runway selection instead of a typical random selection. To test the effectiveness of our greedy selection over the random selection strategy, we have run experiments on all 75 instances with the initialisation method and the parameter values chosen above. Figure 4.7 shows a 95% confidence interval plot of the average errors. However, in this set of experiments, we separately use both insertion and swap based moves. From the plot, the greedy selection with swap based moves appears to be significantly better than the other three configurations.

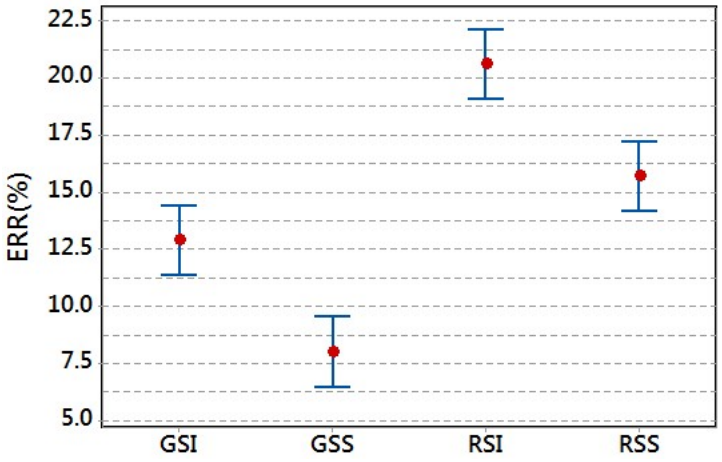


FIGURE 4.7: Comparison of greedy runway selection with swap move (GSS), greedy runway selection with insertion move (GSI), random runway selection with swap move (RSS), and random runway selection with insertion move (RSI)

Neighbourhood Operator Combination

In our EPS, in the VND based exploration procedure, we use two neighbourhood operators: inter-swap and adjacent swap. We mentioned before that using inter-swap as $\mathcal{N}_1$ and adjacent-swap as $\mathcal{N}_2$ produces better solutions than using the other way around. To test this, we have run experiments on all 75 instances with the initialisation method and other settings chosen so far. For this reason, four common neighbourhood move is tested: Inter Swap, Intra Insert, Adjacent Swap, and Intra Swap. Firstly, only one of them is considered as the solo move. Then three different Intra moves are tested with the only inter move. Figure 4.8 shows a 95% confidence interval plot of the average errors. From the plot, we see that firstly, using two moves is better than using only one, and among all cases with two neighbours, employing Adjacent Swap and Inter Swap has the better results than other cases. In addition, due to the nature of VND that first move is used more than the second one, the order of them will affect the performance of the algorithm. So, in all cases, we test two possible orders. As can be seen, our aforementioned selection is significantly better than the other way around, except for the Adjacent Swap/Intra Swap case. So in further experiments, we use inter-swap as $\mathcal{N}_1$ and adjacent-swap as $\mathcal{N}_2$ in the VND algorithm. Besides, in the Intra Swap move, we are selecting one of the runways randomly and for another, the one with the highest objective is selected. Here, we are to show the effectiveness of this selection compared to two other possible selection approaches: (1) selecting both runways randomly, and (2) selecting both runways greedily: the one with the highest and the one with the lowest objectives. A 95% confidence interval plot of the average errors is given in Figure 4.9. As can be observed, our runway selection is significantly better than the other cases.

Overall Performance Comparison

We compare our final EPS algorithm with the SA algorithm [15] and the ILS algorithm [13]. Since initial solutions normally have impacts on the overall performance obtained by a metaheuristic algorithm, we combine the existing AATCSR and our SBH (SBHR to be exact) methods for initialisation with the ILS, the SA, and our EPS algorithms. These give us 6 combinations in total. We use all 75 instances in this experiment. To make it clearer, in our EPS, we use $p = 0.5$, $d = 2$, $c = 3$, SBH for initialisation, greedy

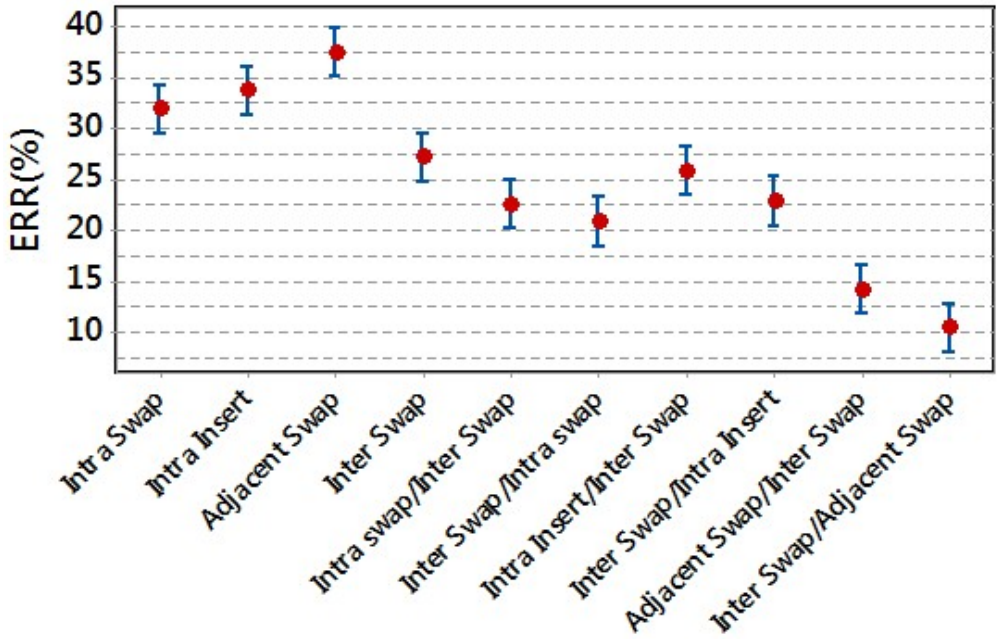


FIGURE 4.8: 95% confidence interval plot for different neighbourhood operator combinations

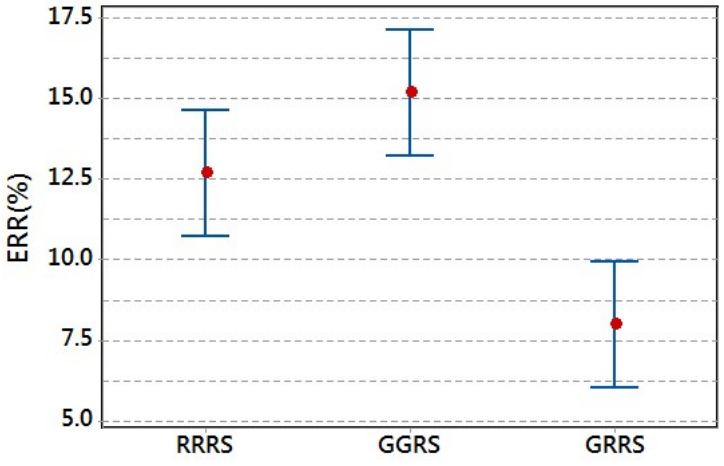


FIGURE 4.9: Comparison of highest and lowest runway selection in inter swap move (GGRS), greedy runway (highest one) and random runway selection in inter swap move (GRRS), random runway selection in inter swap move (RRRS)

runway selection in perturbation, and Inter-Swap/Adjacent-Swap combination in the VND based exploration with greedy selection in the Inter-Swap move.

For runway capacities restricted by $\lceil N/M \rceil$, the experimental results are presented in Table 4.7 and a 95% confidence interval plot is in Figure 4.10. SBH heuristic helps all three algorithms obtain better results than when using AATCSR heuristic, albeit the differences are not statistically significant for ILS and SA. Among all variants of the three algorithms, the EPS-SBH has the best performance and the difference is statistically significant except EPS-AATCSR. The EPS-AATCSR is significantly better than other algorithms except ILS-SBH. There is no statistically significant difference among SS-SBH, ILS-AATCSR and ILS-SBH.

TABLE 4.7: Performance of EPS, ILS, and SA using AATCSR and SBH for ASP problems

Problem Instances	**EPS**		**ILS**		**SA**	
	AATCSR	**SBH**	**AATCSR**	**SBH**	**AATCSR**	**SBH**
$N, M, \#$	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)
15,2,5	2.74	**2.07**	4.50	4.31	10.12	8.04
15,3,5	1.21	**1.11**	15.07	9.83	11.64	7.32
15,4,5	5.10	**0.38**	16.85	13.66	9.24	13.66
20,2,5	9.85	**3.65**	16.48	8.05	47.06	15.11
20,3,5	7.16	**0.71**	17.71	24.32	44.90	34.90
20,4,5	2.60	**0.06**	23.24	17.99	36.92	24.24
20,5,5	11.81	**2.76**	28.71	19.28	30.62	20.68
25,2,5	9.81	**1.82**	9.81	18.98	76.83	36.02
25,3,5	10.08	**8.57**	18.68	24.12	56.89	45.48
25,4,5	8.20	**6.96**	48.43	42.22	87.15	70.50
25,5,5	6.59	**0.63**	39.58	25.76	45.33	30.72
50,2,5	15.40	**0.22**	24.94	7.67	185.81	21.66
50,3,5	19.67	**1.95**	57.22	17.05	204.59	52.18
50,4,5	13.73	**0.87**	53.24	51.14	95.47	78.62
50,5,5	9.71	**0.51**	146.61	53.43	250.78	73.70
Average	8.91	**2.15**	34.74	22.52	79.56	35.52

Number of instances for a given N, M pair Bold: minimum error

To investigate the convergence profiles of the algorithms over time, we also show a plot in Figure 4.11. In this plot, the time is represented by the values of k where our timeout is $T_{\max} = N \times M \times k$ ms CPU time. From this plot, we see that our EPS is much faster than SA and ILS algorithms with both SBH and AATCSR initialisation heuristics to converge to global or local optima. Moreover, with SBH heuristic, both SA and ILS algorithms do not make much progress over time. We ran the experiments up to $k = 20$ to check whether the progress profile changes, but could not find any.

So far we have shown results for only one runway capacity combination where the number of aircraft per runway is at most $\lceil N/M \rceil$. In Table 4.8, and Figure 4.12, we show the

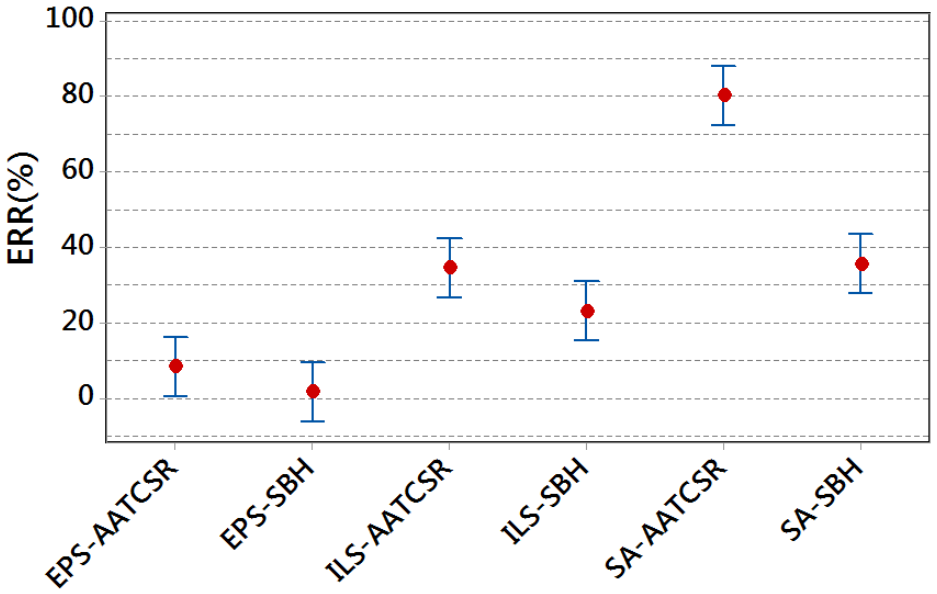


FIGURE 4.10: 95% confidence interval plot for ASP on Farhadi [2] instances

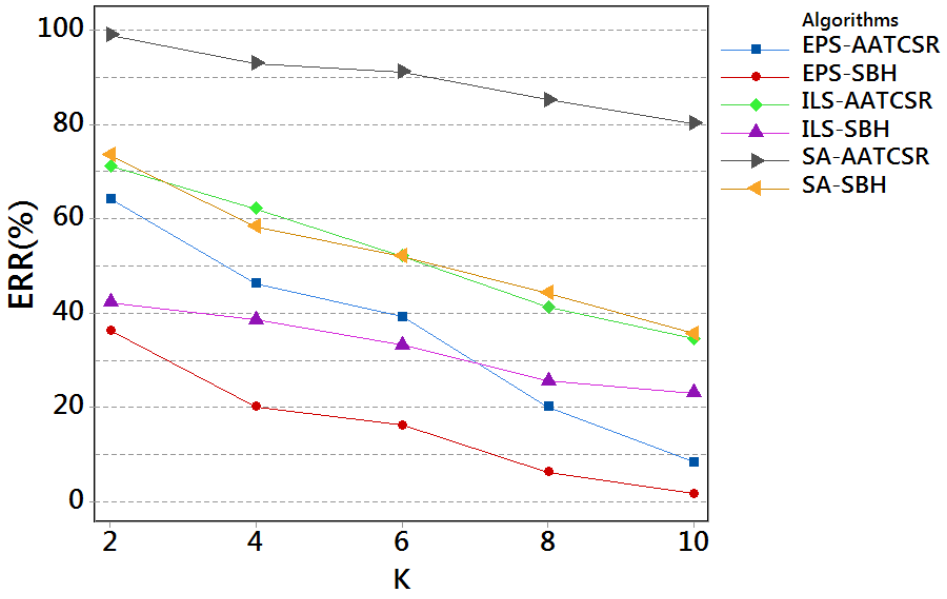


FIGURE 4.11: Covergence profiles of the algorithms for ASP on Farhadi [2] instances

results for a number of combinations with various standard deviations ranging from very low to very high. However, the conclusions that could be drawn from these results remain the same.

TABLE 4.8: Performance of EPS, ILS, and SA using AATCSR and SBH for a number of runway capacity combinations

Problem	**Runway**	**EPS**		**ILS**		**SA**	
Instances	**Capacity**	**AATCSR**	**SBH**	**AATCSR**	**SBH**	**AATCSR**	**SBH**
$N, M, \#$	**Combination**	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)
15,2,5	8-7	2.74	**2.07**	4.50	4.31	10.12	8.04
15,2,5	9-6	5.43	**0.53**	3.56	9.52	14.49	12.39
15,2,5	10-5	3.84	**1.07**	3.66	7.28	13.10	14.62
15,3,5	5-5-5	1.21	**1.11**	15.07	9.83	11.64	7.32
15,3,5	6-4-5	1.01	**0.79**	12.74	10.18	11.32	9.54
15,3,5	7-4-4	1.12	**0.07**	15.67	12.63	13.00	11.81
15,3,5	7-5-3	1.15	**0.58**	19.24	15.61	12.66	14.97
15,4,5	4-4-4-3	5.10	**0.38**	16.85	13.66	9.24	13.66
15,4,5	5-3-4-3	2.69	**0.83**	12.54	9.71	7.09	9.71
15,4,5	6-3-3-3	3.65	**0.08**	17.34	15.67	7.94	15.67
20,2,5	10-10	9.85	**3.65**	16.48	8.05	47.06	15.11
20,2,5	11-9	5.05	**2.34**	16.11	14.97	52.18	44.68
20,2,5	12-8	8.33	**1.72**	13.97	7.77	50.67	33.40
20,2,5	13-7	8.97	**8.86**	19.06	25.95	58.92	60.90
20,3,5	7-7-6	7.16	**0.71**	17.71	24.32	44.90	34.90
20,3,5	8-6-6	5.62	**3.01**	18.62	19.04	42.37	27.81
20,3,5	9-6-5	9.94	**2.74**	22.47	28.52	46.32	42.30
20,3,5	10-5-5	5.14	**1.60**	29.04	38.37	58.99	56.51
20,3,5	10-6-4	3.18	**1.46**	25.16	34.06	61.16	48.44
20,3,5	10-7-3	8.40	**1.99**	26.18	26.05	44.27	36.46
20,4,5	5-5-5-5	2.60	**0.06**	23.24	17.99	36.92	24.24
20,4,5	6-4-5-5	3.41	**0.80**	20.34	18.89	31.39	26.80
20,4,5	7-4-4-5	4.17	**0.18**	34.10	18.62	42.84	23.64
20,4,5	7-5-3-5	2.07	**0.12**	35.83	25.46	37.48	29.08
20,4,5	7-6-3-4	5.04	**1.39**	41.82	19.10	43.36	23.95
20,4,5	7-7-3-3	6.31	**0.38**	38.10	38.46	41.19	35.93
20,5,5	4-4-4-4-4	11.81	**2.76**	28.71	19.28	30.62	20.68
20,5,5	5-3-4-4-4	4.95	**0.30**	31.23	22.69	29.17	22.59
20,5,5	6-3-3-4-4	4.15	**0.28**	31.59	21.78	33.73	24.42
20,5,5	7-3-3-3-4	5.23	**0.00**	41.48	31.17	39.46	34.08
25,2,5	13-12	9.81	**1.82**	9.81	18.98	76.83	36.02
25,2,5	14-11	20.49	**0.61**	19.33	17.35	85.64	31.84

25,2,5	15-10	37.90	**0.86**	35.76	37.81	115.36	82.76
25,2,5	16-9	30.37	**3.05**	30.37	61.93	111.32	296.01
25,3,5	9-8-8	10.08	**8.57**	18.68	24.12	56.89	45.48
25,3,5	9-9-7	3.08	**1.92**	16.01	32.99	67.52	47.98
25,3,5	10-8-7	7.60	**1.07**	36.44	42.51	72.99	60.20
25,3,5	11-8-6	9.98	**0.54**	29.43	70.05	69.02	111.89
25,3,5	12-7-6	12.00	**3.32**	48.62	88.19	97.98	117.06
25,3,5	13-7-5	19.01	**2.38**	61.49	52.78	100.71	118.25
25,3,5	13-8-4	19.19	**1.82**	76.17	103.94	100.86	160.09
25,4,5	7-6-6-6	8.20	**6.96**	48.43	42.22	87.15	70.50
25,4,5	7-7-6-5	**6.22**	9.62	55.72	35.09	86.64	48.30
25,4,5	8-6-6-5	11.86	**1.09**	54.27	43.29	108.16	58.80
25,4,5	7-7-7-4	15.32	**0.51**	46.28	45.65	84.17	58.68
25,4,5	8-6-7-4	8.99	**1.74**	57.27	63.55	102.22	79.74
25,4,5	9-6-6-4	6.65	**4.12**	63.62	54.58	101.37	67.79
25,4,5	9-7-5-4	4.98	**0.78**	57.44	51.01	101.04	61.42
25,4,5	10-6-5-4	3.02	**2.64**	72.00	55.25	112.98	67.01
25,4,5	10-7-4-4	2.10	**0.85**	56.13	52.47	90.69	82.57
25,5,5	5-5-5-5-5	6.59	**0.63**	39.58	25.76	45.33	30.72
25,5,5	6-4-5-5-5	4.73	**1.61**	36.05	25.72	48.68	29.72
25,5,5	7-4-4-5-5	9.45	**0.13**	49.35	48.63	53.33	51.04
25,5,5	7-5-3-5-5	6.69	**0.22**	45.76	35.95	55.61	40.50
25,5,5	7-6-3-4-5	4.62	**1.97**	44.54	44.06	67.53	54.73
25,5,5	7-7-3-4-4	6.30	**1.01**	49.19	44.10	61.04	49.99
25,5,5	7-7-3-5-3	1.59	**1.37**	53.93	39.09	59.05	50.19
50,2,5	25-25	15.40	**0.22**	24.94	7.67	185.81	21.66
50,2,5	26-24	6.08	**0.95**	10.37	7.12	119.18	19.52
50,2,5	27-23	7.72	**0.00**	12.39	10.92	109.93	22.64
50,2,5	28-22	21.58	**2.58**	28.18	13.62	136.18	36.66
50,2,5	29-21	**3.40**	3.44	13.71	7.31	130.98	42.83
50,2,5	30-20	**0.22**	0.56	0.78	1.35	21.20	15.30
50,3,5	17-16-16	19.67	**1.95**	57.22	17.05	204.59	52.18
50,3,5	18-16-16	10.83	**3.40**	31.57	16.14	92.34	59.57
50,3,5	19-16-15	13.05	**4.67**	36.75	30.63	106.59	75.89
50,3,5	20-15-15	23.50	**4.18**	53.65	31.04	170.64	86.86
50,3,5	21-15-14	23.49	**3.81**	53.23	28.26	140.31	94.11
50,3,5	21-16-13	24.09	**9.56**	61.18	34.54	164.39	92.88
50,3,5	21-17-12	14.73	**3.10**	48.76	30.60	152.98	74.20
50,4,5	13-13-12-12	13.73	**0.87**	53.24	51.14	95.47	78.62
50,4,5	13-13-13-11	6.67	**1.92**	43.85	61.60	97.24	97.08
50,4,5	14-12-13-11	7.08	**0.67**	44.32	93.52	93.38	140.89

50,4,5	15-12-12-11	9.19	**6.35**	45.89	97.12	86.15	118.42
50,4,5	16-12-12-10	18.94	**11.19**	50.53	107.27	103.11	162.59
50,4,5	16-13-11-10	**8.22**	15.65	54.90	105.73	113.95	174.80
50,4,5	16-14-11-9	28.61	**8.37**	99.55	128.69	168.96	228.58
50,4,5	16-15-10-9	13.62	**0.45**	81.87	101.91	124.53	193.14
50,4,5	16-16-10-8	11.29	**6.89**	48.63	122.18	114.08	167.33
50,4,5	16-16-9-9	23.24	**5.94**	85.02	163.81	147.35	297.49
50,5,5	10-10-10-10-10	9.71	**0.51**	146.61	53.43	250.78	73.70
50,5,5	11-9-10-10-10	0.76	**0.18**	144.05	49.42	229.51	73.50
50,5,5	12-9-9-10-10	9.34	**0.70**	119.15	133.64	248.57	228.25
50,5,5	13-9-9-9-10	25.32	**2.08**	164.46	482.81	264.77	326.51
50,5,5	13-10-8-9-10	30.04	**4.30**	175.80	584.66	232.15	596.51
50,5,5	13-11-8-8-10	30.96	**11.30**	176.05	573.35	260.13	461.59
50,5,5	13-12-8-8-9	4.62	**2.06**	130.83	115.51	197.69	583.96
50,5,5	13-13-7-8-9	**6.31**	6.72	88.92	208.44	160.72	254.27
50,5,5	13-13-8-8-8	**1.99**	4.78	119.28	200.03	178.36	234.43
50,5,5	13 13 7 10 7	44.08	**8.93**	174.81	188.23	266.08	208.91

Number of instances for a given N, M pair Bold: minimum error

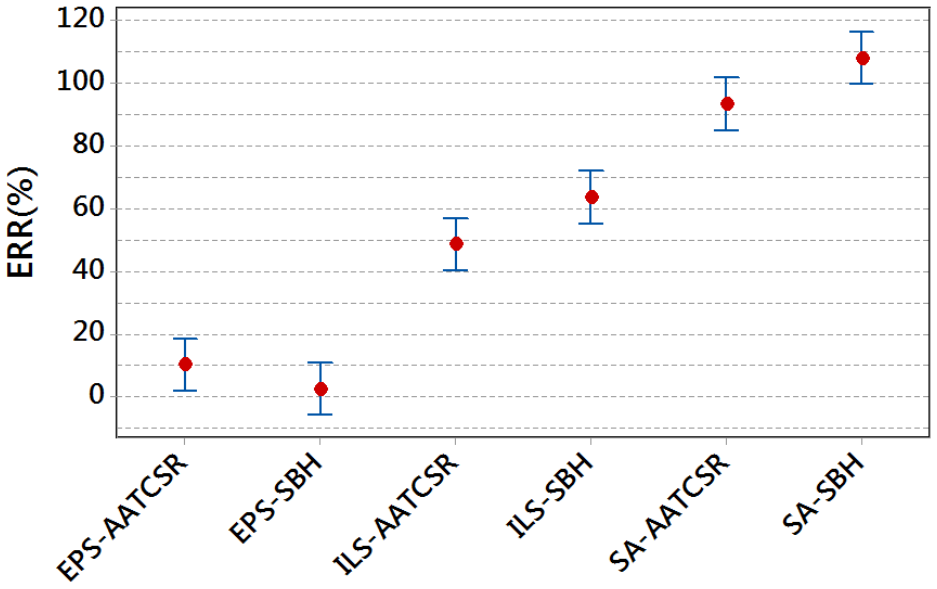


FIGURE 4.12: 95% confidence interval plot for ASP on [2] instances for different runway capacity combination.

To investigate which runway capacity combinations help any of the competitor algorithms return the best found solutions, in Figure 4.13, we show the average of the best found solutions against the standard deviations of runway capacity combinations. It is

clear that in all problem sizes, the lowest standard deviations help find the best solutions. So for further analysis, we only show the results for those cases where the runway capacities are restricted by $\lceil N/M \rceil$.

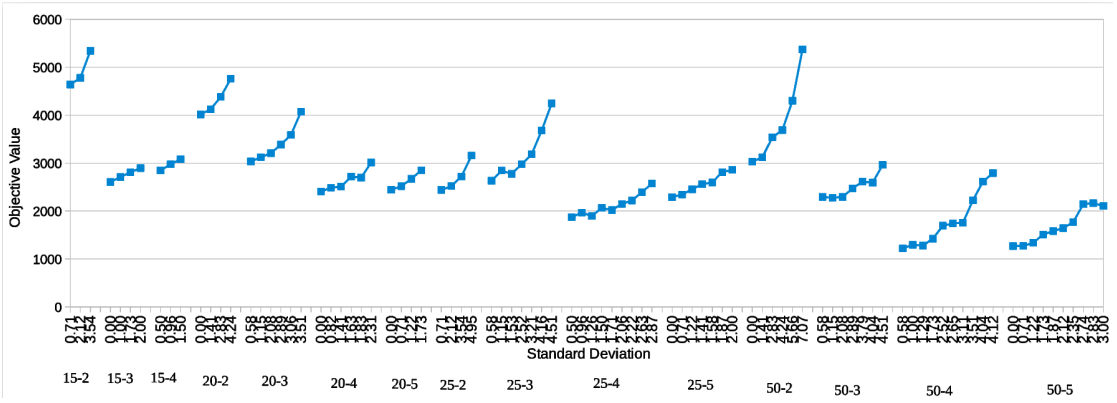


FIGURE 4.13: Average of the best found solutions over all competitor algorithms over various standard deviations of runway capacity combinations

Comparison on the Real Instances

In this section, we test the algorithms on the real-world instances taken from Milan airport. As can be seen in Table 4.9, the proposed EPS-SBH has the best results among the algorithms as it found 9 out of 12 of best solutions whereas EPS-AATCSR, ILS-AATCSR, ILS-SBH, SA-AATCSR and SA-SBH, respectively found 5, 2, 1, 0, 0 best solutions. In addition, this algorithm has the lowest average ERR value with 2.98. To analyze the results more precisely, a 95% confidence interval plot is performed that can be observed in Figure 4.14. The EPS-SBH has the best performance and the difference is statistically significant except with ILS-SBH.

Comparison on ALP Instances

ALP is a subset of ASP as it takes only landing operations into account. In this thesis, our focus is on ASP where both landing and takeoffs are considered. However, the ALP has attracted much greater research interest than than ATP, and many researchers have studied ALP. The ILS algorithm [13] is one of the latest and best performing algorithm in the ALP literature. So, besides the ASP, in this chapter, we also evaluate the three algorithms (SA, ILS, and EPS) on ALP instances. The ALP instances are obtained by replacing all takeoff operations with landing operations in the 75 ASP instances

TABLE 4.9: Performance of EPS, ILS, and SA using AATCSR and SBH for ASP on Milan airport instances

Problem Instances	EPS		ILS		SA	
	AATCSR	SBH	AATCSR	SBH	AATCSR	SBH
	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)
FPT01	**0.00**	3.33	6.67	3.33	34.67	6.67
FPT02	3.03	**0.00**	3.03	3.03	52.73	9.09
FPT03	14.29	**0.00**	14.29	14.29	61.90	14.29
FPT04	15.38	**0.00**	26.92	7.69	15.38	15.38
FPT05	8.33	**0.00**	25.00	4.17	37.50	12.50
FPT06	125.00	**0.00**	135.00	87.50	127.50	12.50
FPT07	12.50	**0.00**	12.50	**0.00**	91.25	18.75
FPT08	**0.00**	**0.00**	83.33	33.33	100.00	75.00
FPT09	**0.00**	5.56	16.67	27.78	41.11	38.89
FPT10	**0.00**	11.60	**0.00**	78.57	135.71	135.71
FPT11	114.29	**15.30**	114.29	114.29	255.71	157.14
FPT12	**0.00**	**0.00**	**0.00**	60.00	160.00	80.00
Average	24.40	**2.98**	36.47	36.16	92.79	47.99

Bold: minimum error

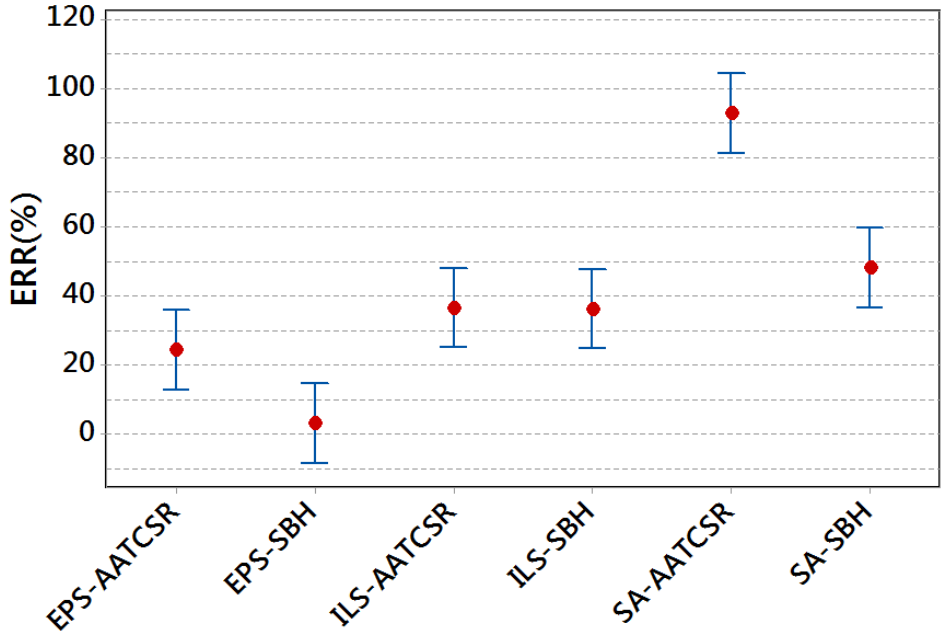


FIGURE 4.14: 95% confidence interval plot for ASP on Milan airport instances

generated by [2]. The results are given in Table 4.10 and a 95% confidence interval plot in Figure 4.15. As we see from these, EPS-SBH gives the best performance and significantly better than the other algorithms except EPS-AATCSR.

TABLE 4.10: Performance of the algorithms on ALP instances

Problem	EPS		ILS		SA	
Instances	AATCSR	SBH	AATCSR	SBH	AATCSR	SBH
$N, M, \#$	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)	ERR (%)
15, 2, 5	3.20	**0.90**	2.85	1.32	7.68	3.95
15, 3, 5	6.12	**0.00**	9.35	9.72	8.12	10.88
15, 4, 5	5.00	**0.00**	10.35	8.47	12.01	9.68
20, 2, 5	**4.02**	6.10	4.56	5.10	12.46	13.82
20, 3, 5	6.59	**2.04**	8.94	12.50	18.71	16.63
20, 4, 5	4.71	**0.00**	14.36	9.23	13.01	10.45
20, 5, 5	8.35	**3.89**	7.49	11.60	19.37	14.78
25, 2, 5	5.26	10.19	**1.10**	17.95	27.62	18.81
25, 3, 5	0.92	**0.83**	11.00	7.79	21.06	21.57
25, 4, 5	6.09	**0.74**	21.51	18.14	28.10	24.20
25, 5, 5	3.45	**0.00**	19.09	17.88	26.93	20.33
50, 2, 5	4.15	9.51	**2.80**	7.56	37.74	21.70
50, 3, 5	**1.39**	3.01	25.78	13.10	25.93	33.22
50, 4, 5	10.54	**9.50**	35.03	24.98	41.10	26.04
50, 5, 5	16.65	**1.20**	39.91	29.79	57.59	34.59
average	5.76	**3.19**	14.27	13.01	23.83	18.71

Number of instances for a given N, M pair Bold: minimum error

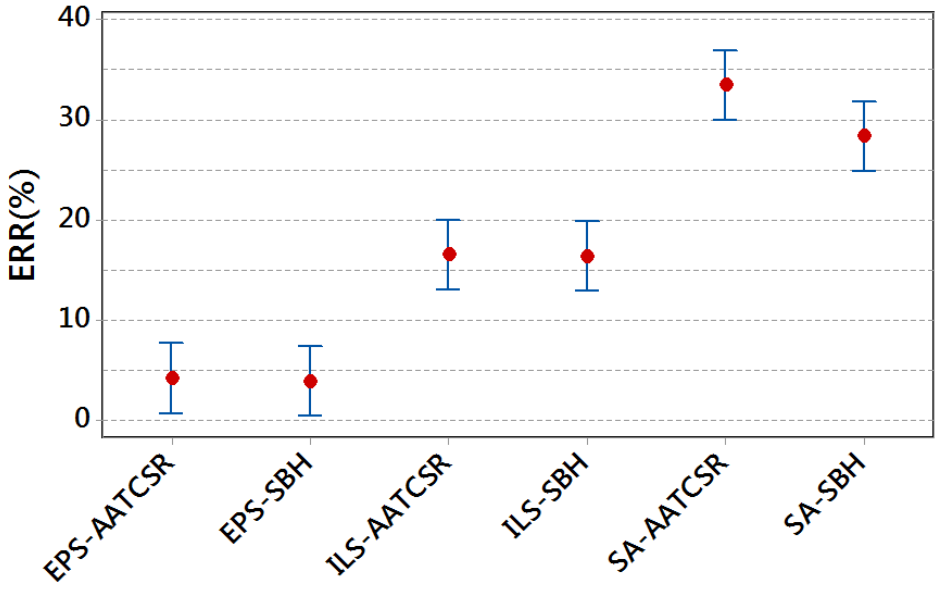


FIGURE 4.15: 95% confidence interval plot for ALP

4.3 Single-Runway Aircraft Scheduling

In this section, we are to exploit and explore the problem specific knowledge of the single-runway ASP. To that end, we propose a constraint-guided local search algorithm.

4.3.1 Single-runway Aircraft Scheduling Problem

Assume there are N aircraft $\{1, \ldots, N\}$ either arriving or departing and one runway to perform the operations on. At any time, the runway can be used by only one aircraft. To solve ASP, we have to determine the *operation time* OT_j of the aircraft j. There are two generic constraint categories: **hard** and **soft** constraints that are to be satisfied to produce a feasible schedule. The aim is to satisfy all the hard constraints and attempt to accommodate the soft constraints as much as possible in order to produce a high-quality schedule.

Hard constraints

- ***Time window:*** Because of several factors such as fuel restriction, airspeed, and possible manoeuvres, the operation time of each aircraft must lie within a specified time window. This time window is bounded by the *desired operation time* DOT_j and *latest operation time* LOT_j i.e. $OT_j \in [DOT_j, LOT_j]$.

- ***Safety separation time:*** Since each aircraft creates wake turbulence that the following aircraft need to avoid, a certain minimum separation time is required between any pair of aircraft. The separation times depend on the aircraft classes (heavy, large, and small) and the aircraft operation types (landing or takeoff). The separation times are determined by appropriate aviation authorities such as Federal Aviation Administration (FAA) in the United States or Civil Aviation Authority (CAA) in the United Kingdom [2]. Note that although the separation time constraints must hold between each pair of aircraft, it has been showed that using FAA standard, the separation times are automatically satisfied between two aircraft when there are three other aircraft in between. [137].

Soft constraints

Deviation from desired operation times: For each aircraft j, DOT_j is its desired operation time; which means that operation at that time has no delays and extra fuel burn. However, because of the capacity limit of runways and the hard constraints mentioned, some flights cannot operate at their DOT_j. Therefore, the operation times

of some flights are deferred from their desired times. If an aircraft j operates after its desired time DOT_j, it would be penalised by $(OT_j - DOT_j)$.

Objective function

The objective function of ASP is to minimise total delay cost of the aircraft resulting from the deviation of their operation times from the respective desired times. However, the delay cost of all aircraft is not the same. So, a *penalty weight* w_j per unit time delay from DOT_j for aircraft j. This *penalty weight* depends on two priorities: *operation priorities* and *aircraft size priorities*. Based on the *operation priorities*, arriving aircraft have greater priorities than departing aircraft because of the higher average fuel burn and safety measures. On the other hand, based on the *size priorities*, heavier aircraft get more weights than the lighter ones owing to again higher average fuel burn and safety measures.

One of the main challenges is how to calculate the operation time OP_j of each aircraft j. Assume π is the current sequence, $[k]$ represents the aircraft at the position k, and $s(j, j')$ shows the separation time between aircraft j and j'. So the operation times of aircraft can be calculated as follows:

$$s([k],[k']) = 0,\ OP_{[k]} = 0 \qquad k < 1 \vee k' < 1 \tag{4.7}$$

$$\begin{aligned} OP_{[k]} = \max\{&DOT_{[k]}, OP_{[k-1]} + s([k-1],[k]), \\ &OP_{[k-2]} + s([k-2],[k]), OP_{[k-3]} + s([k-3],[k])\} \qquad \forall k \in [2,n] \end{aligned} \tag{4.8}$$

The objective function is the total weighted tardiness of a schedule TWT $= \sum_{j=1}^{N} w_j(OT_j - DOT_j)$. This objective allows reduction of delays, maximisation of the runway capacity, and reduction of congestion at the airport [131].

4.3.2 Our Methodology

In order to solve this problem, we propose a constraint-guided local search (CGLS) algorithm. CGLS has two main steps: intensification and diversification. As the main contribution, unlike the most existing techniques in the literature, we use the specific

knowledge of the problem to design our algorithm. In the following sections, each step is described in detail.

The proposed local search algorithm is given in Algorithm 28. It starts with an initial solution. The initial solution is then improved by the intensification method. The algorithm next goes through the loop in which the search restarts with the diversification method. The new solution would be considered as the current solution if it is better in terms of the objective value.

Algorithm 28 Proposed CGLS Algorithm

1: $\pi \leftarrow$ Generate an initial solution.
2: $\pi \leftarrow$ Use the intensification method on π.
3: **while** termination criteria not satisfied **do**
4: $\quad \pi' \leftarrow$ Use the diversification method on π.
5: $\quad \pi'' \leftarrow$ Use the intensification method on π'.
6: $\quad$ **if** $TWT(\pi'') < TWT(\pi)$ **then** $\pi \leftarrow \pi''$.
7: **return:** the global best solution found so far.

Solution representation and initial solution

A single runway ASP solution is represented by a string of numbers containing a permutation of N aircraft, i.e., $\pi = \{[1], [2], ..., [N]\}$. The $[k]$ represents the aircraft at the kth position of the permutation. For example, for a problem with 7 aircraft, one possible solution is $\pi = \{3, 5, 4, 2, 7, 1, 6\}$; which means that aircraft 3 must be operated first, followed by aircraft 5, 4, 2, 7, 1 and aircraft 6.

As the initial solution, we use the simplest and the most common heuristic of aircraft sequencing, first-come-first-served (FCFS). In this method, the permutation of aircraft is based on a non-decreasing order of their *desired operation time*. Note that this very simple heuristic is still used in the air traffic control these days, e.g., in Doha International Airport [2]. However, it is not an efficient heuristic and can lead to the waste of resources and make the congestion in the terminal area severer [187]. Using this heuristic in the initialisation of the proposed algorithm can help find out how much improvement would be obtained by using the proposed method instead of the typical FCFS.

Intensification Method

To intensify the search, we propose an intensification method that is made up of two neighbourhood operators, instead of a single one. The reason is that different neighbourhood operators produce different landscapes and hence different local optima. We use insert and swap operators since they are widely used when solutions are represented as permutations e.g., in the flowshop scheduling problems [7, 30] and the order scheduling problems [34].

Let π be a permutation of the given N aircraft. In the operator $\mathsf{Insert}(\pi, j, k)$, aircraft at position j is selected and then inserted at a different position k. In the $\mathsf{Insert}(\pi, j)$, aircraft at position j is inserted at all k ($k \neq j$). On the other hand, in the $\mathsf{Swap}(\pi, j, k)$ operator, an aircraft at position j is exchanged with another aircraft at position k. However, in the $\mathsf{Swap}(\pi, j)$ operator, an aircraft at position j is exchanged with all aircraft at positions k ($k \neq j$).

In this paper, we use $\mathsf{Insert}(\pi, j)$ and $\mathsf{Swap}(\pi, j)$ operators mentioned in an iterative procedure. It means that they would be applied for all N aircraft in the permutation π, one by one, in a given order and as soon as a better solution is found, it would be considered as the current solution and the procedure is restarted with the new solution. We refer them to $\mathsf{Insert}(\pi)$ and $\mathsf{Swap}(\pi)$.

***Greedy Aircraft Selection*:** Our main contribution in the intensification method is to employ a greedy aircraft selection by using a constraint guidance. In the proposed greedy selection procedure, first, the weighted tardiness (WT_j) of each aircraft j in the current solution π is calculated. Then, the aircraft are sorted in a non-increasing order of WT_j in a reference list $\pi^{\mathcal{L}}$. Next, in the $\mathsf{Insert}(\pi)$ and $\mathsf{Swap}(\pi)$ operators, the aircraft are selected based on their order in the reference list $\pi^{\mathcal{L}}$. For instance, suppose for a problem with 6 aircraft, the current sequence is $\pi = \{2, 3, 6, 4, 5, 1\}$ and the reference list is $\pi^{\mathcal{L}} = \{4, 2, 6, 1, 3, 5\}$. The $\mathsf{Insert}(\pi)$ or $\mathsf{Swap}(\pi)$ operator first selects aircraft 4 for insertion or swap process from the sequence π. Then, it selects aircraft 2 from the sequence π. This process is continued until all aircraft in the reference list $\pi^{\mathcal{L}}$ are selected. The idea behind this greedy procedure is that aircraft with higher objective values should get more priorities over aircraft with lower objective values. Our idea is to reschedule these aircraft and thus fix the sequence.

The proposed Insert(π) and Swap(π) operators with greedy aircraft selection are given in Algorithm 29. With the use of the Insert(π) and the Swap(π) operators, the proposed intensification method is shown in Algorithm 30. At each iteration, it first applies N_1 on the current solution π. If the new solution obtained by N_1 is better than the current solution, it would be considered as the current solution and the process is again continued with N_1; otherwise the algorithm moves to N_2. The current solution would be updated if the new solution obtained by N_2 is better and algorithm also goes back to N_1; otherwise intensification phase is finished. Note that, in the intensification method, Insert(π) and Swap(π) operators are selected as N_1 and N_2 respectively based on the results obtained in the literature [179].

Algorithm 29 Insert(π) and Swap(π) operators

1: Let π be the input solution.
2: **foreach** aircraft j, calculate weighted tardiness WT_j, and sort them in the non-increasing order of WT_j to get a reference list $\pi^{\mathcal{L}} = (\pi_1^{\mathcal{L}}, \pi_2^{\mathcal{L}}, ..., \pi_N^{\mathcal{L}})$. .
3: **for** $k = 1$ to N **do**
4: $\pi_j \leftarrow$ The position of the aircraft $\pi_k^{\mathcal{L}}$ in π.
5: Apply Insert(π, π_j) or Swap(π, π_j) and take the permutation π' with the lowest total weighted tardiness.
6: **if** π' has a lower objective than π **then** let $\pi = \pi'$ and go to Step 2
7: **return:** π as the output solution.

Algorithm 30 Intensification Method

1: **Input:** sequence π.
2: Set Insert(π) as N_1 and Swap(π) as N_2. Also $l = 1$.
3: **while** $l \leq 2$ **do**
4: Find the best neighbour π' of π in $N_l(\pi)$.
5: **if** π' is better than π **then** Set $\pi = \pi'$ and $l = 1$.
6: **else** $l = l + 1$
7: **return:** π as the output solution.

Diversification Method

The proposed algorithm uses a diversification method to avoid getting stuck and convergence towards local optima and also to explore new areas in the solution space. Diversification method helps the algorithm generate new solutions for the intensification method by modifying the current solution instead of a fully random solution. The diversification procedure includes a number of moves, diversification strength λ, that are applied to the current local optimum. In this paper the diversification method also used two neighbourhood operators: Swap(π, j, k) and Insert(π, j, k). In this phase, for

each diversification move, with 50%-50% probabilities, we apply either $\mathsf{Swap}(\pi, j, k)$ or $\mathsf{Insert}(\pi, j, k)$ operators.

The value of the parameter λ is very important. A small λ may lead to the stagnation of the search and cycling among the previously visited solutions. On the other hand, a large λ may lead the algorithm to conduct like a random restart algorithm which in most cases generates low quality solutions. Therefore, we carefully calibrate the parameter λ.

***Bounded-Diversification Technique*:** Unlike the typical diversification procedure that moves the selected aircraft to the completely randomly selected positions, we inject the problem specific knowledge into this method to find diverse as well as reasonable positions for the selected aircraft. As mentioned already, ASP has two types of hard constraint including *time window* constraint that forces each aircraft j to be operated within a window, i.e, $OT_j \in [DOT_j, LOT_j]$. Being operated the more closer to DOT_j leads to the less penalty value. Therefore, moving an aircraft to a position that is far from its current position could not be very effective and reasonable. Therefore, in this paper, we propose a bounded-diversification technique that does not allow a selected aircraft to move far away from its current position. To that end, we introduce a parameter γ that controls the position of each selected aircraft. In detail, when an aircraft at position j is selected for diversification, it could be moved just to the position k such that $\mathsf{max}(1, j-\gamma) \leq k \leq \mathsf{min}(N, j+\gamma)$. Similar to λ, this parameter is also carefully calibrated. The procedure of the proposed diversification method is given in Algorithm 31.

Algorithm 31 Proposed bounded diversification method

1: **Input:** Solution π, the diversification strength λ, the diversification bound γ.
2: **for** $h = 1$ to λ **do**
3: $\quad j \leftarrow$ pick a random position
4: $\quad k \leftarrow$ pick another random position from $[\mathsf{max}(1, j-\gamma), \mathsf{min}(N, j+\gamma)]$
5: $\quad$ **if** rand() ≤ 0.5 **then** $\pi \leftarrow \mathsf{Insert}(\pi, j, k)$
6: $\quad$ **else** $\pi \leftarrow \mathsf{Swap}(\pi, j, k)$
7: **return:** π as the output solution.

4.3.3 Experimental Results

In order to evaluate the performance of the proposed algorithm, we use 20 well-known instances generated based on the Doha International Airport parameters [2]. These

instances are made up of 50 aircraft and time windows of 30 minutes. We compare our algorithm with ILS algorithm [13] (called here as ILS-SK) as one of the leading algorithms for the single runway ASP. The ILS-SK algorithm uses a variant of FCFS as initialisation. In this paper, to have a fair comparison, we use the FCFS as initialisation of the ILS-SK as well. Both algorithms have been implemented in C++ language and on top of the constraint-guided local search system, Kangaroo [185]. The functions and the constraints are defined by using invariants in Kangaroo. Invariants are special constructs that are defined by using mathematical operators over the variables. Algorithms are also tested on the same computer.

To compare the performance of the algorithms, we use the relative percentage deviation $RPD = \frac{\text{TWT}^A - \text{TWT}^{BEST}}{\text{TWT}^{BEST}} \times 100$ where TWT^A is the total weighted tardiness obtained by algorithm A and TWT^{BEST} is the best total weighted tardiness achieved by any of the algorithms compared. We run each algorithm on each instance 5 times and compute average RPD (ARPD) over the 5 runs. We also compute a further average of RPDs or ARPDs over all instances in a benchmark set. As a stopping criterion, the algorithms were run for $20N$ ms CPU time.

CGLS Parameter Calibration

The proposed CGLS contains two parameters: the diversification strength λ, and the diversification bound γ. To analyse the effect of these two parameters, a full factorial design is used by considering 3 different values for each parameter: $\lambda \in \{10, 20, 30\}$ and $\gamma \in \{3, 4, 5\}$. For this experiment, we randomly select 8 instances from those 20 instances in our benchmark. Our algorithm is run 5 times for each of the $3 \times 3 = 9$ settings and for each instance with the same stopping criterion as already mentioned.

The 95% confidence interval plots of the parameters are shown in Figure 4.16. The results of Figure 4.16 says that CGLS algorithm is robust with respect to λ and γ as the tested values are statistically equivalent and each of them could be selected. However, since the λ and γ have lower ARPD in 20 and 4 respectively, these values are selected for further experiments.

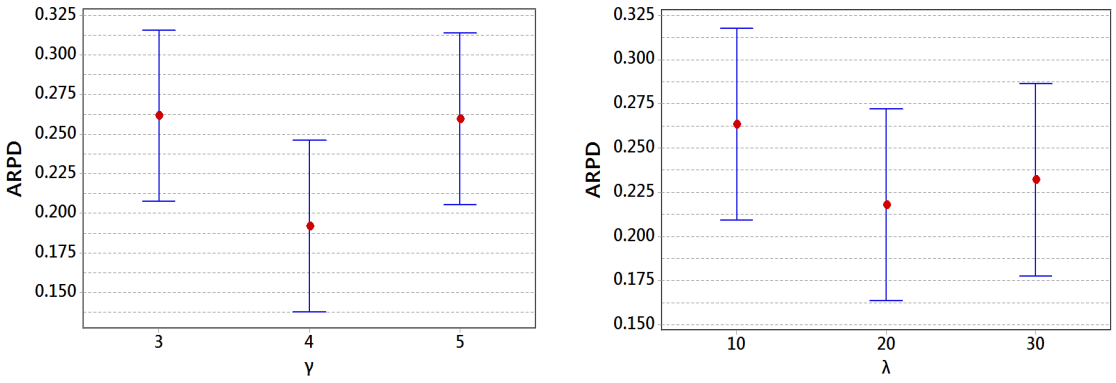


FIGURE 4.16: Mean and 95% confidence intervals for parameters.

Effectiveness of Multi Neighbourhood

The proposed intensification method includes two neighbourhoods N_1 and N_2. In this paper, we use insertion and swap operators with greedy aircraft selection, GI and GS respectively. In this section, we are to evaluate the efficiency of the greedy neighbourhoods against the random insertion and swap operators, RI and RS, and also to find the best order for the neighbourhood operators mentioned. To that end, the following four cases are considered:

1. **Case 1:** Consider GI as N_1 and GS as N_2.
2. **Case 2:** Consider GS as N_1 and GI as N_2.
3. **Case 3:** Consider RI as N_1 and RS as N_2.
4. **Case 4:** Consider RS as N_1 and RI as N_2.

In this experiment, the proposed CGLS is tested by considering each of the cases mentioned as the intensification method on those 8 instances used already for parameter tuning. The 95% confidence interval plot for each case is given in Figure 4.17. From this figure, it can be seen that cases 1 and 2 are significantly better than cases 3 and 4. It can be concluded that the proposed problem-dependent greedy strategies for Insertion and Swap moves statistically outperform the random cases. In addition, although cases 1 and 2 are statistically equivalent, we use case 1 for the intensification phase due to its lower ARPD.

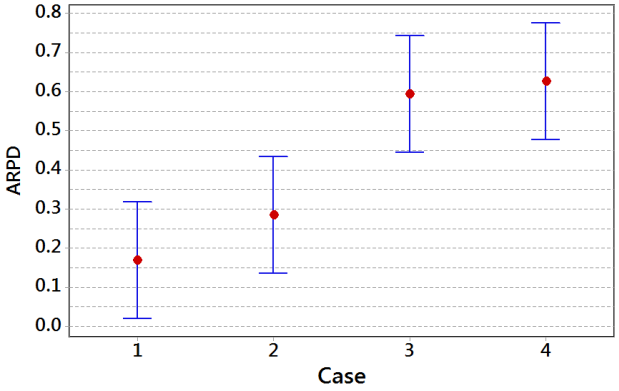


FIGURE 4.17: 95% Confidence intervals for CGLS variants.

Effectiveness of CGLS Components

CGLS has two main contributions: a new constraint based greedy aircraft selection in the neighbourhood operators of the intensification method, and a constraint based bounded-diversification procedure. To test the effectiveness of each component mentioned, we create three variants of CGLS as follows:

1. **CGLS:** Proposed CGLS that includes both greedy intensification and bounded-diversification.
2. **CGLS_R:** CGLS but greedy intensification is replaced by a random one.
3. **CGLS_NB:** CGLS but no bound in the diversification phase.

The algorithms are tested on the 8 instances which are the same as the ones for parameter tuning. A 95% confidence interval plot in Figure 4.18 is carried out to show the effectiveness of the three variants. Note that non-overlapping confidence intervals of each two methods represent a statistically significant difference between them. From Figure 4.18, we can see that both new components significantly affect the performance of CGLS. Among these two components, the bounded-diversification is more crucial as the algorithm obtained worse performance with the absence of this method.

Comparison with FCFS Method

As mentioned before, the first-come-first-served (FCFS) heuristic is the simplest and the most common heuristic for aircraft sequencing, and is still applied in the air traffic

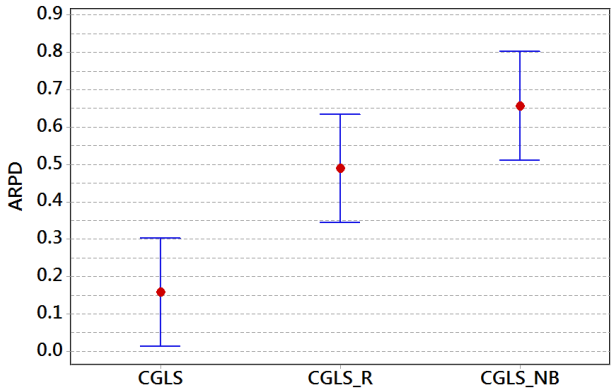


FIGURE 4.18: 95% Confidence interval for CGLS variants.

control these days, e.g., in Doha International Airport [2]. As a result, comparing CGLS with FCFS can show how much the proposed CGLS improves over FCFS. The results are shown in Table 4.11. As can be seen from this table, CGLS hugely outperforms the FCFS obtaining ARPD of 0.157% compared to 99.353% of FCFS.

TABLE 4.11: Comparison of CGLS and FCFS algorithms.

instance	1	2	3	4	5	6	7	8	9	10	11
FCFS	155.67	138.61	212.54	160.93	178.10	102.55	133.63	75.99	90.06	82.97	68.75
CGLS	**0.29**	**0.60**	**0.26**	**0.14**	**0.39**	**0.00**	**0.07**	**0.20**	**0.23**	**0.00**	**0.23**
instance	12	13	14	15	16	17	18	19	20	Average	
FCFS	67.29	71.41	79.03	43.40	59.31	60.14	75.83	69.23	61.64	99.35	
CGLS	**0.10**	**0.06**	**0.07**	**0.00**	**0.08**	**0.05**	**0.28**	**0.04**	**0.13**	**0.16**	

Comparison with the State-of-the-art Method

We compare the results of CGLS with the results of ILS-SK algorithm [13] shown in Table 4.12. In this table, besides the ARPD, we also show the number of times each algorithm finds the TWT^{BEST} (the best total weighted tardiness achieved by any of the tested algorithms) for each instance out of 5 runs. As can be seen, CGLS outperforms ILS-SK i.e., it achieves lower ARPD in 19 instances out of 20. In addition, except in instance 7, CGLS obtains the TWT^{BEST} in all instances at least once, while ILS-SK finds the TWT^{BEST} only in 6 instances out of the 20. To examine the difference of the algorithms statistically, we also perform a student t-test with significance level of $\alpha = 0.05$. Statistical results confirm a significant difference between CGLS and ILS-SK since p-value $= 0.00 < 0.05$

TABLE 4.12: Comparison of CGLS and ILS-SK algorithms

Instance	CGLS		ILS-SK	
	ARPD	#best	ARPD	#best
1	**0.287**	3	3.222	0
2	**0.597**	1	1.119	0
3	**0.257**	3	1.427	0
4	**0.140**	2	1.440	1
5	**0.387**	2	4.230	0
6	**0.000**	5	0.361	1
7	**0.067**	0	0.149	2
8	**0.199**	2	0.668	0
9	**0.229**	1	1.415	0
10	**0.000**	5	0.641	0
11	0.225	2	**0.149**	1
12	**0.103**	1	1.117	0
13	**0.061**	2	0.240	0
14	**0.071**	2	0.150	1
15	**0.000**	5	0.207	0
16	**0.007**	3	0.292	0
17	**0.053**	1	0.244	1
18	**0.275**	2	0.431	0
19	**0.044**	2	0.751	0
20	**0.134**	1	0.355	0
Average	**0.157**		0.930	

4.4 Conclusions

This chapter presents a constraint-based approach to solve the aircraft scheduling problem. we have proposed two new constructive heuristics and a new constraint guided local search algorithm for ASP.

The two proposed constructive heuristics are evaluated against the best known constructive heuristic AATCSR [15]. Table 4.4 shows that the proposed heuristics obtained ARPDs 8.9% and 6.7% while AATCSR obtained 59.9%. AATCSR constructs the solution simply by appending each next aircraft based on a precomputed index without performing any search. In contrast, the two proposed heuristics perform search on the already constructed partial solutions using the objective function to find the best positions for the next aircraft. The guided search in the two proposed heuristics clearly makes the difference between their performance and that of AATCSR, which does not have any search.

The proposed local search algorithm embeds constraint-guided strategies in its exploration and perturbation phases. Taking the time constraint into account, an adjacent swap operator is used in the exploration phase instead of using a typical intra-swap

operator. The adjacent swap exchanges two aircraft that in successive positions in a runway while the intra-swap exchanges any two aircraft in a runway. Taking the objective function into account, runways that have the most objective value are selected for the operators to be applied on. The runway selection allows the search to fix the most problematic part of the current solution. window, moving an aircraft to a position that is far from its current position might not be effective and reasonable. The latter one swaps two aircraft that are in two different runways. In this operator, instead of the typical random runway selection, we insert the greediness into it by selecting one of the runway greedily, the one with highest objective function created. This allows us to fix the most problematic part of the current solution. Results in Figures 4.7, 4.8, and 4.9 demonstrate that the proposed constraint-guided operators and greedy runway selections are significantly better than the typical random counterparts. Table 4.7 shows that our constraint guided local search significantly outperforms SA [15] and ILS [13] as they obtain 2.15%, 22.52%, and 35.52% ARPDs respectively on the instances from [2]. Note that the SA and ILS algorithm do not uses any constraint guided strategies and depend on random or exhaustive selections. Table 4.9 shows similar performances on instances from [186] while Table 4.15 shows on ALP instances.

In this chapter, we also have done another investigation as algorithms are evaluated on different runway capacity combinations. This experiment helps up see the performance of the algorithms on different runway combination scenarios and to find the best runway capacities. Table 4.8 shows that in all problem sizes, the lowest standard deviations help find the best solutions. Moreover, the proposed algorithm outperforms the other algorithm significantly in all runway capacity combinations.

Finally, we also study the ASP with single-runway case. To solve this problem, we also propose a constraint-guided local search algorithm that advances ASP search by injecting the specific knowledge of the problem into its different phases. In the intensification phase, we propose a greedy approach that gives more priorities to aircraft that are more problematic and create more delays. In the diversification phase, we employ a bounded-diversification technique that controls the new position of each selected aircraft and does not allow them to move very far away from their current positions. Computational results show that the proposed algorithm outperforms the existing state-of-the-art methods with considerable margin.

Chapter 5

Customer Order Scheduling

This chapter focuses on extracting and exploiting some structural knowledge of customer order scheduling problem (COSP). COSP has many applications that include the pharmaceutical and the paper industries. However, most existing COSP algorithms struggle to find very good solutions in large-sized problems. One key reason behind is that those algorithms are based on generic templates and as such lack problem specific structural knowledge. In this chapter, we capture such knowledge in the form of heuristics and then embed those heuristics within constructive and perturbative search algorithms. In the proposed deterministic constructive search algorithm, we use processing times in various ways to obtain initial dispatching sequences that are later used in prioritising customer orders during search. We also augment the construction process with solution exploration. This chapter is based on following publications.

- Vahid Riahi, MA Hakim Newton, MMA Polash, and Abdul Sattar. 'Tailoring customer order scheduling search algorithms.' Computers & Operations Research 108 (2019): 155-165.
- Vahid Riahi, MMA Polash, MA Hakim Newton, and Abdul Sattar. 'Mixed neighbourhood local search for customer order scheduling problem.' In Pacific Rim International Conference on Artificial Intelligence (PRICAI), pp. 296-309. Springer, Cham, 2018.

- Vahid Riahi, MA Hakim Newton, and Abdul Sattar. 'Customer order scheduling by scattered wolf packs.' In 7th International Conference on Metaheuristics and Nature Inspired Computing, Morroco, 2018.

5.1 Customer Order Scheduling Problem

The COSP with the total completion time TCT objective is formally formulated as follows. It has n different customer orders and m machines arranged in a parallel layout. Each customer order comprises m jobs (product types) in which each job can be processed on one dedicated machine without interruptions or preemptions. Let p_{ij} denote the processing time of ith job of customer order j to be processed on machine i. A machine can at any time process at most one customer order. Moreover, a machine can start processing another customer order as soon as it completes one. The aim is to find a permutation of customer orders that TCT is minimised.

It is proved that there is always an optimal solution where all machines process the orders in the same sequence [27]. So, a COSP solution is represented by a permutation π of the customer orders indicated the sequence of the orders are executed. Let $[k]$ denote the customer order at the kth position in π. Also, let the completion time point of ith job of customer order $[k]$ at machine i is calculated as $C_{i[k]} = C_{i[k-1]} + p_{i[k]}$ where $C_{i[0]} = 0$. So the completion time point of a customer order $[k]$ is $C_{[k]} = \max_{i=1}^{m} C_{i[k]}$ and the total completion time of a solution π is $\mathsf{TCT}(\pi) = \sum_{i=k}^{n} C_{[k]}$. Computing $\mathsf{TCT}(\pi)$ from scratch clearly needs $\mathcal{O}(mn)$ time. Figure 5.1 depicts an example COSP with 4 customer orders and 3 machines. Nevertheless, the COSP is to find a permutation π of the customer orders such that their total completion time is minimised.

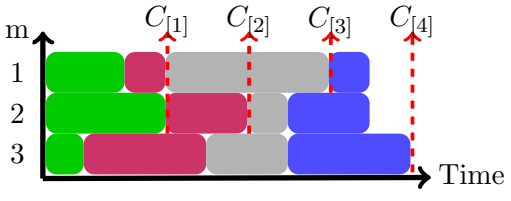


FIGURE 5.1: An example COSP with 4 customer orders and 3 machines

5.2 Preliminaries

COSP has several real-life applications that include car repair shops [25], pharmaceutical industry [26], paper industry [27], and manufacturing of semi finished lenses [24]. Consider a car repairing shop having several mechanics. Each mechanic can do only a particular type of repairing. However, each arriving car needs several types of repairing. So several mechanics work on each car simultaneously to reduce the completion time for the car. As soon as a mechanic finishes his task on a car, s/he can start working on the next car. A similar model can be considered for ship repairing or aircraft maintenance shops [25]. Despite existence of these realistic applications, COSP has not been studied much and so not much progress has been achieved.

Since COSP with TCT objective is NP-hard [188], incomplete algorithms have been developed in the literature using heuristics and metaheuristics. One of the best existing heuristics is *Earliest Completion Time* (ECT) [27]. Starting from an empty sequence, at each iteration ECT appends an unscheduled customer order to the already scheduled partial sequence such that TCT of the new partial sequence is the lowest. The time complexity of ECT is $\mathcal{O}(mn^2)$. A very recent heuristic FP (named after the authors' names) is different from ECT in that TCT is computed for the entire sequence assuming unscheduled customer orders as the trailing ones in the sequence [9]. The time complexity of FP is $\mathcal{O}(mn^2)$. One possible criticism to both ECT and FP is that they consider placing an unscheduled customer order only at the end of the scheduled partial sequence while other positions between the scheduled jobs could also be considered. Among the metaheuristic algorithms, a tabu search (TS) [27] starts from an ECT solution and iteratively improves further by using swap operators. Moreover, a greedy search algorithm (GSA) [9] starts from an FP solution and performs diversification and intensification repeatedly in an interleaving fashion. In the intensification, GSA explores an exhaustive swap neighbourhood while in the diversification it removes a random customer order, appends it at the end, and then considers inserting each of the other customer orders immediately before it.

Besides the above-mentioned algorithms which fully proposed for COSP with TCT objective, there are some other studies which focused on other COSP variants. An LP-based approximation algorithm is proposed for COSP with minimising the number of late jobs as objective [189]. In fact, it is assumed that each customer order has a due

date and they tried to minimise number of jobs operated after their due dates. The COSP is studied with unrelated parallel machine which means that each machine is capable of processing all the jobs [28]. The total tardiness COSP with a learning effect is studied [190]. In that model, they considered that each customer order includes a due date. Also, they considered a learning rate which means that a job is executed in its normal processing times only when it is scheduled first, otherwise its processing times will be shorter than the normal processing times [191]. Some methods including a branch-and-bound algorithm, simulated annealing, and particle swarm optimisation are proposed. A branch-and-bound algorithm as well as particle swarm optimisation for two-agent multi-facility order scheduling with ready times are proposed [192]. In another study, two objectives e.g., total flowtime and maximum tardiness, are considered for COSP and a local search-based algorithm and a population-based algorithm called particle swarm colony (PSC) algorithm is proposed in order to solve the problem [193]. Very recently, the assembly scheduling problems is reviewed [194] and showed that the COSP is classified as a special case of assembly scheduling problems.

5.3 Proposed Scattered Wolf Pack Algorithm

The proposed scattered wolf pack (SWP) population-based algorithm combines various elements of SS and GWO. We borrow three ideas from SS: keeping diverse solutions in the population, combining good solutions with diverse solutions, and applying a local search algorithm on generated solutions. From GWO, we borrow the idea of keeping only three best solutions, and combining these three best solutions with each diverse solution. Moreover, we use a multi-neighbourhood local search to escape from local optima. Experimental results using well-known instances indicate the effectiveness of the proposed SWP algorithm.

5.4 Proposed Constructive Heuristics

As discussed before, existing constructive heuristic algorithms such as ECT [27] and FP [9] place an unscheduled customer order only at the end of the already scheduled sequence. In our view, the end of the partial schedule actually might not be the best position for that unscheduled customer order to be placed, some other earlier positions

might be better suited. Those algorithms, however, try each unscheduled customer order and select one of those that results in the best objective value of the new partial sequence.

In our permutation construction and exploration algorithm (PCE) in Algorithm 32, we consider all possible positions in the partial sequence for an unscheduled customer order to be placed in with a view to finding a better position (Lines 3 and 5). Moreover, we consider the unscheduled jobs one by one as they appear in a predefined list $\mathcal{L}$ that is constructed using the processing times of the customer orders (Line 1). The predefined list essentially captures problem instance specific characteristics of a given COSP. We consider 8 ways to create $\mathcal{L}$ and these are described below.

While PCE follows a typical insertion based iterative permutation construction procedure (Lines 5–9), but it then augments the procedure at each iteration by applying swapping based exploration of the same customer order (Line 11–14). The augmented exploration does not increase the over complexity level of the algorithm. The integration of such exploration as part of a constructive method is, we believe, new in the literature, even considering similar other problems. Note that the condition $k' \notin [k_* - 1, k_* + 1]$ in Line 11 helps avoid revisitation of the permutations created in Lines 5–9.

To summarise, we make two contributions in our PCE algorithm. As part of the construction process, we find the best position for the best unscheduled customer order in the scheduled partial sequence. Also, we augment the construction process with an exploration procedure.

Figure 5.2 shows an example of insertion based construction followed by swap based exploration in PCE; which is just one lap of the loop in Line 3.

Constructing List $\mathcal{L}$

We propose the following 8 ways to construct $\mathcal{L}$. The rough idea is to put a customer order early in $\mathcal{L}$ if it has "short" processing times or has "less" variations in its processing times. This is because an earlier customer order in the solution has an cumulative effect on a later one, since total completion time of all customer orders is the objective. The proposed rules below quantify the abstract measures in various ways.

Algorithm 32 Permutation Construction and Exploration (PCE)

$\mathcal{L} \leftarrow$ a sequence of customer orders created using a predefined rule
$\pi \leftarrow \varnothing$ ▷ start from an empty solution
for $k = 1$ to n **do** ▷ iterate over customer orders in $\mathcal{L}$
  $\pi_* \leftarrow \varnothing$ with $\mathsf{TCT}(\varnothing) = \infty$, $k_* = 0$ ▷ to track best permutations
  for $k' = k$ to 1 **do** ▷ a position in π to insert $\mathcal{L}[k]$
    $\pi' \leftarrow$ insert customer order $\mathcal{L}[k]$ at position k' in π,
      shifting those at position k' or later towards the end.
    if $\mathsf{TCT}(\pi') < \mathsf{TCT}(\pi_*)$ **then** ▷ if better than the best
      $k_* \leftarrow k'$, $\pi_* \leftarrow \pi'$ ▷ update the best
  $\pi \leftarrow \pi_*$. ▷ take the best as the current
  for $k' = 1$ to k but $(k' \notin [k_* - 1, k_* + 1])$ **do** ▷ avoid revisitation
    $\pi' \leftarrow$ swap customer orders at positions k_* and k' in π
    if $\mathsf{TCT}(\pi') < \mathsf{TCT}(\pi_*)$ **then** ▷ if better than the best
      $\pi_* \leftarrow \pi'$ ▷ update the best
  $\pi \leftarrow \pi_*$. ▷ take the best as the current
return π as the output solution

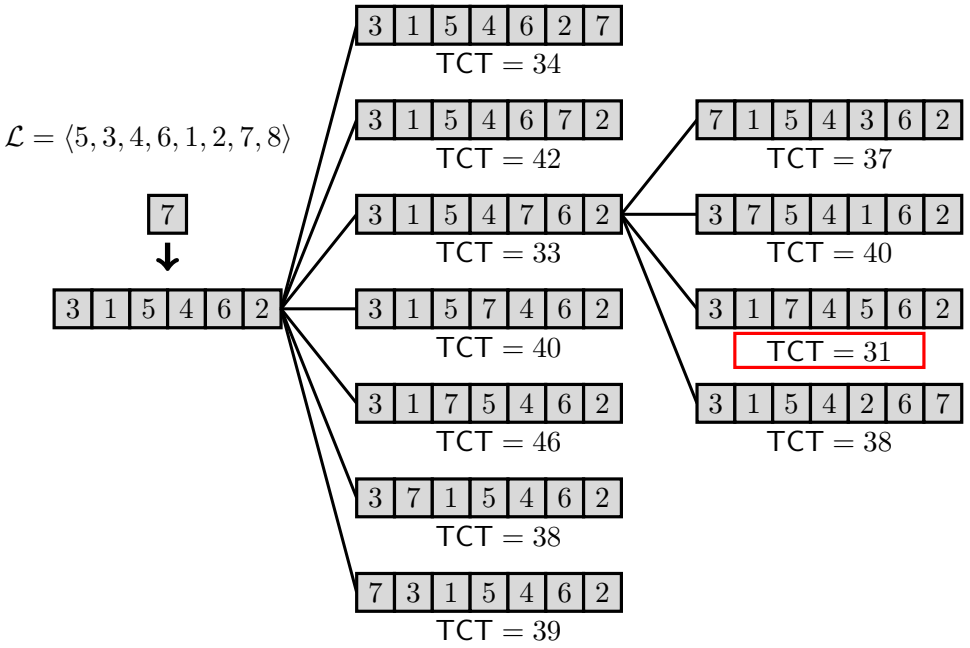


FIGURE 5.2: Construction by insertion and exploration by swap in PCE; TCTs are nominal.

- **Machine with the Highest Total Processing Time (HM):** Find the machine i' such that $i' = \text{argmax}_i \sum_{j=1}^{n} p_{ij}$. Such a machine determines the makespan of any schedule for a given COSP. Arrange the customer orders in the non-decreasing arrangement on their processing times $p_{i'j}$ at machine i' to obtain $\mathcal{L}$; the shorter the earlier. The time complexity of using this rule is $\mathcal{O}(nm + n\log n)$.

- **Machine with the Lowest Total Processing Time (LM):** Find the machine i' such that $i' = \text{argmin}_i \sum_{j=1}^{n} p_{ij}$. Such a machine determines the minimum time period when each machine is busy; after that, some machines would be idle. Arrange the customer orders in the non-decreasing arrangement on their processing times $p_{i'j}$s at machine i' to obtain $\mathcal{L}$; the shorter the earlier. The time complexity of using this rule is $\mathcal{O}(nm + n\log n)$.

- **Machine with the Lowest Total Completion Times (LC):** For each machine i, obtain a list $\mathcal{L}_i$ of customer orders from their non-decreasing arrangement on the processing times p_{ij}s on machine i. Then, take $\mathcal{L}_i$ as $\mathcal{L}$ where $\mathsf{TCT}(\mathcal{L}_i)$ is the minimum. The time complexity of using this rule is $\mathcal{O}(nm^2 + mn\log n)$. This rule is used by [9].

- **Machine with the Lowest Weighted Total Completion Time (WLC):** This rule is a variant of LC. The only difference is $\mathcal{L}_i$ for each i is obtained by arranging customer orders on non-decreasing order of $w_j p_{ij}$s instead of just p_{ij}s where $w_j = \sum_{i=1}^{m} p_{ij} / \sum_{j=1}^{n} \sum_{i=1}^{m} p_{ij}$. The intuition behind this rule is to put the same level of emphasis (i.e. w_j) on a customer order on each machine and the emphasis is based on the relative total processing time of the customer order with respect to the total for all customer orders. The time complexity of using this rules is the same as that of using LC.

- **Standard Deviation of Processing Times (SD):** In this rule, for each customer order j, the standard deviation σ of the processing times is calculated from $\sigma_j^2 = \sum_{i=1}^{m} (p_{ij} - \mu_j)^2/(m-1)$, where mean $\mu_j = (\sum_{i=1}^{m} p_{ij})/m$. Then, the customer orders are sorted in a non-decreasing order of the σ_j. The intuition is to give priority to the customer orders with less deviation in the processing times. Such customer orders early will not cause much dispersion of the processing times of the same customer order over all the machines. Thus the time period when a

customer order gets processing from at least one machine is reduced. The time complexity of using this rule is $\mathcal{O}(nm + nlogn)$.

- **Total Processing Times (TPT):** For each customer order j, compute the total processing time $\mathsf{TPT}_j = \sum_{i=1}^{m} p_{ij}$ over all machines, and then sort the customer orders in the non-decreasing arrangement of TPT_j. The time complexity of using this rule is $\mathcal{O}(nm + nlogn)$. This rule is used by [195] to construct a solution directly while we use it just to obtain a predefined list that determines which customer order is inserted and swapped in the each iteration of PCE.

- **Total Weighted Processing Times (TWPT):** This rule is a variant of TPT. The only difference is customer orders are sorted on the non-decreasing order of $\mathsf{TWPT}_j = \sum_{i=1}^{m} p_{ij} \times w_i$ instead of $\mathsf{TPT}_j = \sum_{i=1}^{m} p_{ij}$ where $w_i = {\sum_{j=1}^{n} p_{ij}}/{\sum_{j=1}^{n}\sum_{i=1}^{m} p_{ij}}$. The intuition behind this rule is that some machines are more important than others in terms of their relative total processing times measured in w_i. The time complexity of using this rules is the same as that of using TWPT.

- **Total Completion Weighted Processing Time (TCWPT):** This rule is a variant of TWPT. The only difference is w_i is calculated in a different way. For each machine i, obtain a list $\mathcal{L}_i$ of customer orders from their non-decreasing arrangement on the processing times p_{ij}s on machine i. Then, for each machine i, compute $w_i = {\mathsf{TCT}(\mathcal{L}_i)}/{\sum_{i=1}^{m} \mathsf{TCT}(\mathcal{L}_i)}$. The time complexity of using this rule is $\mathcal{O}(nm^2 + mnlogn)$.

Lemma 5.1. *PCE has a time complexity of* $\mathcal{O}(n^3m)$.

Proof. Obtaining list $\mathcal{L}$ in the worst case needs $\mathcal{O}(nm^2 + mnlogn)$ time. The nested two level loops and TCT computation altogether needs $\mathcal{O}(n^3m)$ time. So the overall complexity is $\mathcal{O}(n^3m)$. □

5.5 Proposed Perturbative Search Algorithm

As mentioned before, existing COSP algorithms struggle to find very good solutions in large-sized problems. One key reason is these algorithms are based on generic templates and as such lack problem specific structural knowledge. As a result, they either go for

random decisions or for exhaustive neighbourhood exploration. Fully random decisions normally lead to poor performance. On the other hand, exhaustive neighbourhood exploration is very costly and very greedy in nature. These lead to wastage of search effort and premature search convergence. Overall, maintaining diversity level is in general a key issue in a perturbative search. Our proposed perturbative search algorithm intensifies its diversification phase and diversifies its intensification phase. We define a measure of extra time a customer order stays under processing and use this measure in the diversification phase to select customer orders that should be rescheduled. It helps us to identify the customer orders that are very badly affected in a given solution. By this, we try to make informed decisions when the search has lost its direction in a local minimum or in a plateau. In the intensification phase, we use multiple neighbourhood operators randomly so the search can get out of the local minima for one neighbourhood operator using another neighbourhood operator as an escape route.

We apply our new strategies on top of a typical perturbative search algorithm (PSA) shown in Algorithm 33. PSA is a single solution iterative search algorithm that explores the solution space. It has four key elements: initialisation, intensification, diversification, and acceptance. PSA starts from an initial solution, in this case, from the solution generated by our constructive heuristic PCE. The initial solution is then improved by using an intensification method. Then, within a loop, the current solution undergoes a diversification method and then the same intensification method in an interleaving fashion. The resultant solution in each iteration replaces the current solution, if the acceptance method accepts it. We describe our intensification, diversification, and acceptance methods in details.

Algorithm 33 Perturbative Search Algorithm

1: $\pi \leftarrow$ initialiseSolution()
2: $\pi \leftarrow$ performIntensification(π)
3: **while** *termination criteria not satisfied* **do**
4: $\quad \pi' \leftarrow$ performDiversification(π)
5: $\quad \pi'' \leftarrow$ performIntensification(π')
6: $\quad \pi \leftarrow$ performAcceptance(π'', π)
7: **return** The global best solution found so far

5.5.1 Intensification Method

Algorithm 34 describes our proposed performIntensification() method. It starts from an input solution π. For a given number of iterations N, it then applies a randomly selected neighbourhood operator from a set of such operators $\mathcal{N}$. If the result solution π' is better than the current solution π in terms of TCT, then π' replaces π. We use one of the operators at a time from $\mathcal{N}$. This is because applying all of them every time is very time costly. Also, using one at a time helps the search get out of local optima of another one used before. In fact this is the reason, we use multiple neighbourhood operators in this way to diversify the intensification phase.

Algorithm 34 performIntensification()

1: Let π be the input solution to undergo intensification
2: Let $\mathcal{N}$ be the given set of neighbourhood operators
▷ We assume $\mathcal{N} = \{\mathsf{insert}, \mathsf{swap}, \mathsf{insert\text{-}pair}, \mathsf{swap\text{-}pair}\}$
3: **for** a given number of iteration N **do** ▷ We assume $N = n(n-1)/2$
4: $\mathsf{oper} \leftarrow$ select an operator uniformly randomly from $\mathcal{N}$
5: $\pi' \leftarrow$ apply operator oper on the solution π
6: **if** $\mathsf{TCT}(\pi') < \mathsf{TCT}(\pi)$ **then** $\pi \leftarrow \pi'$
return π as the best solution found

In this chapter, we use the following four neighbourhood operators. These operators are common in problems where the solutions can be represented by permutations, e.g. in flowshops [30].

$\mathsf{insert}(\pi, k, k')$: the customer order at position k in π is at position k' in the resultant permutation π'. If $k > k'$, any customer orders at positions $k' \leq k'' < k$ move to positions $k'' + 1$. If $k < k'$, any customer orders at positions $k < k'' \leq k$ move to positions $k'' - 1$.

$\mathsf{swap}(\pi, k, k')$: swap two customer orders at positions k and k' in π to obtain the resultant permutation π'.

$\mathsf{insert\text{-}pair}(\pi, k, k')$: two customer orders at positions k and $k+1$ are moved to positions k' and $k'+1$ in the resultant permutation π' where $|k - k'| \geq 2$. If $k > k'$, any customer orders at positions $k' \leq k'' < k$ move to positions $k'' + 2$. If $k < k'$, any customer orders at positions $k + 1 < k'' \leq k' + 1$ move to positions $k'' - 2$.

$\mathsf{swap\text{-}pair}(\pi, k, k')$: swap two customer orders at positions $k + 1$ and $k' + 1$ in π along with swapping those at positions k and k' to obtain the resultant permutation π

where $|k - k'| \geq 2$. This operator can be considered as doing two swap moves, $\mathsf{swap}(\pi, k, k')$; $\mathsf{swap}(\pi, k+1, k'+1)$.

Whenever one of the four operators selected we also uniformly randomly select positions for the parameters k and k'. For the number iterations, we set $N = n(n-1)/2$ since this value is equal to the number of possible swaps in an exhaustive swap based neighbourhood operator and is used by the state-of-the-art GSA algorithm [9]. We later experimentally compare our proposed PSA with GSA.

Given all four operators described above, customer orders at positions before $k'' = \min(k, k')$ remain the same before and after the application of a selected operator. So in our implementation, while calculating the total completion time for the resultant permutation, we reuse the completion times of these customer orders and recompute only for those after position k''.

Note that the proposed method has some similarities and differences with VND (see Section 2.1.2). Both methods include multiple neighbourhood operators. It helps them to search in larger space and escape from local optima. However, there are some differences as well. VND uses k neighbourhood operators in a given order, and starts from first operator N_i. After applying an iterative improvement steps, If better solution is not found, it goes through the second one N_2, and this process continue until all operators are examined. However, whenever a better solution is found by N_i, VND backs to N_1 and restarts the process. In our method, we are using the operators without any specific order. In fact, algorithm can pick randomly each of them in each iteration. The algorithm is free to back to each neighbourhood operator in several times consecutively or do not select them for some consecutive iterations, depends on their chance of being selected. Another differences is that VND uses an iterative improvement steps to ensure that operator can not find improving solution. However, since we have not any powerful acceleration method for COSP, using several heavy neighbourhood operator would be costly.

5.5.2 Diversification Method

Algorithm 35 describes our proposed performDiversification() method. Assuming D is a parameter to be chosen by experiments, the proposed diversification method removes

D selected customer orders from the input solution π (Line 5). We discuss selection of the D customer orders later, but $\mathcal{L}$ is the list to hold them. Given $\mathcal{L}$ the list of customer orders removed and π the resultant solution after removal, each of the customer order in $\mathcal{L}$ is then inserted back to π and also swapped with other customer orders already in π (Lines 7–17). Note that Lines 7–17 in Algorithm 35 are the same as Lines 4–14 in Algorithm 32. As such we do not describe them further, but do reiterate that a swapping based exploration is an important augmentation to the insertion based construction typically seen in similar methods.

Algorithm 35 performDiversification()

1: Let π be the input solution with n customer orders
2: Let D be the number of customer orders to be removed from π
3: Let E be the list of customer orders in the non-increasing order of extra time e_j (defined below) computed for each customer order j in π
4: Let $\mathcal{L}$ be the list of D customer orders having first half taken from the beginning of E and the second half filled in by random selection from the other customer orders, ensuring uniqueness in the list
5: $\pi \leftarrow$ remove all D customer orders in $\mathcal{L}$ from the given solution π
6: **for** $k = D$ down to 1 **do** ▷ iterate over customer orders in $\mathcal{L}$
7: $\quad \pi_* \leftarrow \varnothing$ with $\mathsf{TCT}(\varnothing) = \infty$, $k_* = 0$ ▷ to track best permutations
8: $\quad$ **for** $k' = k$ to 1 **do** ▷ a position in π to insert $\mathcal{L}[k]$
9: $\quad\quad \pi' \leftarrow$ insert customer order $\mathcal{L}[k]$ at position k' in π,
10: $\quad\quad\quad$ shifting those at position k' or later towards the end.
11: $\quad\quad$ **if** $\mathsf{TCT}(\pi') < \mathsf{TCT}(\pi_*)$ **then** ▷ if better than the best
12: $\quad\quad\quad k_* \leftarrow k'$, $\pi_* \leftarrow \pi'$ ▷ update the best
13: $\quad \pi \leftarrow \pi_*$. ▷ take the best as the current
14: $\quad$ **for** $k' = 1$ to k but $(k' \notin [k_* - 1, k_* + 1])$ **do** ▷ avoid revisitation
15: $\quad\quad \pi' \leftarrow$ swap customer orders at positions k_* and k' in π
16: $\quad\quad$ **if** $\mathsf{TCT}(\pi') < \mathsf{TCT}(\pi_*)$ **then** ▷ if better than the best
17: $\quad\quad\quad \pi_* \leftarrow \pi'$ ▷ update the best
18: $\quad \pi \leftarrow \pi_*$. ▷ take the best as the current
19: **return** π as the output solution

Our key contribution in the proposed diversification method is mainly in the selection of the D customer orders to be removed and inserted back to the given solution. Typically only random selections are used in such scenarios. Note that the diversification method is called after the intensification phase, which perhaps have already taken the search to a local minimum or a plateau. The search has therefore lost its direction. Making only random decisions at that point appears to be relying only on luck while an intelligent search approach would rather make informed decisions.

In this study, we try to identify the customer orders that are very badly affected in a given solution π. We prefer to reschedule them which can be viewed as fixing the problematic part of a solution. For this, we define a measure of the extra time e_j a customer order j stays under processing in a given solution π. A customer order j should ideally be staying under processing for a time window with length at most $\eta_{[k]} = \max_{i=1}^{m} p_{ij}$, which is the longest processing time of j on any machine. Since a machine can start processing the next customer order as soon as it completes the previous one, the time a customer order $[k]$ in π actually remains under processing is given by $\omega_{[k]} = \max_{i=1}^{m} C_{i[k]} - \min_{i=1}^{m} C_{i[k-1]}$. So $e_{[k]} = \omega_{[k]} - \eta_{[k]}$ can be used to measure the extra time customer order $[k]$ stays under processing. In the proposed diversification method, the higher $e_{[k]}$ values are used to select $D/2$ customer orders to be removed (Lines 3–4 in Algorithm 35). The other $D/2$ customer orders to be removed are however selected randomly.

Figure 5.3 shows computation of the extra times 4 customer orders in an example COSP solution with 4 machines. Notice that $e_{[1]} = 4 - 0 - 4 = 0$, $e_{[2]} = 8 - 2 - 5 = 1$, $e_{[3]} = 11 - 5 - 3 = 3$, $e_{[4]} = 13 - 7 - 4 = 2$.

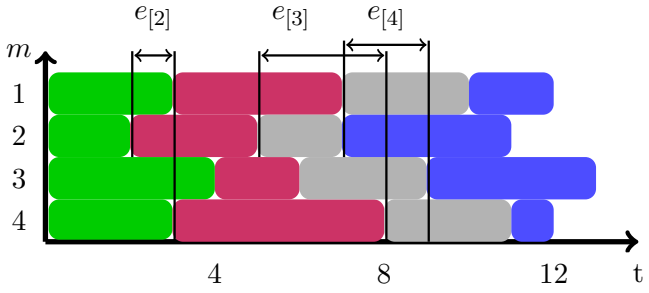


FIGURE 5.3: An example of computing extra time for each customer order

5.5.3 Acceptance Method

For the acceptance method, we consider Δ_0 that is the difference in the TCT obtained by the search procedure (lines 3-15 of Algorithm 32) of the PCE algorithm. Then, given a parameter R to be experimentally determined, we define a threshold $R \times \Delta_0$. Now, in each iteration of Algorithm 33, π is the solution before diversification and π'' is the solution after intensification. We accept π'' to be next current solution π if $\mathsf{TCT}(\pi'') - \mathsf{TCT}(\pi) \leq R \times \Delta_0$. Notice that improving solutions are always accepted in this method and worsening solutions are also accepted but only when the differences are

within the threshold limit. Notice that the threshold limit depends on each problem size and even problem instances.

5.5.4 Overall Difference with GSA

GSA algorithm is the state-of-the-art algorithm for COSP with TCT. Here, we are to specify the main difference of our algorithm with GSA. In the intensification phase, the GSA used an exhaustive swap move while our PSA uses a multi neighbourhood mechanism. It allows our algorithm to navigate through the search space, to explore more areas around the current solution, and to escape from local optima.

In the diversification phase, the GSA randomly move one of the customer orders to the last position and then try to find the best customer order for the position $n-1$. If a better solution is found, the process will continue. However, PSA uses an informed diversification by using the problem specific knowledge of the problem. In order to do that, it calculates a measure of extra time that a customer order stays under processing and use this measure to select customer orders that should be rescheduled.

5.6 Experimental Results

We evaluate our algorithms using the standard benchmark set [9]. However, we only use 720 large instances. Those instances are equally divided into two groups: Test-1 and Test-2. Each group is further divided into $3 \times 4 = 12$ sets, one for each combination of $n \in \{50, 100, 200\}$ and $m \in \{2, 5, 10, 20\}$. Each of the 12 sets thus has 30 instances. While Test-2 instances allow zero processing times $p_{ij} \in [0, 100)$, Test-1 instances does not allow that $p_{ij} \in (0, 100)$. We also use another set of 90 instances, which are generated by us and are divided into two similar instance groups. For this, we use $n \in \{50, 100, 200\}$ and $m \in \{5, 10, 20\}$, and for each combination of n and m, we generate 5 instances per group. The processing times have the same distributions as in Test-1 and Test2. These generated instances are used to calibrate the parameters and components of our algorithms while the benchmark set mentioned before is used in comparing our algorithm with the state-of-the-art algorithms.

We run each deterministic constructive algorithm only once on each instance. We run each stochastic perturbative algorithm on each instance 5 times on the same machine. The termination criterion in the proposed PSA algorithm and the GSA algorithm that we compare PSA with is the maximum iteration IterMax. This criterion was used in the evaluation of GSA.

The values used for IterMax in PSA and GSA are mentioned later whenever needed. Given that both algorithms in each iteration explore the same number of neighbouring solutions, and as we practically found, the time taken by the two algorithms are very close for a given value of IterMax. Note that it is not possible to run the GSA executable program provided by the respective author using timeout criterion.

Let TCT_a be the TCT value found by a given algorithm and TCT_r be the reference TCT value. For each run of each algorithm on each instance, we calculate the relative percentage deviation $\mathrm{RPD} = \frac{\mathsf{TCT}_\mathrm{a} - \mathsf{TCT}_\mathrm{r}}{\mathsf{TCT}_\mathrm{r}} \times 100$ of the total completion time of the schedule found. To summarise further, we calculate average RPD (ARPD) for a given algorithm over the runs of an instance, or over the instances in an instance set or in an instance group.

5.6.1 Analysis of Constructive Heuristics

Using 8 ways to construct an initial priority list $\mathcal{L}$ of customer orders, and using and not using the swap based exploration, we obtain 16 versions of the proposed PCE algorithm. These are compared with FP [9] on the 720 benchmark instances. Figure 5.4 shows the ARPDs using the best TCTs found by any of the 17 algorithms as the reference and also the Tukeys Honest Significant Difference (HSD) confidence intervals. As we see PCE versions with the swap based exploration significantly outperform the versions without. So in Table 5.1, we show in details the ARPDs for each instance set in each instance group only for the PCE versions with swap based exploration. Among all PCE versions, TPT, TWPT, and TCWPT significantly outperform the other versions and also FP. However, the difference between the three best performing versions are not significant (no overlapping intervals). We do not show the running times of these algorithms since they all have time complexity $\mathcal{O}(mn^3)$ and practically for instances with 200 customer orders, they take about 0.5 second.

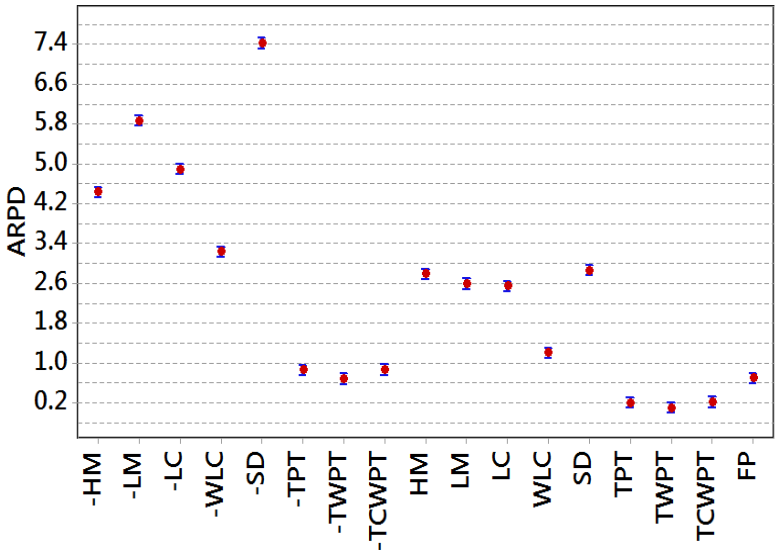


FIGURE 5.4: ARPDs and 95% HSD confidence intervals for PCE versions and FP. Names with/without minus signs denote PCE versions without/with swap-based exploration.

TABLE 5.1: Comparison of constructive heuristic algorithms PCE and FP.

TEST-1									
Instance	HM	LM	LC	WLC	SD	TPT	TWPT	TCWPT	FP
50×2	2.031	2.111	2.056	0.596	2.133	0.053	**0.027**	0.105	0.289
50×5	2.184	1.753	1.718	0.785	2.766	0.213	**0.104**	0.184	0.478
50×10	1.867	2.002	1.719	1.023	2.514	0.258	**0.144**	0.329	0.493
50×20	1.754	2.153	1.532	1.038	2.137	0.405	**0.279**	0.351	0.315
100×2	2.984	2.985	2.984	0.984	2.628	0.050	**0.015**	0.093	0.246
100×5	2.107	1.516	1.935	0.814	2.486	0.137	**0.058**	0.133	0.568
100×10	1.936	1.914	1.665	0.954	2.339	0.162	**0.071**	0.148	0.540
100×20	1.828	2.049	1.765	1.243	2.209	0.203	**0.169**	0.186	0.308
200×2	3.344	3.330	3.344	1.091	2.484	0.043	**0.005**	0.083	0.200
200×5	2.342	1.705	2.116	0.959	2.496	0.072	**0.022**	0.074	0.841
200×10	1.838	1.701	1.592	1.028	2.352	0.065	**0.040**	0.081	0.558
200×20	1.819	2.071	1.793	1.346	2.166	0.089	**0.071**	0.075	0.362
average	3.518	3.008	3.184	1.448	3.430	0.195	**0.078**	0.208	1.057
TEST-2									
Instance	HM	LM	LC	WLC	SD	TPT	TWPT	TCWPT	FP
50×2	2.883	2.939	2.863	0.976	2.902	0.104	**0.026**	0.219	0.606
50×5	2.957	2.434	2.429	1.204	3.703	0.308	**0.208**	0.415	0.924
50×10	3.279	3.083	2.614	1.459	3.480	0.548	**0.175**	0.475	0.843
50×20	3.274	3.614	2.671	1.923	2.807	0.510	**0.311**	0.407	0.897
100×2	3.567	3.554	3.565	1.097	2.777	0.063	**0.009**	0.150	0.574
100×5	3.740	2.496	3.169	1.278	4.021	0.277	**0.038**	0.242	1.466
100×10	3.377	2.906	2.918	1.472	3.684	0.226	**0.096**	0.294	1.063
100×20	2.963	3.056	2.953	1.384	2.927	**0.223**	0.237	0.228	0.695
200×2	4.380	4.368	4.383	1.510	3.132	0.046	**0.001**	0.111	0.471
200×5	3.648	2.154	3.070	1.245	3.822	0.155	**0.010**	0.198	1.839
200×10	3.368	2.683	2.808	1.482	3.534	0.170	**0.081**	0.170	1.234
200×20	3.277	3.352	3.124	1.812	3.006	0.227	**0.056**	0.167	0.740
average	3.393	3.053	3.047	1.403	3.316	0.238	**0.104**	0.256	0.946

5.6.2 Parameter Tuning for PSA

The proposed PSA has only two parameters that are to be carefully calibrated. These parameters are D the total number of customer orders to be removed in the diversification phase and R the threshold determining ratio to be used in the acceptance method. We consider $D \in \{2, 3, 4, 5, 6\}$ and $R \in \{0.01, 0.001, 0.05, 0.005\}$ after preliminary experiments. We then use the Design of Experiments method [109] to find the best value for each parameter. For this, we use the 90 generated instances mentioned before and use $\mathsf{IterMax} = 200$, and also TWPT as the initial solution. In computing the ARPDs, we use the best TCT found by any of the versions run in this calibration process.

TABLE 5.2: ANOVA results for the two parameters of PSA.

Source	Degrees of freedom	Sum of squares	Adjusted mean square	F-statistic	P-value
D	4	43.119	10.7798	1127.01	**0.000**
R	3	1.293	0.4312	45.08	**0.000**
$R \times D$	12	0.601	0.0501	5.24	**0.000**
Error	8980	85.893	0.0096		
Total	8999	130.907			

We analyse the results by means of multi-factor analysis of variance (ANOVA) method. The ANOVA results are given in Table 5.2, where those with the p-value less than 0.05 are highlighted. Three main ANOVA hypotheses i.e., normality, homoskedasticity of the different levels, and independence of the residuals are firstly checked and no significant deviations are found. From the table, we can first see that the D is the most significant factor, since it has the highest F-statistic. The p-value of both factors are less than 0.05, which means that there is a significant difference between at least two levels of each factor. To find out the best level for the factors, the 95% HSD confidence interval plots are shown in Figure 5.5.

From Figure 5.5-left, we see that $D = 2$ obtained the worst results. Overall, PSA obtained better results with larger D values. There is no statistically significant difference between using $D = 5$ and $D = 6$, where ARPDs are 0.0931 and 0.0923 respectively. However, we select $D = 6$ since it results in a lower ARPD. In terms of parameter R, we see in Figure 5.5-right that $R = 0.001$ is the worst case because it rejects most non-improving solutions. Among 0.005, 0.10, and 0.05, we select $R = 0.005$ as it gives the lowest ARPD although these three values produce results that are statistically not

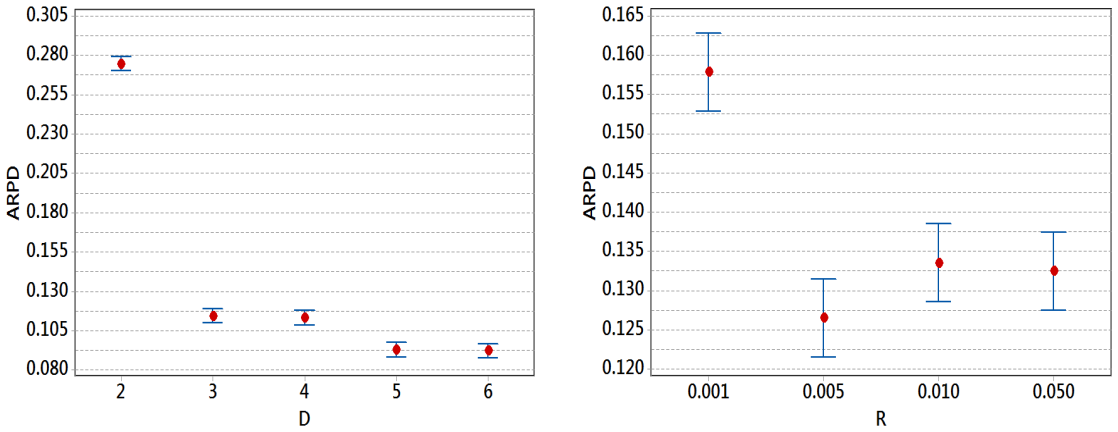


FIGURE 5.5: ARPDs and 95% HSD confidence intervals for parameters D (left) and R (right)

significantly different. Since the interaction of factors $R \times D$ is significant, we also see the interaction plot shown in Figure 5.6. From this figure, it is clear that the version with $D = 6$ and $R = 0.005$ produces the lowest ARPD.

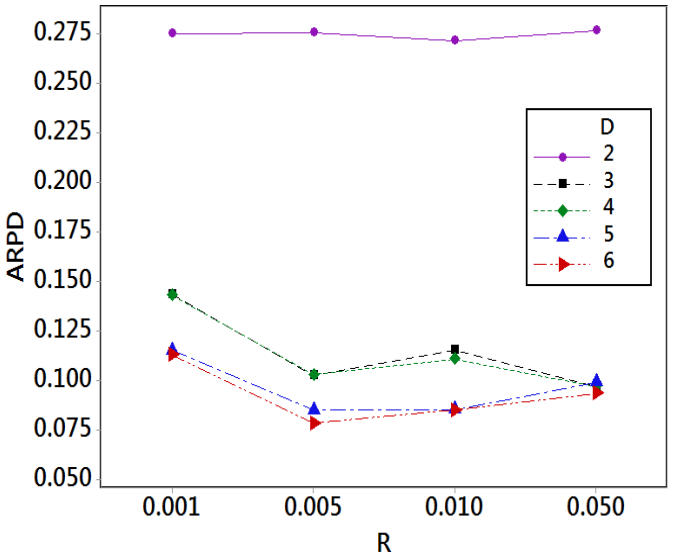


FIGURE 5.6: Interaction plot for D and R.

5.6.3 Analysis of Diversified Intensification

In the intensification phase, we use four neighbourhood operators: insert (I), swap (S), insert-pair (IP), and swap-pair (SP). We evaluate their impact on the search performance the proposed PSA. For this, we use the 90 generated instances mentioned before, IterMax = 200, and also TWPT as the initial solution. In computing the ARPDs, we use the

best TCT found by any of the versions run in this calibration process. The 95% HSD confidence interval plot is shown in Figure 5.7.

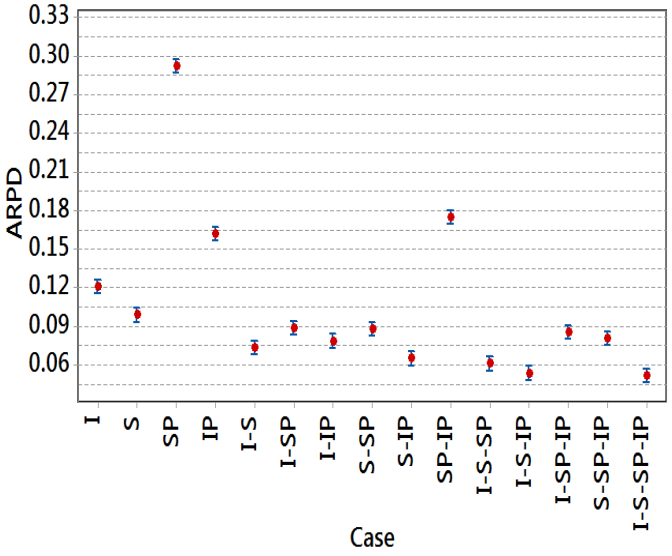


FIGURE 5.7: 95% HSD interval plot for different intensification cases.

From Figure 5.7, we see that when each operator is used separately, the swap operator obtains the best results while the swap-pair operator obtains the worst. When using two operators at a time, results are significantly better than using single operators and {insert, swap} and {swap, insert-pair} appears to be performing the best. When considering three or four operators at a time, using all four operators appears to be the best but not significantly better than using {insert, swap, insert-pair}. Therefore, we use all four operators in the final version of our PSA algorithm.

5.6.4 Analysis of Intensified Diversification

In the diversification phase, we use extra time that a customer order stays under processing to select customer orders that are to be removed from a schedule and inserted back. We compare this greedy strategy (denoted by G) with a random selection strategy (denoted by R). For this, we use the 90 generated instances mentioned before, IterMax = 200, and also TWPT as the initial solution. In computing the ARPDs, we use the best TCT found by any of two versions run in this calibration process. The 95% HSD confidence interval plot reported in Figure 5.8 confirms that our proposed strategy G significantly outperforms the typical random one.

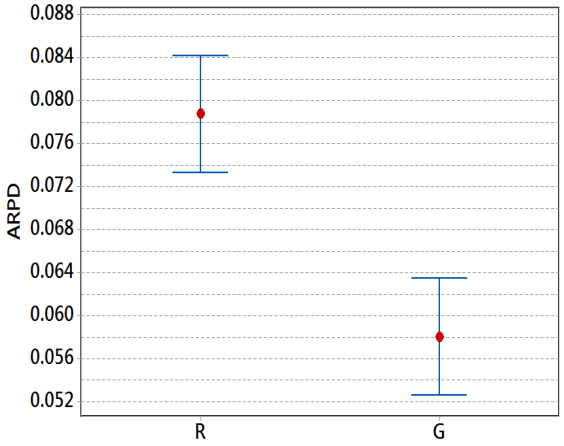


FIGURE 5.8: 95% confidence interval plot of the PSA with different diversification variants.

5.6.5 Comparison of PSA with GSA

We compare the final version of the proposed PSA with the state-of-the-art COSP algorithm GSA [9]. For initialisation in PSA, we use our proposed PCE algorithm with TWPT and SD initial sequencing strategies. TWPT and SD respectively obtain the best and the worst performance in PCE. We include SD to see how PSA performs even with the worst initial solution (2% worse than FP produced solutions). We also use FP [9] within PSA. Proposed algorithms are all implemented in programming language C, while GSA implementation is obtained from the respective authors. In this experiment, we use 720 benchmark instances and six different stopping criteria IterMax $\in \{10, 50, 100, 200, 500, 1000\}$. Note that, GSA algorithm was tested only for IterMax $\in \{10, 50, 100\}$. For ARPD calculation, the best solutions found by any algorithm are used as references. The results are shown in Table 5.3 and Table 5.4 for Test-1 and Test-2 instance groups respectively.

From those two tables, we see that all PSA variants obtained ARPDs lower than GSA. Interestingly, PSA obtains similar performance regardless of which initial solution is used. This shows PSA's robustness. Further, ARPDs obtained by PSA variants after 100 iterations are better than those obtained by GSA and after 1000 iterations, become four times better. However, ARPDs improve for all algorithms with the increase of the numbers of iterations. Similar observations can also be made from the 95% HSD confident intervals for Test-1 and Test-2 in Figure 5.9 and 5.10 respectively.

TABLE 5.3: ARPDs for the search algorithms on Test-1

	GSA						PSA-FP					
Ins.	10	50	100	200	500	1000	10	50	100	200	500	1000
50×2	0.076	0.042	0.035	0.026	0.023	0.019	0.024	0.007	0.005	0.003	0.001	0.001
50×5	0.306	0.192	0.147	0.132	0.108	0.097	0.175	0.079	0.057	0.040	0.024	**0.016**
50×10	0.471	0.264	0.232	0.202	0.163	0.141	0.372	0.200	0.156	0.114	0.079	**0.050**
50×20	0.536	0.316	0.265	0.216	0.186	0.159	0.440	0.240	0.188	0.146	0.105	**0.081**
100×2	0.074	0.053	0.048	0.043	0.037	0.034	0.031	0.010	0.006	0.004	0.002	**0.001**
100×5	0.286	0.198	0.190	0.170	0.150	0.141	0.180	0.108	0.082	0.060	0.039	0.025
100×10	0.406	0.304	0.279	0.241	0.215	0.202	0.311	0.182	0.136	0.103	0.065	**0.042**
100×20	0.497	0.316	0.277	0.245	0.212	0.190	0.436	0.265	0.208	0.155	0.101	**0.074**
200×2	0.044	0.037	0.035	0.033	0.030	0.029	0.022	0.008	0.006	0.004	**0.002**	**0.002**
200×5	0.251	0.173	0.159	0.146	0.138	0.133	0.136	0.081	0.059	0.043	0.026	**0.015**
200×10	0.422	0.247	0.221	0.204	0.190	0.182	0.232	0.142	0.109	0.076	0.046	**0.024**
200×20	0.583	0.283	0.240	0.213	0.200	0.189	0.356	0.223	0.177	0.127	0.075	0.040
Avg	0.329	0.202	0.177	0.156	0.138	0.126	0.226	0.129	0.099	0.073	0.047	**0.031**
	PSA-SD						PSA-TWPT					
Ins.	10	50	100	200	500	1000	10	50	100	200	500	1000
50×2	0.024	0.007	0.004	0.003	0.001	**0.000**	0.027	0.007	0.004	0.002	0.001	0.001
50×5	0.186	0.087	0.066	0.049	0.029	0.018	0.179	0.086	0.064	0.042	0.026	**0.016**
50×10	0.376	0.220	0.168	0.126	0.086	0.064	0.354	0.188	0.149	0.117	0.075	0.054
50×20	0.446	0.254	0.200	0.149	0.103	0.084	0.442	0.258	0.204	0.162	0.117	0.090
100×2	0.030	0.010	0.006	0.004	0.002	**0.001**	0.031	0.010	0.007	0.004	0.002	**0.001**
100×5	0.188	0.108	0.080	0.058	0.036	**0.023**	0.183	0.104	0.080	0.060	0.036	**0.023**
100×10	0.307	0.177	0.141	0.104	0.066	0.044	0.308	0.182	0.134	0.103	0.063	0.043
100×20	0.427	0.259	0.199	0.153	0.105	**0.074**	0.435	0.272	0.213	0.169	0.110	0.077
200×2	0.022	0.008	0.006	0.004	**0.002**	**0.002**	0.022	0.008	0.006	0.004	0.003	**0.002**
200×5	0.138	0.079	0.060	0.043	0.026	**0.015**	0.138	0.082	0.061	0.045	0.026	**0.015**
200×10	0.229	0.143	0.111	0.083	0.049	0.026	0.234	0.144	0.109	0.080	0.047	0.026
200×20	0.352	0.223	0.163	0.122	0.069	**0.034**	0.355	0.214	0.166	0.121	0.070	0.038
Avg	0.227	0.131	0.100	0.075	0.048	0.032	0.226	0.130	0.100	0.076	0.048	0.032

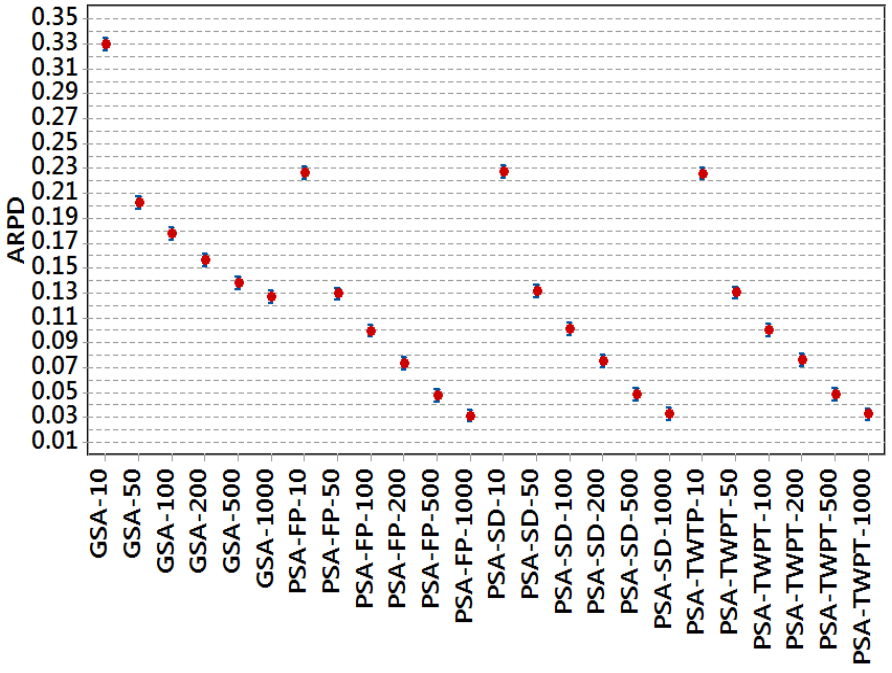


FIGURE 5.9: 95% confidence interval plot of the algorithms compared on Test-1.

TABLE 5.4: ARPDs for the search algorithms on Test-2

	GSA						PSA-FP					
Ins.	10	50	100	200	500	1000	10	50	100	200	500	1000
50×2	0.106	0.052	0.037	0.029	0.019	0.014	0.033	0.009	0.004	0.002	0.001	**0.000**
50×5	0.439	0.242	0.201	0.160	0.128	0.103	0.259	0.123	0.085	0.063	0.039	0.028
50×10	0.623	0.322	0.257	0.211	0.167	0.143	0.369	0.173	0.131	0.097	0.060	0.040
50×20	0.689	0.297	0.235	0.208	0.149	0.135	0.427	0.226	0.170	0.127	0.082	**0.047**
100×2	0.096	0.056	0.051	0.040	0.035	0.031	0.028	0.010	0.007	0.004	0.002	**0.001**
100×5	0.408	0.294	0.286	0.251	0.219	0.209	0.258	0.142	0.109	0.079	0.047	0.029
100×10	0.557	0.382	0.352	0.315	0.270	0.255	0.437	0.254	0.185	0.135	0.083	**0.052**
100×20	0.588	0.382	0.354	0.325	0.272	0.258	0.589	0.349	0.264	0.206	0.126	0.085
200×2	0.051	0.044	0.040	0.036	0.032	0.030	0.020	0.007	0.004	0.003	0.002	**0.001**
200×5	0.289	0.228	0.218	0.209	0.190	0.178	0.208	0.116	0.085	0.061	0.036	0.022
200×10	0.475	0.344	0.321	0.297	0.276	0.261	0.402	0.235	0.183	0.133	0.077	0.044
200×20	0.554	0.365	0.312	0.275	0.247	0.234	0.548	0.363	0.288	0.219	0.121	0.071
Avg	0.406	0.251	0.222	0.196	0.167	0.154	0.298	0.167	0.126	0.094	0.056	**0.035**
	PSA-SD						PSA-TWPT					
Ins.	10	50	100	200	500	1000	10	50	100	200	500	1000
50×2	0.031	0.008	0.004	0.002	0.001	**0.000**	0.033	0.008	0.005	0.002	0.001	**0.000**
50×5	0.263	0.122	0.094	0.061	0.037	0.028	0.256	0.129	0.097	0.063	0.037	**0.025**
50×10	0.391	0.193	0.137	0.093	0.054	**0.036**	0.397	0.200	0.141	0.095	0.060	0.041
50×20	0.413	0.224	0.162	0.113	0.078	0.060	0.424	0.200	0.148	0.107	0.061	**0.047**
100×2	0.027	0.010	0.006	0.003	0.002	**0.001**	0.029	0.009	0.005	0.003	0.002	**0.001**
100×5	0.260	0.142	0.106	0.077	0.046	**0.027**	0.257	0.143	0.103	0.078	0.047	0.030
100×10	0.434	0.243	0.187	0.136	0.081	0.054	0.433	0.251	0.184	0.138	0.094	0.059
100×20	0.595	0.353	0.272	0.207	0.131	0.089	0.571	0.341	0.268	0.201	0.125	**0.084**
200×2	0.021	0.008	0.005	0.003	0.002	**0.001**	0.020	0.007	0.005	0.003	0.002	**0.001**
200×5	0.211	0.112	0.081	0.061	0.037	0.020	0.215	0.116	0.083	0.060	0.035	**0.019**
200×10	0.406	0.248	0.193	0.139	0.076	0.042	0.407	0.247	0.185	0.132	0.074	**0.041**
200×20	0.559	0.371	0.290	0.211	0.121	**0.066**	0.551	0.366	0.289	0.211	0.124	**0.066**
Avg	0.301	0.170	0.128	0.092	0.055	**0.035**	0.299	0.168	0.126	0.091	0.055	**0.035**

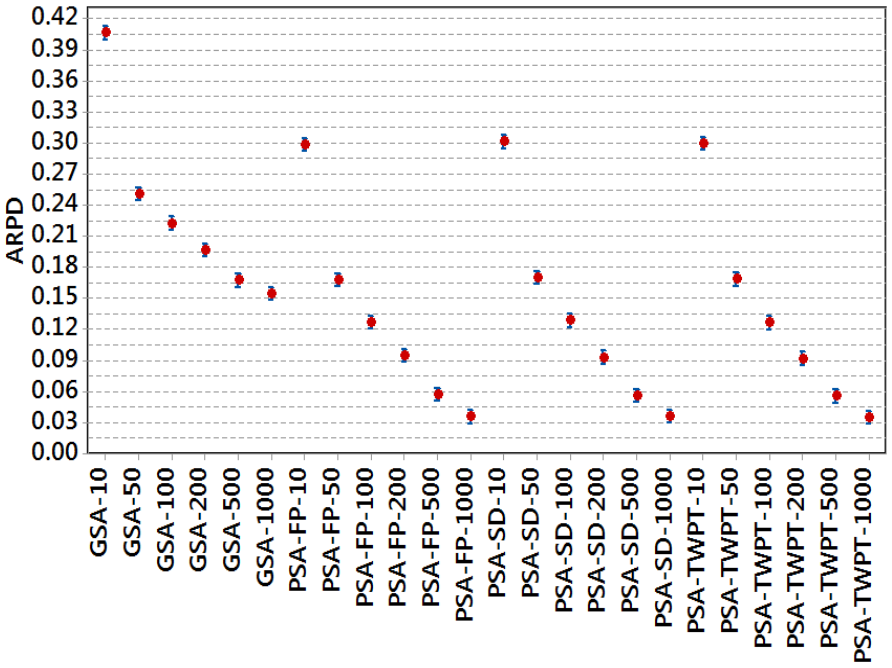


FIGURE 5.10: 95% confidence interval plot of the algorithms compared on Test-2.

To observe the effectiveness of the proposed PSA further, we also show the convergence of GSA and PSA algorithms in four selected instances–Test-1-347, Test-1-463, Test-2-312, and Test-2-469–in Figure 5.11. Since GSA used FP as the initial solution, we also use it as our starting point although our results showed that our PSA does not depend much on the quality of initial solution. From these graphs, we see the convergence speed of GSA is faster than that of PSA. Convergence graphs for other instances are similar and hence not shown.

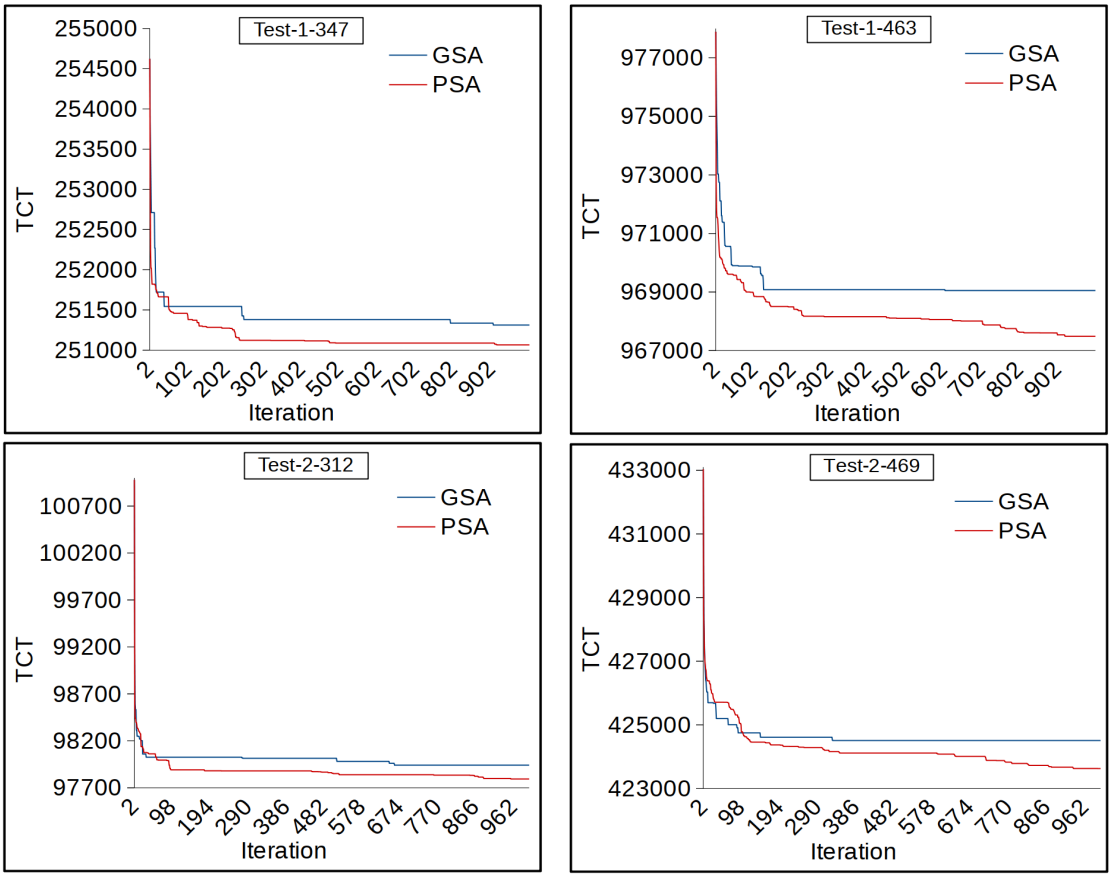


FIGURE 5.11: Convergence curves of GSA and PSA algorithms on four selected instances.

Since the search process of the PSA (using FP heuristic as initial point) and GSA in each iteration is not the same, we also report the CPU times of those two algorithms in Table 5.5. Moreover, we report the relative deviation $RD = 100 \times (GSA - PSA)/PSA$ of CPU time. As can be seen, the PSA algorithm is much faster than GSA, 11% in overall average, yet finds better solutions as shown in Tables 5.3 and 5.4.

Finally, the best found solutions for the instances of [9] obtained by any of the comparing algorithms are compared. These best found solutions are already used in this research

TABLE 5.5: CPU times needed for GSA and PSA algorithms.

Instance	GSA	PSA	RD
50×2	1.99	1.74	14.44
50×5	3.59	3.51	2.07
50×10	6.48	6.35	2.13
50×20	12.16	11.65	4.33
100×2	11.80	11.34	4.05
100×5	26.65	23.94	11.30
100×10	48.74	43.88	11.07
100×20	92.05	82.91	11.01
200×2	89.79	80.70	11.27
200×5	208.20	175.79	18.44
200×10	386.46	321.24	20.30
200×20	765.25	609.57	25.54
average	137.76	114.39	11.33

in the ARPD calculation. Table 5.6 summarises the key results. As can be seen, almost all of the best found solutions are obtained by the proposed PSA algorithms.

TABLE 5.6: The number of best found solutions obtained by the four algorithms

Instances	Algorithms			
Total 720	GSA	PSA-FP	PSA-SD	PSA
Test-1	3	**177**	173	153
Test-2	8	201	197	**202**
Total	11	**378**	370	355

5.7 Conclusions

In this chapter, we study the customer order scheduling problem (COSP) with total completion times, which is known to be NP-hard. The COSP has realistic applications that include the paper and the pharmaceutical industries. We propose constructive and perturbative search algorithms for COSP embedding problem specific heuristics within. We propose effective dispatching rules to be used during solution construction. We propose greedy ways to select customer orders that should be rescheduled during the diversification phase of a perturbative search algorithm. We also diversify the intensification phase of the perturbative search algorithm by using multiple neighbourhood operators. Our experimental results show that the proposed algorithms outperform existing state-of-the-art COSP algorithms. Moreover, our proposed algorithms have found the best solutions in almost all benchmark problem instances used in the experiments.

Chapter 6

Conclusions

This chapter summarises the contributions of the research presented in this thesis. This chapter also covers the objectives and aims set before the research and hypotheses verified after the experiments. We also outline a few possible potential future directions of this research and conclude the thesis.

6.1 Constraint Directed Scheduling

This thesis is to advance scheduling problems using constraint based approaches. Scheduling is a decision making process dealing with allocation of resources to tasks over given time periods in order to optimise one or more objective functions. Scheduling problems commonly involve finding values to a set of decision variables. These variable-value assignments are often restricted by a set of constraints. Scheduling problems are in general mostly NP-hard. Thus, incomplete algorithms such as local search algorithms and evolutionary algorithms have been used to solve scheduling problems. However, within practical time limits, most of existing incomplete algorithms still either find low quality solutions or struggle with large problems. In this thesis, we try to show that one key reason behind this is the typical way of using generic heuristics or metaheuristics that usually lack problem specific structural knowledge. In this work, we aim to advance scheduling problems' search by better exploiting the problem specific structural knowledge. We use constraints and objective functions to obtain such problem specific knowledge and we exploit such knowledge both in constructive search methods and local

search methods. Our motivation comes from the constraint optimisation paradigm in artificial intelligence, where instead of random decisions, constraint-guided more informed optimisation decisions are of particular interest.

6.2 Flowshop Scheduling Problems with Blocking

Permutation flowshop scheduling problem (PFSP) is a hard combinatorial optimisation problem. We study a generalised and more realistic variant of that problem called mixed blocking permutation flowshop scheduling problem (MBPFSP). MBPFSP has several real applications such as cider industry. We make the following contributions to this problem:

- A new accelerated method is proposed for insertion-based neighbourhoods of MBPFSP. The results demonstrate that using this speedup method speeds up the algorithms more than 95%. Also, we study the cider industry and show that the cider industry is a real example of MBPFSP.

- One population-based search algorithm called scatter search (SS), and one local search based algorithm is proposed in order to solve this problem. In the proposed algorithms, we advance MBPFSP search by better exploiting the problem specific structural knowledge. We use constraints and objective functions to obtain such problem specific knowledge and we exploit such knowledge both in a constructive search method and in the search algorithms. In MBPFSP, a machine could be blocked with the currently finished job until the subsequent machine is available to process the same job. These blocking constraints affect the makespan. So MBPFSP search should naturally take explicit steps to take the blocking constraints into account.

- After studying the MBPFSP, inspired by the cider industry, we propose a new PFSP variant that generalises over simultaneous use of several types of blocking constraint and various settings of sequence-dependent setup time. We also present a computational model for makespan minimisation of the new variant and show that solving this variant remains NP-hard. We then present an acceleration method to compute makespan efficiently and thus evaluate the neighbourhoods

generated by insertion operators. We develop a new constructive heuristic taking both blocking constraints and setup times into account. We also develop a new local search algorithm that uses a constraint guided intensification method and a random-path guided diversification method. Our comprehensive experimental results on a set of benchmark instances demonstrate that our proposed algorithms significantly outperform several state-of-the-art adapted algorithms.

6.3 Aircraft Scheduling Problems

The Aircraft scheduling problem (ASP) has attracted much attention in recent years. Because flight delays are a very important growing challenge that needs to be tackled with proper emphasis. For example, 58.93 million passengers travelled in year 2016 in Australia, which was 2.5% more compared to that in year 2015. According to Federal Aviation Administration (FAA), the total cost of all US air transportation delays in 2007 was estimated to be $31.2 billion. Because of having different characteristics of multiple-runway and single-runway cases, we perform a study on each of them. In this thesis, using a constraint-based approach, we make the following contributions on the ASP:

- ASP has made significant progress in recent years. However, within practical time limits, existing incomplete algorithms still either find low quality solutions or struggle with large problems. In this work, we advance ASP search by better exploiting the problem specific structural knowledge. We use the constraints and the objective functions to obtain such problem specific knowledge and we exploit such knowledge both in constructive search method and local search algorithms.

- We have proposed two new constructive heuristics. The two proposed constructive heuristics are evaluated against the best known constructive heuristic AATCSR. The results shows that the proposed heuristics obtained ARPDs 8.9% and 6.7% while AATCSR obtained 59.9%. AATCSR constructs the solution simply by appending each next aircraft based on a precomputed index without performing any search. In contrast, the two proposed heuristics perform search on the already constructed partial solutions using the objective function to find the best positions for the next aircraft. The guided search in the two proposed heuristics clearly makes

the difference between their performance and that of AATCSR, which does not have any search.

- The proposed local search algorithm embeds constraint-guided strategies in its exploration and perturbation phases. Taking the time constraint into account, an adjacent swap operator is used in the exploration phase instead of using a typical intra-swap operator. The adjacent swap exchanges two aircraft that in successive positions in a runway while the intra-swap exchanges any two aircraft in a runway. Taking the objective function into account, runways that have the most objective value are selected for the operators to be applied on. The runway selection allows the search to fix the most problematic part of the current solution. In fact, instead of the typical random runway selection, we insert the greediness into it by selecting one of the runway greedily, the one with highest objective function created. This allows us to fix the most problematic part of the current solution.

- In terms of single-runway ASP, to solve this problem, we also propose a constraint-guided local search algorithm that advances ASP search by injecting the specific knowledge of the problem into its different phases. In the intensification phase, we propose a greedy approach that gives more priorities to aircraft that are more problematic and create more delays. In the diversification phase, we employ a bounded-diversification technique that controls the new position of each selected aircraft and does not allow them to move very far away from their current positions.

6.4 Customer Order Scheduling Problems

In this thesis, we exploit the structure information of customer order scheduling problem (COSP) in constructive heuristics and also propose a constraint-based local search. The details of our contributions are outlined as follows:

- In this research, we propose a constructive heuristic. we first study the existing constructive algorithms and observe that they place an unscheduled customer order only at the end of the already scheduled sequence. However, in our view, the end of the partial schedule actually might not be the best position for that unscheduled customer order to be placed, some other earlier positions might be

better suited. In our permutation construction and exploration algorithm (PCE), we consider all possible positions in the partial sequence for an unscheduled customer order to be placed in with a view to finding a better position. Moreover, we consider the unscheduled jobs one by one as they appear in a predefined list that is constructed using the processing times of the customer orders. We use processing times in various ways to obtain initial dispatching sequences that are later used in prioritising customer orders during search.

- We also propose a perturbative search algorithm intensifies its diversification phase and diversifies its intensification phase. We define a measure of extra time a customer order stays under processing and use this measure in the diversification phase to select customer orders that should be rescheduled. By this, we try to make informed decisions when the search has lost its direction in a local minimum or in a plateau. In fact, we try to identify the customer orders that are very badly affected in a given solution. We prefer to reschedule them which can be viewed as fixing the problematic part of a solution. In the intensification phase, we use multiple neighbourhood operators randomly so the search can get out of the local minima for one neighbourhood operator using another neighbourhood operator as an escape route.

6.5 Future Directions

In this thesis every chapter deals with a different kind of scheduling problem and a different kind of technique. Thus, this research leads to a number of possible future directions to solve these problems more efficiently.

- In this thesis, algorithms have been developed to solve single objective MBPFSPs. Often the problems require considering more than one objective, like minimising makespan and also optimising total earliness and tardiness costs. The models and constraint-based search algorithms can be developed to consider the multi-objective MBPFSP, and its variants such as distributed MBPFSPs. In addition, other real examples can be studied and the performance of the proposed techniques can be tried on the real-world instances.

- In this thesis, only local search approaches are used to handle the aircraft scheduling problem. It will be interesting to see the performance of complete approaches. In addition, the proposed techniques can be extended to study other ASPs variants such as optimisation of flight trajectories, considering taxi ways into the model. Also, other objective functions such as minimising the air/noise emissions can be studied. In addition, the real airports can be studied and a more realistic and complex models can be developed taking real constraints such as gate limitations, crew scheduling, and emergency landing flights into account.

- For COSP, other realistic variants such as COSP with learning effects, unrelated parallel machines, multi-objective functions can be studied and the proposed constraint-based search algorithms can be extended for those variants. In addition, population-based algorithms can also be developed for this problem. Also, some exact algorithms can be developed for this NP-hard problems to at least obtain some upper bounds. Moreover, the performance of the proposed algorithm can be studied on the benchmarks taken from real industries.

In the future, we plan to continue our investigation to improve the effectiveness and efficiency of the search algorithms by exploiting structure information for other scheduling problems and also combinatorial problems.

Bibliography

[1] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[2] Farbod Farhadi, Ahmed Ghoniem, and Mohammed Al-Salem. Runway capacity management–an empirical study with application to doha international airport. *Transportation Research Part E: Logistics and Transportation Review*, 68:53–63, 2014.

[3] Eva Vallada, Rubén Ruiz, and Jose M Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677, 2015.

[4] Shih-Wei Lin and Kuo-Ching Ying. Optimization of makespan for no-wait flowshop scheduling problems using efficient matheuristics. *Omega*, 64:115–125, 2016.

[5] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[6] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[7] Vahid Riahi, Mostafa Khorramizadeh, MA Hakim Newton, and Abdul Sattar. Scatter search for mixed blocking flowshop scheduling. *Expert Systems with Applications*, 79:20–32, 2017.

[8] Sergio Martinez, Stéphane Dauzère-Pérès, Christelle Gueret, Yazid Mati, and Nathalie Sauer. Complexity of flowshop scheduling problems with a new blocking constraint. *European Journal of Operational Research*, 169(3):855–864, 2006.

[9] Jose M Framinan and Paz Perez-Gonzalez. New approximate algorithms for the customer order scheduling problem with total completion time objective. *Computers & Operations Research*, 78:181–192, 2017.

[10] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[11] Ruhul Sarker, Masoud Mohammadian, and Xin Yao. *Evolutionary optimization*, volume 48. Springer Science & Business Media, 2002.

[12] Wajdi Trabelsi, Christophe Sauvey, and Nathalie Sauer. Heuristics and metaheuristics for mixed blocking constraints flowshop scheduling problems. *Computers & Operations Research*, 39(11):2520–2527, 2012.

[13] Nasser R Sabar and Graham Kendall. An iterated local search with multiple perturbation operators and time varying perturbation strength for the aircraft landing problem. *Omega*, 56:88–98, 2015.

[14] Cheng Wang, Shiji Song, Jatinder ND Gupta, and Cheng Wu. A three-phase algorithm for flowshop scheduling with blocking to minimize makespan. *Computers & Operations Research*, 39(11):2880–2887, 2012.

[15] Gulsah Hancerliogullari, Ghaith Rabadi, Ameer H Al-Salem, and Mohamed Kharbeche. Greedy algorithms and metaheuristics for a multiple runway combined arrival-departure aircraft sequencing problem. *Journal of Air Transport Management*, 32:39–48, 2013.

[16] Mostafa Khorramizadeh and Vahid Riahi. A bee colony optimization approach for mixed blocking constraints flow shop scheduling problems. *Mathematical Problems in Engineering*, 2015, 2015.

[17] Zhongshi Shao, Dechang Pi, and Weishi Shao. A novel discrete water wave optimization algorithm for blocking flow-shop scheduling problem with sequence-dependent setup times. *Swarm and Evolutionary Computation*, 40:53–75, 2018.

[18] Wajdi Trabelsi, Christophe Sauvey, and Nathalie Sauer. Heuristic methods for problems with blocking constraints solving jobshop scheduling. In *Proceedings of the eighth International Conference on Modelling and Simulation, Hammamet, MOSIM*, 2010.

[19] bitre. *Bureau of Infrastructure, Transport and Regional Economics*, 2017. `https://bitre.gov.au/`.

[20] M Ball, C Barnhart, M Dresner, M Hansen, K Neels, A Odoni, E Peterson, L Sherry, A Trani, and B Zou. Total delay impact study: Institute of transportation studies. *University of California, Berkeley*, 2010.

[21] Amir Salehipour, Mohammad Modarres, and Leila Moslemi Naeni. An efficient hybrid meta-heuristic for aircraft landing problem. *Computers & Operations Research*, 40(1):207–213, 2013.

[22] H Pinol and John E Beasley. Scatter search and bionomic algorithms for the aircraft landing problem. *European Journal of Operational Research*, 171(2):439–462, 2006.

[23] BS Girish. An efficient hybrid particle swarm optimization algorithm in a rolling horizon framework for the aircraft landing problem. *Applied Soft Computing*, 44: 200–221, 2016.

[24] Reza Ahmadi, Uttarayan Bagchi, and Thomas A Roemer. Coordinated scheduling of customer orders for quick response. *Naval Research Logistics (NRL)*, 52(6):493–512, 2005.

[25] Jaehwan Yang. *Scheduling with batch objectives*. PhD thesis, The Ohio State University, 1998.

[26] J Leung, H Li, and M Pinedo. Multidisciplinary scheduling: Theory and applications. *Chapter Order Scheduling Models: an overview*, 2005.

[27] Joseph Y-T Leung, Haibing Li, and Michael Pinedo. Order scheduling in an environment with dedicated resources in parallel. *Journal of Scheduling*, 8(5): 355–386, 2005.

[28] Xiaoyun Xu, Ying Ma, Zihuan Zhou, and Yaping Zhao. Customer order scheduling on unrelated parallel machines to minimize total completion time. *IEEE Transactions on Automation Science and Engineering*, 12(1):244–257, 2015.

[29] Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. Constraint guided accelerated search for mixed blocking permutation flowshop scheduling. *Computers & Operations Research*, 102:102–120, 2019.

[30] Vahid Riahi, MA Hakim Newton, Kaile Su, and Abdul Sattar. Local search for flowshops with setup times and blocking constraints. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 199–207, 2018.

[31] MA Hakim Newton, Vahid Riahi, Kaile Su, and Abdul Sattar. Scheduling blocking flowshops with setup times via constraint guided and accelerated local search. *Computers & Operations Research*, 109:64–76, 2019.

[32] Vahid Riahi, MA Hakim Newton, MMA Polash, Kaile Su, and Abdul Sattar. Constraint guided search for aircraft sequencing. *Expert Systems with Applications*, 118:440–458, 2019.

[33] Vahid Riahi, MA Hakim Newton, and Abdul Sattar. Constraint-guided local search for single mixed-operation runway. In *Australasian Joint Conference on Artificial Intelligence*, pages 329–341. Springer, 2018.

[34] Vahid Riahi, MMA Polash, MA Hakim Newton, and Abdul Sattar. Mixed neighbourhood local search for customer order scheduling problem. In *Pacific Rim International Conference on Artificial Intelligence*, pages 296–309. Springer, 2018.

[35] Vahid Riahi, MA Hakim Newton, MMA Polash, and Abdul Sattar. Customer order scheduling by scattered wolf packs. In *7th International Conference on Metaheuristics and Nature Inspired Computing, Morroco*, 2018.

[36] Vahid Riahi, MA Hakim Newton, MMA Polash, and Abdul Sattar. Tailoring customer order scheduling search algorithms. *Computers & Operations Research*, 108:155–165, 2019.

[37] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[38] Quan-Ke Pan and Rubén Ruiz. Local search methods for the flowshop scheduling problem with flowtime minimization. *European Journal of Operational Research*, 222(1):31–43, 2012.

[39] M Fatih Tasgetiren, Damla Kizilay, Quan-Ke Pan, and Ponnuthurai N Suganthan. Iterated greedy algorithms for the blocking flowshop scheduling problem with makespan criterion. *Computers & Operations Research*, 77:111–126, 2017.

[40] Rubén Ruiz and Thomas Stützle. An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. *European Journal of Operational Research*, 187(3):1143–1159, 2008.

[41] T.F. Gonzalez. *Handbook of approximation algorithms and metaheuristics*. chapman and hall/crc computer and information science series, Chapman and Hall/CRC, 2007.

[42] Edmund K Burke and Yuri Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.

[43] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[44] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European journal of operational research*, 130(3):449–467, 2001.

[45] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[46] Thomas Stützle and Rubén Ruiz. Iterated greedy. *Handbook of Heuristics*, pages 1–31, 2017.

[47] Elena Marchiori and Adri Steenbeek. An evolutionary algorithm for large scale set covering problems with application to airline crew scheduling. In *Workshops on Real-World Applications of Evolutionary Computation*, pages 370–384. Springer, 2000.

[48] A. Cesta, A. Oddi, and S. F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. *AAAI/IAAI*, pages 742–747, 2000.

[49] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[50] AJ Richmond and JE Beasley. An iterative construction heuristic for the ore selection problem. *Journal of Heuristics*, 10(2):153–167, 2004.

[51] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[52] Lawrence Davis. Handbook of genetic algorithms. 1991.

[53] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[54] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in engineering software*, 69:46–61, 2014.

[55] Daniel Câmara. *Bio-inspired networking*. Elsevier, 2015.

[56] Xin-She Yang. *Nature-inspired optimization algorithms*. Elsevier, 2014.

[57] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[58] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[59] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999.

[60] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[61] Fred Glover. Genetic algorithms and scatter search: unsuspected potentials. *Statistics and Computing*, 4(2):131–140, 1994.

[62] Fred Glover. Tabu search and adaptive memory programming–advances, applications and challenges. In *Interfaces in computer science and operations research*, pages 1–75. Springer, 1997.

[63] Rafael Martí, Manuel Laguna, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[64] Fred Glover, Manuel Laguna, and Rafael Martí. Scatter search and path relinking: Foundations and advanced designs. In *New optimization techniques in engineering*, pages 87–99. Springer, 2004.

[65] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[66] JM Framinan and R Leisten. An efficient constructive heuristic for flowtime minimisation in permutation flow shops. *Omega*, 31(4):311–317, 2003.

[67] Rubén Ruiz and Concepción Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165 (2):479–494, 2005.

[68] Chandrasekharan Rajendran. Heuristic algorithm for scheduling in a flowshop to minimize total flowtime. *International Journal of Production Economics*, 29(1): 65–73, 1993.

[69] Ling Wang, Quan-Ke Pan, and M Fatih Tasgetiren. A hybrid harmony search algorithm for the blocking permutation flow shop scheduling problem. *Computers & Industrial Engineering*, 61(1):76–83, 2011.

[70] Shahriar Farahmand Rad, Rubén Ruiz, and Naser Boroojerdian. New high performing heuristics for minimizing makespan in permutation flowshops. *Omega*, 37 (2):331–345, 2009.

[71] Eric Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65–74, 1990.

[72] Michael L Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[73] Annie Proulx and Lew Nichols. *Sweet & hard cider*. Garden Way Pub., 1980.

[74] Qun Lin, Liang Gao, Xinyu Li, and Chunjiang Zhang. A hybrid backtracking search algorithm for permutation flow-shop scheduling problem. *Computers & Industrial Engineering*, 85:437–446, 2015.

[75] Lovner. Optimal planning of parts machining on a number of machines. *Automation and Remote Control*, 12:1972–1978, 1969.

[76] Imma Ribas, Ramon Companys, and Xavier Tort-Martorell. An iterated greedy algorithm for the flowshop scheduling problem with blocking. *Omega*, 39(3):293–301, 2011.

[77] Yu-Yan Han, Dunwei Gong, and Xiaoyan Sun. A discrete artificial bee colony algorithm incorporating differential evolution for the flow-shop scheduling problem with blocking. *Engineering Optimization*, 47(7):927–946, 2015.

[78] S Dauzère-Pérès, C Pavageau, and N Sauer. Modélisation et résolution par plne d'un problème réel d'ordonnancement avec contraintes de blocage. *3ème congrès de la société Française de Recherche Opérationnelle et d'Aide à la Dècision, Nantes*, 2000.

[79] Sergio Martinez de La Piedra. *Ordonnancement de systèmes de production avec contraintes de blocage*. PhD thesis, Nantes, 2005.

[80] Kun Yuan and Sauer. Application of EM algorithm to flowshop scheduling problems with a special blocking. In *Proceedings of the ISEM*, 2007.

[81] Kun Yuan, Nathalie Sauer, and Christophe Sauvey. Application of em algorithm to hybrid flow shop scheduling problems with a special blocking. In *Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation*, 2009.

[82] Pawel Jan Kalczynski and Jerzy Kamburowski. On the NEH heuristic for minimizing the makespan in permutation flow shops. *Omega*, 35(1):53–60, 2007.

[83] Victor Fernandez-Viagas and Jose M Framinan. NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness. *Computers & Operations Research*, 60:27–36, 2015.

[84] Quan-Ke Pan and Ling Wang. Effective heuristics for the blocking flowshop scheduling problem with makespan minimization. *Omega*, 40(2):218–229, 2012.

[85] Xingye Dong, Houkuan Huang, and Ping Chen. An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion. *Computers & Operations Research*, 36(5):1664–1669, 2009.

[86] M Fatih Tasgetiren, Quan-Ke Pan, Ponnuthurai N Suganthan, and Angela HL Chen. A discrete artificial bee colony algorithm for the total flowtime minimization in permutation flow shops. *Information Sciences*, 181(16):3459–3475, 2011.

[87] Ling Wang, Quan-Ke Pan, and M Fatih Tasgetiren. Minimizing the total flow time in a flow shop with blocking by using hybrid harmony search algorithms. *Expert Systems with Applications*, 37(12):7929–7936, 2010.

[88] Christophe Sauvey and Nathalie Sauer. A genetic algorithm with genes-association recognition for flowshop scheduling problems. *J. of Intelligent Manufacturing*, 23 (4):1167–1177, 2012.

[89] Colin R Reeves and Takeshi Yamada. Genetic algorithms, path relinking, and the flowshop sequencing problem. *Evolutionary Computation*, 6(1):45–60, 1998.

[90] Eva Vallada and Rubén Ruiz. Genetic algorithms with path relinking for the minimum tardiness permutation flowshop problem. *Omega*, 38(1):57–67, 2010.

[91] Miguel A González, Angelo Oddi, Riccardo Rasconi, and Ramiro Varela. Scatter search with path relinking for the job shop with time lags and setup times. *Computers & Operations Research*, 60:37–54, 2015.

[92] François Berthaut, Robert Pellerin, Adnène Hajji, and Nathalie Perrier. *A Path Relinking-based Scatter Search for the Resource-constrained Project Scheduling Problem*. Octobre, 2014.

[93] Mohammad Ranjbar and F Kianfar. A hybrid scatter search for the RCPSP. *Scientia Iranica. Transaction E, Industrial Engineering*, 16(1):11, 2009.

[94] Mohammad Mahdi Nasiri and Farhad Kianfar. A hybrid scatter search for the partial job shop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 52(9-12):1031–1038, 2011.

[95] Qingxin Guo and Lixin Tang. An improved scatter search algorithm for the single machine total weighted tardiness scheduling problem with sequence-dependent setup times. *Applied Soft Computing*, 29:184–195, 2015.

[96] Reza Tavakkoli-Moghaddam, Nikbakhsh Javadian, A Khorrami, and Yousef Gholipour-Kanani. Design of a scatter search method for a novel multi-criteria group scheduling problem in a cellular manufacturing system. *Expert Systems with Applications*, 37(3):2661–2669, 2010.

[97] Bahman Naderi and Rubén Ruiz. A scatter search algorithm for the distributed permutation flowshop scheduling problem. *European Journal of Operational Research*, 239(2):323–334, 2014.

[98] Jiafu Tang, Jun Zhang, and Zhendong Pan. A scatter search algorithm for solving vehicle routing problem with loading cost. *Expert Systems with Applications*, 37 (6):4073–4083, 2010.

[99] M Fatih Tasgetiren, Quan-Ke Pan, Ponnuthurai N Suganthan, and Ozge Buyukdagli. A variable iterated greedy algorithm with differential evolution for the no-idle permutation flowshop scheduling problem. *Computers & Operations Research*, 40(7):1729–1743, 2013.

[100] Quan-Ke Pan and Rubén Ruiz. An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem. *Omega*, 44:41–50, 2014.

[101] Quan-Ke Pan and Ling Wang. No-idle permutation flow shop scheduling based on a hybrid discrete particle swarm optimization algorithm. *The International Journal of Advanced Manufacturing Technology*, 39(7-8):796–807, 2008.

[102] Kuo-Ching Ying, Shih-Wei Lin, and Wen-Jie Wu. Self-adaptive ruin-and-recreate algorithm for minimizing total flow time in no-wait flowshops. *Computers & Industrial Engineering*, 101:167–176, 2016.

[103] Jérémie Dubois-Lacoste, Federico Pagnozzi, and Thomas Stützle. An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Computers & Operations Research*, 81:160–166, 2017.

[104] Ibrahim H Osman and CN Potts. Simulated annealing for permutation flow-shop scheduling. *Omega*, 17(6):551–557, 1989.

[105] Richard Lowry. One way anova–independent samples. *Vassar. edu*, 2008.

[106] Dragan Vasiljevic and Milos Danilovic. Handling ties in heuristics for the permutation flow shop scheduling problem. *Journal of Manufacturing Systems*, 35:1–9, 2015.

[107] Douglas C Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2008.

[108] Quan-Ke Pan and Rubén Ruiz. An estimation of distribution algorithm for lot-streaming flow shop problems with setup times. *Omega*, 40(2):166–180, 2012.

[109] Douglas C Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2017.

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

[111] Yuyan Han, Dunwei Gong, Junqing Li, and Yong Zhang. Solving the blocking flow shop scheduling problem with makespan using a modified fruit fly optimisation algorithm. *International Journal of Production Research*, 54(22):6782–6797, 2016.

[112] Yavuz Ince, Korhan Karabulut, M Fatih Tasgetiren, and Quan-ke Pan. A discrete artificial bee colony algorithm for the permutation flowshop scheduling problem with sequence-dependent setup times. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 3401–3408. IEEE, 2016.

[113] Mauricio Iwama Takano and Marcelo Seido Nagano. A branch-and-bound method to minimize the makespan in a permutation flow shop with blocking and setup times. *Cogent Engineering*, page 1389638, 2017.

[114] RA Dudek, ML Smith, and SS Panwalkar. Use of a case study in sequencing/scheduling research. *Omega*, 2(2):253–261, 1974.

[115] JM Framinan, R Leisten, and C Rajendran. Different initial sequences for the heuristic of nawaz, enscore and ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. *International Journal of Production Research*, 41(1):121–148, 2003.

[116] Hamid Abedinnia, Christoph H Glock, and Andreas Brill. New simple constructive heuristic algorithms for minimizing total flow-time in the permutation flowshop scheduling problem. *Computers & Operations Research*, 74:165–174, 2016.

[117] Weibo Liu, Yan Jin, and Mark Price. A new improved neh heuristic for permutation flowshop scheduling problems. *International Journal of Production Economics*, 193:21–30, 2017.

[118] Yamin Wang, Xiaoping Li, Rubén Ruiz, and Shaochun Sui. An iterated greedy heuristic for mixed no-wait flowshop problems. *IEEE Transactions on cybernetics*, 48(5):1553–1566, 2018.

[119] Helena R Lourenco, Olivier Martin, and Thomas Stützle. A beginner's introduction to iterated local search. In *Proceedings of 4th Metaheuristics International Conference.*, 2001.

[120] Rubén Ruiz, Concepción Maroto, and Javier Alcaraz. Solving the flowshop scheduling problem with sequence dependent setup times using advanced metaheuristics. *European Journal of Operational Research*, 165(1):34–54, 2005.

[121] Quan-Ke Pan, Mehmet Fatih Tasgetiren, and Yun-Chia Liang. A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers & Industrial Engineering*, 55(4):795–816, 2008.

[122] M Fatih Tasgetiren, Quan-Ke Pan, Damla Kizilay, and Mario C Vélez-Gallego. A variable block insertion heuristic for permutation flowshops with makespan criterion. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pages 726–733. IEEE, 2017.

[123] ACI. *Airports Council International*, 2012. `http://www.aci.aero/`.

[124] Eurocontrol. *Eurocontrol - Driving excellence in ATM performance*, 2017. `http://www.eurocontrol.int/`.

[125] BTS. *Bureau of Transportation Statistics*, 2017. `ttps://www.bts.gov/`.

[126] BNE. *Brisbane Airport Corporation*, 2017. `http://www.bne.com.au/`.

[127] V Mehta, TG Reynolds, MA Ishutkina, D Joachim, Y Glina, SW Troxel, BJ Taylor, and JE Evans. Airport surface traffic management decision support: Perspectives based on tower flight data manager prototype. Technical report, 2013.

[128] Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: a survey. In *Deterministic and stochastic scheduling*, pages 35–73. Springer, 1982.

[129] John E Beasley, Mohan Krishnamoorthy, Yazid M Sharaiha, and D Abramson. Scheduling aircraft landings—the static case. *Transportation science*, 34(2):180–197, 2000.

[130] Satish Vadlamani and Seyedmohsen Hosseini. A novel heuristic approach for solving aircraft landing problem with single runway. *Journal of Air Transport Management*, 40:144–148, 2014.

[131] A Rodríguez-Díaz, Belarmino Adenso-Díaz, and Pilar Lourdes González-Torre. Minimizing deviation from scheduled times in a single mixed-operation runway. *Computers & Operations Research*, 78:193–202, 2017.

[132] Una Benlic, Alexander EI Brownlee, and Edmund K Burke. Heuristic search for the coupled runway sequencing and taxiway routing problem. *Transportation Research Part C: Emerging Technologies*, 71:333–355, 2016.

[133] Marcella Samà, Andrea D'Ariano, Paolo D'Ariano, and Dario Pacciarelli. Scheduling models for optimal aircraft traffic control at busy airports: tardiness, priorities, equity and violations considerations. *Omega*, 67:81–98, 2017.

[134] Alexander Lieder and Raik Stolletz. Scheduling aircraft take-offs and landings on interdependent and heterogeneous runways. *Transportation research part E: logistics and transportation review*, 88:167–188, 2016.

[135] Ahmed Ghoniem, Farbod Farhadi, and Mohammad Reihaneh. An accelerated branch-and-price algorithm for multiple-runway aircraft sequencing problems. *European Journal of Operational Research*, 246(1):34–43, 2015.

[136] FAA. *American federal aviation administration safety alert for operators (safo)*, 2015. `https://www.faa.gov/other_visit/aviation_industry/airline_operators/airline_safety/safo/`.

[137] HD Sherali, A Ghoniem, H Baik, and AA Trani. A combined arrival-departure aircraft sequencing problem. *Manuscript, Grado Department of Industrial and Systems Engineering (0118). Virginia Polytechnic Institute and State University*, 250, 2010.

[138] Husni Idris, Bertrand Delcaire, Ioannis Anagnostakis, William Hall, Nicolas Pujet, Eric Feron, R Hansman, John-Paul Clarke, and Amedeo Odoni. Identification of flow constraint and control points in departure operations at airport systems. In *Guidance, Navigation, and Control Conference and Exhibit*, page 4291, 1998.

[139] Sander J Heblij and Roland AA Wijnen. Development of a runway allocation optimisation model for airport strategic planning. *Transportation Planning and Technology*, 31(2):201–214, 2008.

[140] SESAR. *SESAR Joint Undertaking — High performing aviation for Europe*, 2017. `http://www.sesarju.eu/`.

[141] A Rodríguez-Díaz, B Adenso-Díaz, and Pilar Lourdes González-Torre. A review of the impact of noise restrictions at airports. *Transportation Research Part D: Transport and Environment*, 50:144–153, 2017.

[142] Sabine A Janssen, Marjolein R Centen, Henk Vos, and Irene van Kamp. The effect of the number of aircraft noise events on sleep quality. *Applied Acoustics*, 84:9–16, 2014.

[143] Nesimi Ozkurt, Deniz Sari, Ali Akdag, Murat Kutukoglu, and Aliye Gurarslan. Modeling of noise pollution and estimated human exposure around istanbul atatürk airport in turkey. *Science of the Total Environment*, 482:486–492, 2014.

[144] Heathrow. *runway alternation — noise — heathrow*, 2017. `https://www.heathrow.com/noise/heathrow-operations/runway-alternation`.

[145] Les Frair. Airport noise modelling and aircraft scheduling so as to minimize community annoyance. *Applied Mathematical Modelling*, 8(4):271–281, 1984.

[146] Nicole Adler, Vanessa Liebert, and Ekaterina Yazhemsky. Benchmarking airports from a managerial perspective. *Omega*, 41(2):442–458, 2013.

[147] Julia A Bennell, Mohammad Mesgarpour, and Chris N Potts. Airport runway scheduling. *Annals of Operations Research*, 204(1):249–270, 2013.

[148] Julia A Bennell, Mohammad Mesgarpour, and Chris N Potts. Airport runway scheduling. *4OR: A Quarterly Journal of Operations Research*, 9(2):115–138, 2011.

[149] Lorenzo Castelli, Raffaele Pesenti, and Andrea Ranieri. The design of a market mechanism to allocate air traffic flow management slots. *Transportation research part C: Emerging technologies*, 19(5):931–943, 2011.

[150] Hamsa Balakrishnan and Bala G Chandran. Optimal large-scale air traffic flow management, 2014.

[151] Fabio Furini, Martin Philip Kidd, Carlo Alfredo Persiani, and Paolo Toth. Improved rolling horizon approaches to the aircraft sequencing problem. *Journal of Scheduling*, 18(5):435–447, 2015.

[152] Andreas T Ernst, Mohan Krishnamoorthy, and Robert H Storer. Heuristic and exact algorithms for scheduling aircraft landings. *Networks*, 34(3):229–241, 1999.

[153] Vic Ciesielski and Paul Scerri. An anytime algorithm for scheduling of aircraft landing times using genetic algorithms. *Australian Journal of Intelligent Information Processing Systems*, 4(3/4):206–213, 1997.

[154] Julia A Bennell, Mohammad Mesgarpour, and Chris N Potts. Dynamic scheduling of aircraft landings. *European Journal of Operational Research*, 258(1):315–327, 2017.

[155] Mayara Condé Rocha Murça and Carlos Müller. Control-based optimization approach for aircraft scheduling in a terminal area with alternative arrival routes. *Transportation research part E: logistics and transportation review*, 73:96–113, 2015.

[156] Irene Moser and Tim Hendtlass. Solving dynamic single-runway aircraft landing problems with extremal optimisation. In *Computational Intelligence in Scheduling, 2007. SCIS'07. IEEE Symposium on*, pages 206–211. IEEE, 2007.

[157] Geert De Maere and Jason AD Atkin. Pruning rules for optimal runway sequencing with airline preferences. *Lecture Notes in Management Science*, 7:76–82, 2015.

[158] JDMAG Abela, D Abramson, M Krishnamoorthy, A De Silva, and Graham Mills. Computing optimal schedules for landing aircraft. In *proceedings of the 12th national conference of the Australian Society for Operations Research, Adelaide*, pages 71–90, 1993.

[159] Lucio Bianco, Giovanni Rinaldi, and Antonio Sassano. A combinatorial optimization approach to aircraft sequencing problem. In *Flow Control of Congested Networks*, pages 323–339. Springer, 1987.

[160] Lucio Bianco, Paolo Dell'Olmo, and Stefano Giordani. Scheduling models and algorithms for tma traffic management, 1997.

[161] Ameer Al-Salem, Farbod Farhadi, Mohamed Kharbeche, and Ahmed Ghoniem. Multiple-runway aircraft sequencing problems using mixed-integer programming. In *IIE Annual Conference. Proceedings*, page 1. Institute of Industrial Engineers-Publisher, 2012.

[162] Ahmed Ghoniem, Hanif D Sherali, and Hojong Baik. Enhanced models for a mixed arrival-departure aircraft sequencing problem. *INFORMS Journal on Computing*, 26(3):514–530, 2014.

[163] Konstantin Artiouchine, Philippe Baptiste, and Christoph Dürr. Runway sequencing with holding patterns. *European Journal of Operational Research*, 189(3): 1254–1266, 2008.

[164] Min Wen, Jesper Larsen, and Jens Clausen. An exact algorithm for aircraft landing problem. Technical report, 2005.

[165] Andrea D'Ariano, Marco Pistelli, and Dario Pacciarelli. Aircraft retiming and rerouting in vicinity of airports. *IET Intelligent Transport Systems*, 6(4):433–443, 2012.

[166] Dionyssios A Trivizas. Optimal scheduling with maximum position shift (mps) constraints: A runway scheduling application. *The Journal of Navigation*, 51(2): 250–266, 1998.

[167] Amrish Ravidas, Sivakumar Rathinam, and Zachary Wood. An optimal algorithm for a two runway scheduling problem. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of aerospace engineering*, 227(7):1122–1129, 2013.

[168] Justin Montoya, Sivakumar Rathinam, and Zachary Wood. Multiobjective departure runway scheduling using dynamic programming. *IEEE Transactions on Intelligent Transportation Systems*, 15(1):399–413, 2014.

[169] Hamsa Balakrishnan and Bala G Chandran. Algorithms for scheduling runway operations under constrained position shifting. *Operations Research*, 58(6):1650–1665, 2010.

[170] Dirk Briskorn and Raik Stolletz. Aircraft landing problems with aircraft classes. *Journal of Scheduling*, 17(1):31–45, 2014.

[171] Torsten Fahle, Rainer Feldmann, Silvia Götz, Sven Grothklags, and Burkhard Monien. The aircraft sequencing problem. In *Computer science in perspective*, pages 152–166. Springer, 2003.

[172] Xiao-Bing Hu and Ezequiel Di Paolo. Binary-representation-based genetic algorithm for aircraft arrival sequencing and scheduling. *IEEE Transactions on Intelligent Transportation Systems*, 9(2):301–310, 2008.

[173] Siliang Wang. Solving aircraft-sequencing problem based on bee evolutionary genetic algorithm and clustering method. In *Dependable, Autonomic and Secure Computing, 2009. DASC'09. Eighth IEEE International Conference on*, pages 157–161. IEEE, 2009.

[174] G Bencheikh, J Boukachour, A El Hilali Alaoui, and FE Khoukhi. Hybrid method for aircraft landing scheduling based on a job shop formulation. *International Journal of Computer Science and Network Security*, 9(8):78–88, 2009.

[175] Jason AD Atkin, Edmund K Burke, John S Greenwood, and Dale Reeson. Hybrid metaheuristics to aid runway scheduling at london heathrow airport. *Transportation Science*, 41(1):90–106, 2007.

[176] Paul W Stiverson. *A study of heuristic approaches for runway scheduling for the Dallas-Fort Worth Airport*. PhD thesis, Texas A & M University, 2010.

[177] Andrea D'Ariano, Dario Pacciarelli, Marco Pistelli, and Marco Pranzo. Real-time scheduling of aircraft arrivals and departures in a terminal maneuvering area. *Networks*, 65(3):212–227, 2015.

[178] Nicholas G Hall and Chelliah Sriskandarajah. A survey of machine scheduling problems with blocking and no-wait in process. *Operations research*, 44(3):510–525, 1996.

[179] Bulent Soykan and Ghaith Rabadi. A tabu search algorithm for the multiple runway aircraft scheduling problem. In *Heuristics, Metaheuristics and Approximate Methods in Planning and Scheduling*, pages 165–186. Springer, 2016.

[180] Ahmed Ghoniem and Farbod Farhadi. A column generation approach for aircraft sequencing problems: a computational study. *Journal of the Operational Research Society*, 66(10):1717–1729, 2015.

[181] Eneko Osaba, Xin-She Yang, Fernando Diaz, Pedro Lopez-Garcia, and Roberto Carballedo. An improved discrete bat algorithm for symmetric and asymmetric

traveling salesman problems. *Engineering Applications of Artificial Intelligence*, 48:59–71, 2016.

[182] Slim Belhaiza, Pierre Hansen, and Gilbert Laporte. A hybrid variable neighborhood tabu search heuristic for the vehicle routing problem with multiple time windows. *Computers & Operations Research*, 52:269–281, 2014.

[183] Bulent Soykan. *A hybrid Tabu/Scatter Search algorithm for simulation-based optimization of multi-objective runway operations scheduling.* PhD thesis, Old Dominion University, 2016.

[184] Marcella Samà, Andrea D'Ariano, Paolo D'Ariano, and Dario Pacciarelli. Optimal aircraft scheduling and routing at a terminal control area during disturbances. *Transportation Research Part C: Emerging Technologies*, 47:61–85, 2014.

[185] MA Hakim Newton, Duc Nghia Pham, Abdul Sattar, and Michael Maher. Kangaroo: An efficient constraint-based local search system using lazy propagation. In *International Conference on Principles and Practice of Constraint Programming*, pages 645–659. Springer, 2011.

[186] Fabio Furini, Carlo Alfredo Persiani, and Paolo Toth. Aircraft sequencing problems via a rolling horizon algorithm. In *International Symposium on Combinatorial Optimization*, pages 273–284. Springer, 2012.

[187] Salvatore Capri and Matteo Ignaccolo. Genetic algorithms for solving the aircraft-sequencing problem: the introduction of departures into the dynamic model. *Journal of Air Transport Management*, 10(5):345–351, 2004.

[188] TA Roemer and RH Ahmadi. The complexity of scheduling customer orders. 2001.

[189] Bertrand MT Lin and Alexander V Kononov. Customer order scheduling to minimize the number of late jobs. *European Journal of Operational Research*, 183(2): 944–948, 2007.

[190] Jianyou Xu, Chin-Chia Wu, Yunqiang Yin, Chuanli Zhao, Yi-Tang Chiou, and Win-Chin Lin. An order scheduling problem with position-based learning effect. *Computers & Operations Research*, 74:175–186, 2016.

[191] Dirk Biskup. Single-machine scheduling with learning considerations. *European Journal of Operational Research*, 115(1):173–178, 1999.

[192] Win-Chin Lin, Yunqiang Yin, Shuenn-Ren Cheng, TC Edwin Cheng, Chia-Han Wu, and Chin-Chia Wu. Particle swarm optimization and opposite-based particle swarm optimization for two-agent multi-facility customer order scheduling with ready times. *Applied Soft Computing*, 52:877–884, 2017.

[193] Chin-Chia Wu, Shang-Chia Liu, Tzu-Yun Lin, Tzu-Hsuan Yang, I-Hong Chung, and Win-Chin Lin. Bicriterion total flowtime and maximum tardiness minimization for an order scheduling problem. *Computers & Industrial Engineering*, 117: 152–163, 2018.

[194] Jose M Framinan, Paz Perez-Gonzalez, and Victor Fernandez-Viagas. Deterministic assembly scheduling problems: A review and classification of concurrent-type scheduling models and solution procedures. *European Journal of Operational Research*, 2018.

[195] Chang Sup Sung and Sang Hum Yoon. Minimizing total weighted completion time at a pre-assembly stage composed of two feeding machines. *International Journal of Production Economics*, 54(3):247–255, 1998.