Constraint Guided Accelerated Search for Mixed Blocking Permutation Flowshop Scheduling

Vahid Riahi*, M A Hakim Newton, Kaile Su, Abdul Sattar

Institute for Integrated and Intelligent Systems (IIIS), Griffith University, Australia

Abstract

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. One key reason behind this is the typical way 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. In this paper, we also present an acceleration method to efficiently evaluate insertion-based neighbourhoods of MBPFSP. Our experimental re-

*170 Kessels Rd, Nathan QLD 4111

Email addresses: vahid.riahi@griffithuni.edu.au (Vahid Riahi), mahakim.newton@griffith.edu.au (M A Hakim Newton), k.su@griffith.edu.au (Kaile Su), a.sattar@griffith.edu.au (Abdul Sattar)

sults on three standard testbeds demonstrate that our proposed algorithms significantly improve over existing best-performing algorithms.

Keywords: Flowshop, Scheduling, Blocking Constraints, Local Search

1. Introduction

Permutation Flowshop Scheduling Problem (PFSP) is one of the most well-known machine scheduling problems. In a PFSP, a sequence of machines are to process a given number of jobs. Each machine must process all the jobs in the same order. In the classical PFSP, it is generally assumed that the intermediate buffer capacity between successive machines is infinite and a job can be stored in a buffer for an unlimited period of time. However, in many real-world industrial problems, there is no buffer between machines, causing blocking of the preceding machine holding its currently finished job until the succeeding machine is available. Such a PFSP is called a Blocking Permutation Flowshop Scheduling Problem (BPFSP) (Lovner, 1969).

Various types of blocking constraint have been identified in real-world industries, for example in the chemical and the pharmaceutical industries (Hall and Sriskandarajah, 1996), and the waste treatment and the aeronautics parts fabrication industries (Martinez de La Piedra, 2005). There are three such known blocking constraints. Under **RSb** (Release when Starting Blocking) (Lovner, 1969) constraint, a machine after completing its current job remains blocked and cannot start processing the next job until the next machine starts processing the current job. Under **RCb** (Release when Completing Blocking) constraint, a machine cannot start processing the next available job until the current job is finished by the next machine. Under **RCb*** (Release when Completing Blocking*) constraint, to start processing the next job by a machine, the current job is to be finished by the next machine regardless of whether the current job leaves the next machine or not. In existing research in the BPFSP literature, it is assumed that we have only one type of blocking constraint between all machines. However, real life production lines are mixed i.e., the capacity of the buffer between two consecutive machines is either unlimited (i.e. **Wb** (Without blocking)) or limited under any of **RSb**, **RCb**, or **RCb*** constraints. Thus, two or more types of blocking constraint can appear simultaneously in the production shops. This real-world problem is called the *Mixed Blocking Permutation Flowshop Scheduling Problem* (MBPFSP).

MBPFSP, as an example, appears in the cider industry (Riahi et al., 2017). This industry consists of seven stages in general: stock, washing, sorting, pressing, fermentation, filtration, and bottling. In the stock step, apples are held in large bins after harvesting from trees. Next, the apples are washed carefully to remove wastes such as leaves, insects and twigs. Then, the apples are sorted based on different types of apples. However, taking into account the limited capacity of washing and sorting machines, apples remain at the stock until those machines are available (**RSb** constraint). After that, to get the juice from pulp, apples are transferred to pressing machines. In the pressing process, after each processing, machines are cleaned and evacuated to get rid of the previous apples. Thus, apples in the sorting machines (previous stage) stay there until pressing is finished, the pressing machine is cleaned and evacuated, and pressed apples leave this stage (**RCb** constraint). Similarly, there is an **RCb** constraint between the pressing and the next stage i.e. the fermentation stage. In the fermentation stage, yeast nutrients such as ammonium sulfate and thiamine, or maybe extra sugar, honey clients or other sweeteners are added to the juice. The amount of those materials is very sensitive to amount of juice. So when apple juices are moved to this stage, the next job must remain in the previous stage until juices leave this stage (**RCb** constraint). The following stage is filtration to completely remove pulps from juice. The capacity of these machines is limited. So juice in the previous stage waits there until jobs (juices) in this stage are finished (**RCb*** constraint). The last stage is bottling which has no blocking. Likely, there are other examples in real-life production lines which the authors are not aware of. Thus, the MBPFSP is a realistic problem that has been rarely studied before and is thus the motivation of this research.

Garey et al. (1976) proved that the PFSP is $\mathcal{NP}$ -hard for more than two machines and with the makespan criterion. Hall and Sriskandarajah (1996) showed that PFSPs with only **RSb** constraints and makespan objective is $\mathcal{NP}$ -hard when the number of machine is more than two. Martinez et al. (2006) showed that PFSP with **RCb** is NP-Hard when the number of machines is at least five. Martinez et al. (2006) also showed that all PFSPs with **RCb** and **RSb** constraints and with more than four machines are NP-Hard. Trabelsi (2012) showed that PFSP with **RCb*** is NP hard for more than three machines. As a result, MBPFSP with **RSb**, **RCb**, **RCb*** and **Wb** is $\mathcal{NP}$ -Hard. Therefore, in order to solve practical-sized problems, heuristic and metaheuristic algorithms are required. In this paper, we propose a new constructive heuristic and an effective search algorithm as metaheuristic.

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. One key reason behind this is the typical way of using generic heuristics or metaheuristics that usually lack problem specific structural knowledge. Moreover, existing such methods use a random neighbourhood generation strategy. Also, their search guidance comes only from the evaluation function that is used after the neighbourhood generation. In MBPFSP, 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.

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. We exploit such knowledge both in a constructive search method and in a local search method e.g. in tie-breaking, in neighbourhood generation, and in perturbation. Our using of problem specific knowledge is in a way generic enough to be used in similar other problems. In this paper, we also present an acceleration method to efficiently evaluate insertion-based neighbourhoods of MBPFSP.

We have evaluated our proposed constructive heuristic on 150 Taillard standard benchmark instances (120 instances of (Taillard, 1993) plus 30 instances added by (Pan and Ruiz, 2012)). Also, we have evaluated our local search method on three testbeds: aforementioned 150 Taillard instances, 240 large instances of Vallada et al. (2015), and 30 very large instances of Lin and Ying (2016) (up to 2000 jobs). A comprehensive set of computational and statistical experiments have demonstrated that our proposed algorithms achieve a very significant improvement over state-of-the-art algorithms.

The rest of the paper is organised as follows: Section 2 discusses the related work; Section 3 provides the preliminary knowledge; Section 4 describes the proposed acceleration method for makespan computation; Section 5 describes the proposed constructive search method; Section 6 describes the proposed constraint guided local search method; Section 7 presents our experimental results; and finally, Section 8 concludes the paper.

2. Related Work

Ruiz and Stützle (2007) proposed an effective iterated greedy algorithm

(IGA) for PFSPs. In each iteration of IGA, a number of jobs are randomly removed from the current solution and then are greedily re-inserted again one by one. Recently, (Lin et al., 2015) proposed a hybrid algorithm based on a new evolutionary algorithm, backtracking search algorithm for PFSPs.

The classical blocking constraint **RSb** (Release when Starting Blocking) (Lovner, 1969) arises in many industries that include the chemical and pharmaceutical industry. Research on PFSP with **RSb** constraints is long. Ribas et al. (2011) proposed an IGA variant (Ruiz and Stützle, 2007) with a new initialisation method, a random-swap based local search, and a simple acceptance criterion of accepting non-improving solutions with 50% probability. Wang et al. (2012) proposed a simulated annealing (SA) algorithm called three phase algorithm (TPA). TPA has an NEH variant for initialisation and it uses only insertion moves. Han et al. (2015) proposed an artificial bee colony algorithm (ABC) using crossover and mutation operators from differential evaluation (DE) techniques. Very recently, Tasgetiren et al. (2017) proposed an IGA variant IG.IJ with a new initialisation heuristic PFT-NEH(x) and a local search method that highly prefers insertion moves to swap moves. 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 Dautère-Pérès et al. (2000). **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 (Martinez de La Piedra, 2005), and an electromagnetism-like (EM) algorithm (Yuan and Sauer, 2007; Yuan et al., 2009). Recently, a new blocking model, **RCb*** (Release when Completing Blocking*), has been introduced by Trabelsi et al. (2010). **RCb*** is seen at cider production factories (Riahi et al., 2017).

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, Trabelsi et al. (2012) proposed a new heuristic named TSS, which is competitive with the well-known NEH (Nawaz et al., 1983) 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, Khorramizadeh and Riahi (2015) proposed a bee colony optimisation algorithm (BCO) and showed it to be outperforming the GA by Trabelsi et al. (2012) on the Taillard benchmarks (Taillard, 1993), which include medium and large-sized problem instances.

Recently, Riahi et al. (2017) proposed a heuristic based on NEH for this problem and showed that it is more effective than the original NEH and the TSS. Besides, they also proposed a metaheuristic algorithm based on the scatter search (SS). They evaluated their algorithm on a newly generated benchmark set, since they proved that the blocking order used by Trabelsi et al. (2012) and Khorramizadeh and Riahi (2015) is not appropriate. The computational results indicated that SS outperforms BCO and GA significantly as it updates 81 out of the 120 best known solutions.

3. Mixed Blocking Permutation Flowshop 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 $1, 2, \dots, m$. 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.

3.1. 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 $i + 1$.

3.2. 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 k th 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 .

3.3. NEH-based Flowshop Heuristics

The original NEH algorithm (Nawaz et al., 1983) is the most used heuristic in the flowshop literature. The effectiveness of the NEH method in flowshop scheduling has been shown in the literature (Framinan and Leisten, 2003; Ruiz and Maroto, 2005). 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 re-inserts 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 $\mathcal{O}(n^3m)$.

There exist some NEH variants that outperform the original one. In one variant, Riahi et al. (2017) focused on the first step of NEH and proposed NNEH, a new variant for MBPFSP. NNEH benefits from a new priority rule that combines initial orderings of Rajendran (1993) and NEH-WPT (Wang et al., 2011). In another variant, Rad et al. (2009) focused on the **prefix insertion** procedure of the NEH and proposed five methods, 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 re-inserted in all positions to get a better fitting of those p jobs.

3.4. MBPFSP Makespan calculation

Let π be a permutation of the given n jobs and $[k]$ denotes the job in the k th 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 (1)–(8) (Riahi et al., 2017). For convenience of computation, in (1), $S_{i,[k]} = 0$ and $C_{i,[k]} = 0$ whenever $i < 1$ or $i > m$ or $k < 1$ or $k > n$.

$$S_{i,[k]} = 0, C_{i,[k]} = 0 \quad i < 1 \vee i > m \vee k < 1 \vee k > n \quad (1)$$

$$C_{i,[k]} = S_{i,[k]} + P_{i,[k]} \quad k = 1, \dots, n, \quad i = 1, \dots, m \quad (2)$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, S_{i+2,[k-1]}) \quad k = 1, \dots, n, \quad i \leq m-2, \quad B_i = \text{RCb} \quad (3)$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, C_{i+1,[k-1]}) \quad k = 1, \dots, n, \quad i \leq m-1, \quad B_i = \text{RCb}^* \quad (4)$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, S_{i+1,[k-1]}) \quad k = 1, \dots, n, \quad i \leq m-1, \quad B_i = \text{RSb} \quad (5)$$

$$S_{i,[k]} = \max(C_{i-1,[k]}, C_{i,[k-1]}) \quad k = 1, \dots, n, \quad i \leq m-1, \quad B_i = \text{Wb} \quad (6)$$

$$S_{m,[k]} = \max(C_{m-1,[k]}, C_{m,[k-1]}) \quad k = 1, \dots, n \quad (7)$$

$$C(\pi) = C_{m,[n]} \quad (8)$$

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 .

$$C(\pi_*) \leq C(\pi) \quad \forall \pi \quad (9)$$

Lemma 1. *Computing $C(\pi)$ from scratch using (1)–(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. $\square$

4. Proposed Accelerated Makespan Computation

Insertion-based operators are the most frequently used neighbourhood operator in the flowshop scheduling literature. As per Lemma 1, computing $C(\pi)$ from scratch takes $\mathcal{O}(mn)$ time. So the time complexities of calculating makespan from scratch for all permutations in $\text{insert-neighbourhood}(\pi, k)$ and $\text{insert-neighbourhood}(\pi)$ are $\mathcal{O}(mn^2)$ and $\mathcal{O}(mn^3)$ respectively. Thus using $\text{insert-neighbourhood}(\pi, k)$ and $\text{insert-neighbourhood}(\pi)$ are computationally very costly particularly for large-sized problems. Inspired by the acceleration method proposed by Taillard (1990) for PFSP, in this paper, we develop an acceleration method to efficiently compute $C(\pi')$ for each π' produced by $\text{insert-neighbourhood}(\pi, k)$ and $\text{insert-neighbourhood}(\pi)$.

4.1. Forward and Backward Calculations

Computation of $C(\pi)$ by using (1)–(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*.

$$T_{i,[k]} = 0, P_{i,[k]} = 0 \quad i < 1 \vee i > m \vee k < 1 \vee k > n \quad (10)$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, T_{i-1,[k+1]}, \\ (T_{i-2,[k+1]} \text{ when } B_{i-2} = \text{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \text{RSb} \quad (11)$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, T_{i-1,[k+1]} + P_{i,[k]}, \\ (T_{i-2,[k+1]} \text{ when } B_{i-2} = \text{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \text{RCb}^* \quad (12)$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, \\ (T_{i-2,[k+1]} \text{ when } B_{i-2} = \text{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \text{RCb} \quad (13)$$

$$T_{i,[k]} = \max(T_{i+1,[k]} + P_{i,[k]}, T_{i,[k+1]} + P_{i,[k]}, \\ (T_{i-2,[k+1]} \text{ when } B_{i-2} = \text{RCb})) \quad i \geq 2, n \geq k, B_{i-1} = \text{Wb} \quad (14)$$

$$T_{1,[k]} = \max(T_{2,[k]}, T_{1,[k+1]}) + P_{1,[k]} \quad n \geq k \geq 1 \quad (15)$$

Example MBPFSP schedules with 4 jobs and 5 machines are shown in Figure 1 to illustrate the forward (top) and the backward (bottom) calculations of the makespan. According to the reversibility property of flowshops (Pinedo, 2016), 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.

Figure 1: MBPFSP schedules by forward (top) and backward calculation (bottom)

4.2. 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 $\text{remove}(\pi, k')$ to produce an intermediate permutation π' , and then we also have applied $\text{insert}(\pi', j, k'')$ 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 (1)–(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 (10)–(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 (1)–(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 (16)–(19) below.

$$L''_{i,[k'']} = S''_{i+2,[k'']} \quad B_i = \text{RCb} \quad (16)$$

$$L''_{i,[k'']} = S''_{i+1,[k'']} \quad B_i = \text{RSb} \quad (17)$$

$$L''_{i,[k'']} = C''_{i+1,[k'']} \quad B_i = \text{RCb}^* \quad (18)$$

$$L''_{i,[k'']} = C''_{i,[k'']} \quad B_i = \text{Wb} \quad (19)$$

Given all $L''_{i,[k'']}$ and $T''_{i,[k'']}$, in (20), we can compute $C(\pi'')$ from the time points when the machines complete processing all jobs.

$$C(\pi'') = \max_{i=1}^m (L''_{i,[k'']} + T''_{i,[k'']}) \quad (20)$$

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 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 2 (top). Moreover, $L''_{i,[k]}$, $T''_{i,[k]}$ values for π'' in Figure 2 (bottom) demonstrates how the makespan is essentially calculated. The same examples shown graphically in Figure 1 and 2 are shown numerically below. The blocking constraints between successive machines are $\langle \text{RCb}, \text{RCb}^*, \text{Wb}, \text{RSb} \rangle$. The processing times for each job at each machine are in the matrix $[P_{i,j}]_{5 \times 5}$ below.

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

Figure 2: An example of MBPFSP schedule after inserting job 5 at position 3 (top) and an illustration of its incremental makespan calculation (bottom)

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

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 (1)–(8) of forward calculation and (10)–(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:

$$[S''_{i,[3]}]_{5 \times 1} = \begin{bmatrix} 8 \\ 10 \\ 12 \\ 14 \\ 17 \end{bmatrix} \quad [C''_{i,[3]}]_{5 \times 1} = \begin{bmatrix} 10 \\ 12 \\ 13 \\ 17 \\ 18 \end{bmatrix} \quad [L''_{i,[3]}]_{5 \times 1} = \begin{bmatrix} 12 \\ 14 \\ 14 \\ 17 \\ 18 \end{bmatrix} \quad [T''_{i,[3]}]_{5 \times 1} = \begin{bmatrix} 12 \\ 9 \\ 8 \\ 6 \\ 5 \end{bmatrix}$$

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

4.3. Incremental Evaluation of Insertion Neighbourhood

Given the incremental makespan computation in Section 4.2, we develop an efficient evaluation method for the permutations in the insertion neighbourhoods. Note that for incremental computation of all permutations in $\text{insert-neighbourhood}(\pi, k')$ where $j = [k']$ in π , we need to apply $\text{remove}(\pi, k')$ only once to get π' and then we apply $\text{insert}(\pi', j, k'')$ for each k'' .

1. $\text{remove}(\pi, k')$: 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' in Section 4.2
 - (a) Calculate $S'_{i,[k]}$ and $C'_{i,[k]}$ for $i = 1, \dots, m$ and $k = 1, \dots, n - 1$ for π' .
 - (b) Calculate $T'_{i,[k]}$ for $i = m, \dots, 1$ and $k = n - 1, \dots, 1$ for π' .
3. For $k'' = 1, \dots, n$ do the following steps:
 - (a) $\text{insert}(\pi', j, k'')$: Insert job j at position k'' of π' to obtain π'' .
 - (b) Using incremental computation of S'' , C'' , and T'' for π'' in Section 4.2
 - i. Calculate $S''_{i,[k'']}$ and $C''_{i,[k'']}$ for $i = 1, 2, \dots, m$ for π'' .
 - ii. Calculate $T''_{i,[k'']}$ for $i = m, m - 1, \dots, 1$ for π'' .
 - iii. Calculate $L''_{i,[k'']}$ for $i = 1, 2, \dots, m$ for π'' ;
 - iv. Calculate $C(\pi'')$ for π'' using (20).

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

Lemma 2. *The above incremental method computes makespan of the permutations produced by $\text{insert-neighbourhood}(\pi, k')$ and $\text{insert-neighbourhood}(\pi)$ in $\mathcal{O}(mn)$ and $\mathcal{O}(mn^2)$ time respectively. 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 $\text{insert-neighbourhood}(\pi, k')$ $\text{insert-neighbourhood}(\pi)$ are $\mathcal{O}(mn)$ and $\mathcal{O}(mn^2)$ respectively.

5. Proposed Constructive Heuristic

In this paper, we propose an NEH-based constructive heuristic named Improved NEH or INEH that uses elements from two other NEH-variants NNEH (Riahi et al., 2017) and FRB4 (Rad et al., 2009). We then further improve INEH by using a constraint-guided tie-breaking method.

5.1. 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 re-inserted in position k' , jobs at position $k' - 1$ (if $k' > 1$) and $k' + 1$ (if $k' < k$) are also reinserted. Algorithm 1 presents the INEH algorithm.

Lemma 3. *Given a constant p , the INEH algorithm shown in Algorithm 1 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 $\mathcal{O}(pn^2)$ partial solutions and each of these needs $\mathcal{O}(nm)$ time to compute the makespan value from scratch using $\mathcal{O}(n)$ space. Assuming a constant p , the INEH algorithm thus takes $\mathcal{O}(n^3m)$ time (if the makespan is computed from scratch) and $\mathcal{O}(n)$ space. However, with the proposed acceleration method, the time complexity will be $\mathcal{O}(n^2m)$.

Algorithm 1: 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 endfor
5 Sort the jobs in non-decreasing order of  $A[j]$  breaking ties in favor of
   the lowest  $j$  to get a sequence  $\pi$ .
6 // Insertion procedure by FRB4 below
7 for  $k = 1$  to  $n$  with a given tie-breaking strategy do
8   | Find the schedule  $\pi'$  with lowest  $C_{m,[k]}$  by inserting job  $[k]$  in all
     | the possible positions  $k' \leq k$  in  $\pi$ . Let  $\bar{k}$  is  $k'$  with the lowest
     |  $C_{m,[k]}$ .
9   | Find the schedule  $\pi''$  with lowest  $C_{m,[k]}$  by inserting job  $[k']$  of  $\pi'$ 
     | in all the possible positions  $k'' \leq k$  in the sequence  $\pi'$  where
     |  $k' \in [\bar{k} - p, \bar{k} + p]$  and  $1 \leq k' \leq k$  and  $p = 1$ .
10 endfor
11 return  $\pi''$ 

```

5.2. 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 (Kalczyński and Kamburowski, 2007; Fernandez-Viagas and Framinan, 2015). Different simple tie-breaking method in the literature are used: selecting the first tie (Pan and Wang, 2012; Riahi et al., 2018a), or selecting random one (Ribas et al., 2011). However, as mentioned by Pan and Wang (2012) and Fernandez-Viagas and Framinan (2015), 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 21, and the solution with the

Figure 3: An example of the problem-dependent tie breaking

smallest TWT value is considered to be better than the other.

$$\text{TWT}(\pi) = \sum_{i=1}^m L_{i,[n]} - \sum_{j=1}^n P_j \quad (21)$$

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

Table 1: A MBPFSP instance with 3 jobs and 6 machines.

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

Consider the problem instance described in Table 1. The blocking constraints between the 6 machines are $\langle \text{RCb}, \text{RCb}^*, \text{RSb}, \text{RCb}^*, \text{RCb}^* \rangle$. 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.

As can be seen from Figure 3, 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.

6. 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 2. 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.

Algorithm 2: Local Search Framework

1. $\pi \leftarrow \text{initialiseSolution}()$
 2. $\pi \leftarrow \text{performIntensification}(\pi)$
 3. **while** *termination criteria not satisfied*
 - $\pi' \leftarrow \text{performDiversification}(\pi)$
 - $\pi' \leftarrow \text{performIntensification}(\pi')$
 - $\pi \leftarrow \text{acceptForNextIteration}(\pi')$
- return** the global best solution found so far
-

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 guidances in different phases of the local search framework in Algorithm 2. We name the resultant algorithm as Constraint Guided Local Search (CGLS) for MBPFSP.

6.1. 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 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 (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 $B(k)$.

$$B(k) = \sum_{i=1}^{m-1} (B(i, k)) \quad (22)$$

where

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

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 $\hat{\pi}$.

Assume a solution $\pi = \langle 2, 3, 1, 4 \rangle$ of 4 jobs that are processed on 5 machines with blocking order $\langle \text{RCb}^*, \text{RSb}, \text{RCb}, \text{RCb}^* \rangle$ is depicted in Figure 4. 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 $\hat{\pi} = \langle 2, 4, 3, 1 \rangle$.

Figure 4: Blocking times associated with jobs

The outline of the proposed intensification method is given in Algorithm 3. In its first and second steps, it obtains the job ordering $\hat{\pi}$. In its third step, for each job j in $\hat{\pi}$ in order, it explores the permutations in $\text{insert-neighbourhood}(\pi, k')$ where $[k'] = j$. If an improving solution is found, the procedure restarts.

Algorithm 3: 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 $\hat{\pi} = (\hat{\pi}_1, \hat{\pi}_2, \dots, \hat{\pi}_n)$.
 3. **foreach** $k = 1$ **to** n **do**
 - (a) Take the job $\hat{\pi}_k$ 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 $\text{insert-neighbourhood}(\pi, k')$.
 - (b) **if** $C(\pi') < C(\pi)$ **then** let $\pi = \pi'$ and go to Step 1
-

6.2. 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, Ruiz and Stützle (2007) removes a number of randomly selected jobs but then inserts each job greedily by finding its best position (just as in NEH). This procedure has been used by Ribas et al. (2011); Tasgetiren et al. (2013); Pan and Ruiz (2014); Pan and Wang (2008); Ying et al. (2016); Riahi et al. (2018b). In this work, 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 $\text{TDS} - \text{GDS}$. These two parameters will be experimentally calibrated in Section 7.2.

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 2 with blocking constraints $\langle \text{RSb}, \text{Wb}, \text{RCb}, \text{RCb}^* \rangle$, assume $\pi = \langle 1, 2, 3, 4 \rangle$ is permutation. The corresponding schedule is shown in Figure 5. 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 $\text{GDS} = 2$, then jobs $[3]$ and $[4]$ are selected greedily.

Table 2: An example of 4 jobs and 5 machines.

Job j	Machine i					Processing Time P_j
	1	2	3	4	5	
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 5: An example schedule for an MBPFSP in Table 2

Intensification of Partial Solutions. Local search have been applied in various ways on partial sequence by Rad et al. (2009); Pan and Ruiz (2014); Dubois-Lacoste et al. (2017). 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 6.1 on π' to obtain a better permutation. We then insert each of the TDS – 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 4 shows the pseudocode of the procedure.

6.3. Acceptance Criterion

Among many acceptance criterion procedure used in the literature, the one proposed by Osman and Potts (1989) and used by Ruiz and Stützle (2007) 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

Algorithm 4: performDiversification(π)

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

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 (23) is used to determine acceptance probability P where TAR (Threshold Acceptance Ratio) is a parameter.

$$P = \begin{cases} 1 & \text{if } \Delta \leq \text{TAR} \times \Delta_0, \\ 0 & \text{otherwise,} \end{cases} \quad (23)$$

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 (Talbi, 2009), 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. As mentioned, TAR is a parameter that will be calibrated in Section 7.2.

7. Computational Results

We perform experiments to test the effectiveness of the proposed INEH and INEHtie initialisation heuristics, to evaluate the components of the proposed CGLS algorithm, 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 benchmark instances of Taillard (1993). 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 $\{500\} \times \{5, 10\}$. Pan and Ruiz (2012) 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.
2. **VRF Instances.** The second benchmark set has 480 new hard benchmark instances of Vallada et al. (2015). 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 instances generated by Lin and Ying (2016) using the same experiment design as Taillard (1993). These instances comprise three combinations of $n = \{1000, 1500, 2000\}$ with 20 machines.

We compare our INEH 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. **SS**: a scatter search algorithm by Riahi et al. (2017)
2. **BCO**: a bee colony optimisation algorithm by Khorramizadeh and Riahi (2015)
3. **IGA-RS**: an iterated greedy algorithm by Ruiz and Stützle (2007)
4. **HBS**: a hybrid backtracking search algorithm by Lin et al. (2015)
5. **TPA**: a three-phase algorithm by Wang et al. (2012)
6. **IGA-R**: an iterated greedy algorithm by Ribas et al. (2011)
7. **ABC-DE**: a hybrid of artificial bee colony algorithm and differential evolution by Han et al. (2015)
8. **IG_IJ**: an iterated greedy algorithm by Tasgetiren et al. (2017)

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

$$RPD = \frac{C - C_{\text{ref}}}{C_{\text{ref}}} \times 100 \quad (24)$$

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 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 CGLS, 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.

7.1. 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 Nawaz et al. (1983) for PFSP, and later used in BCO (Khorramizadeh and Riahi, 2015) for MBPFSP and in IGA (Ruiz and Stützle, 2007) for PFSP
2. **mNEH**: proposed in TPA (Wang et al., 2011) for BPFSP with RSb constraints
3. **MME**: used in IGA-R (Ribas et al., 2011) and in ABC-DE (Han et al., 2015), both for BPFSP with RSb constraints
4. **PFT-NEH**: proposed in IG-IJ (Tasgetiren et al., 2017) for BPFSP with RSb constraints
5. **NNEH**: proposed in SS (Riahi et al., 2017) for MBPFSP

The comparison is done using the 150 problem instances of the (Tallard, 1993) benchmark set. Table 3 (bottom) shows that computation of the heuristics often take negligible times. Table 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 Tukeys Honest Significant Difference (HSD) is shown in Figure 6 Left. 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 6 Right. The conclusions remain the same both on the VFF and the LY instances.

Table 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

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:

Figure 6: ARPD and 95% confidence intervals with Tukeys Honest Significant Difference (HSD) for different heuristics: Left) Taillard's instances Right) VRF instances.

1. **FT**: chooses the first one over any subsequent ones (Nawaz et al., 1983)
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 $IT1 = \sum_{i=1}^m (C_{i,[n]}) - \sum_{j=1}^n P_j$ Fernandez-Viagas and Framinan (2015)
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 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 Vasiljevic and Danilovic (2015) on handling ties for PFSP. First, MBPFSPs have significantly different characteristics than FSPs. 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 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 CGLS. In addition, for a fair comparison, INEHtie method is also used in all compared

metaheuristic algorithms.

7.2. CGLS Parameter Tuning

We use the Design of Experiments method (Montgomery, 2017) 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 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 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 $12[\text{DS}] \times 3[\text{TAR}] \times 5[\text{replication}] \times 60[\text{instance}] = 10800$ runs.

Table 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 7: ARPD and 95% confidence intervals with Tukeys Honest Significant Difference (HSD) for different values of parameters, DS: Left and TAR: Right.

The ANOVA results are given in Table 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 Honest Significant Difference (HSD) are shown in Figure 7. From Figure 7-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, 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 7-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.

7.3. 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, Ruiz and Stützle (2007) showed that the temperature factor is very robust. The same is in IG.IJ (Tasgetiren et al., 2017) 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 7. Moreover, ARPD and 95% confidence intervals with HSD for different values of the algorithm parameters are provided as online material.

Table 7: The selected parameter values of the algorithms compared.

Algorithm	Parameter	Selected value
SS (Riahi et al., 2017)	SizeofPop	45
	(R1, R2)	(7,3)
	MaxIter	13
BCO (Khorramizadeh and Riahi, 2015)	P_{size}	10
	p_{LS}	0.05
	MaxIter	$P_{size} \times 2$
	MaxLoop	$P_{size} \times 0.5$
IG.IJ (Tasgetiren et al., 2017)	dS	5
	tP	0.5
	jP	0.01
IGA-RS (Ruiz and Stützle, 2007)	d	3
	T	0.4
IGA-R (Ribas et al., 2011)	d	4
	γ	7
TPA (Wang et al., 2012)	N_{iter}	200000
	P	30
HBS (Lin et al., 2015)	T_0	30
	λ	0.90
	Mu	0.2
	CR	0.9
	PS	15

ABC-DE (Han et al., 2015)

<i>pmu</i>	0.9
<i>pc</i>	0.3
<i>pls</i>	0.2

7.4. 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 7.2 and shown in Figure 8. Based on the results, the INEH is obviously faster than INEH_{NS}.

Figure 8: Effect of the acceleration method on INEH

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 7.2. 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 average iterations of each instance group is shown in Table 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 9. As can be seen from the figure, there is a statistically very significant difference between CGLS and CGLS_{NS} .

Table 8: ARPD and number of iterations of CGLS and CGLS_{NS} .

Instance n×m	ARPD		Iteration		Iteration diff (%)
	CGLS	CGLS_{NS}	CGLS	CGLS_{NS}	
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

Figure 9: 95% confidence intervals with Tukey's Honest Significant Difference (HSD) for CGLS and CGLS_{NS} .

7.5. Effectiveness of Proposed Algorithm 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 Ruiz and Stützle (2007).
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* (Riahi et al., 2017). 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 9.

As can be seen from Table 9, 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 Tukeys Honest Significant Difference (HSD) with a significance level of 0.05 is performed.

According to Figure 10, 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 effectiveness of the constraint aware

Table 9: 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

Figure 10: 95% confidence intervals with Tukey's Honest Significant Difference (HSD) for CGLS variants.

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

7.6. Comparison With Existing Algorithms

To ascertain the CGLS's overall performance, we compare it with the 8 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 7 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 (Taillard (1993)). The ARPDs of the algorithms for each instance group are shown in Tables 10-12. 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 13. As shown, at a confidence level of $\alpha = 0.05$, the ARPDs of the competing algorithms are statistically different because of $p\text{-value} < 0.05$. The mean plots with 95% Tukeys Honest Significant Difference (HSD) of the compared algorithms are also reported in Figure 11. 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 12. 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 10: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 30$) for Taillard (1993) instances. 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 11: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 60$) for Taillard (1993) instances. 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)

Table 12: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 90$) for Taillard (1993) instances. 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 13: Comparison of CGLS variants

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

Figure 11: Means plots with 95% Tukey's Honest Significant Difference (HSD) of the competing algorithms for the 150 Taillard (1993) instances in three different timeout scenarios.

Figure 12: 95% confidence interval of the competing algorithms for the 150 Taillard (1993) instances in different number of jobs and machines.

As the second experiment, the algorithms compared are tested on the 240 large VRF hard benchmarks (Vallada et al., 2015). The ARPDs and the ranks of the algorithms for each instance group are shown in Tables 14-16. In all instance groups and for each timeout, the 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, Riahi et al. (2017) showed that the SS algorithm is significantly better than BCO

algorithm. 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 because of employing very stronger 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.

Table 14: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 30$) for VRF instances Vallada et al. (2015). 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)

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 17. From this table, we can conclude that there is a significant difference between at least two algorithms as p-value < 0.05 . Consequently, mean plots with 95% HSD interval plot of algorithms for each timeout are shown in Figure 13 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.

Table 15: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 60$) for VRF instances Vallada et al. (2015). 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 16: ARPDs and the ranks (in brackets) of the algorithms ($\rho = 90$) for VRF instances Vallada et al. (2015). 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 17: ANOVA results of the compared algorithms for VRF instances.

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

Figure 13: Means plots with 95% Tukeys Honest Significant Difference (HSD) of the competing algorithms for the 240 VRF instances in three different timeout scenarios.

As the third testbed, the algorithms are tested on the 30 very large LY instances (Lin and Ying, 2016). The results are summarized in Table 18. As we observe, this experiment confirms the same conclusions discussed above. An ANOVA test and 95% confidence intervals are also shown in Table 19 and Figure 14 respectively. The ANOVA table says that there is a statistically significance among compared algorithms, and Figure 14 make it clear that this significant difference is between which algorithms. The proposed CGLS algorithm statistically outperforms other algorithms.

Table 18: ARPDs and the ranks (in brackets) of the algorithms for LY instances Lin and Ying (2016). 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)

Table 19: ANOVA results of the compared algorithms for LY instances.

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

Figure 14: Means plots with 95% Tukey's Honest Significant Difference (HSD) of the competing algorithms for the 30 LY instances in three different timeout scenarios.

7.7. Best Known Solutions

In this section, the best know solution found by the algorithms compared for all three testbeds, 150 Taillard instances (Taillard, 1993), 240 large VRF instances (Vallada et al., 2015), and 30 very large LY instances (Lin and Ying, 2016) are given in Tables 20-22. These tables are used in this paper to calculate the RPDs of the results, and can also be used by future researchers.

Table 20: 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 21: 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 22: 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

8. Conclusions

In this paper 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. In this paper, 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, in this paper, we have presented a problem-dependent threshold-based simple acceptance criterion and a very effective acceleration method for makespan calculation with insertion based operators. We have performed required experiments to tune the algorithm parameters and evaluated our algorithms using three well-known benchmark problem instances. Overall our proposed algorithm has significantly outperformed existing state-of-the-art algorithms for MBPFSP.

Acknowledgement

This research is supported by the Australian Research Council under Grant DP150101618.

References

- Dauzère-Pérès, S., Pavageau, C., and Sauer, N. (2000). Modélisation et résolution par plne dun problème réel dordonnancement avec contraintes de blocage. *3ème congrès de la société Française de Recherche Opérationnelle et d'Aide à la Décision, Nantes*.
- Dubois-Lacoste, J., Pagnozzi, F., and Stützle, T. (2017). An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Computers & Operations Research*, 81:160–166.
- Fernandez-Viagas, V. and Framinan, J. M. (2015). NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness. *Computers & Operations Research*, 60:27–36.
- Framinan, J. and Leisten, R. (2003). An efficient constructive heuristic for flowtime minimisation in permutation flow shops. *Omega*, 31(4):311–317.
- Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129.
- Hall, N. G. and Sriskandarajah, C. (1996). A survey of machine scheduling problems with blocking and no-wait in process. *Operations research*, 44(3):510–525.
- Han, Y.-Y., Gong, D., and Sun, X. (2015). A discrete artificial bee colony algorithm incorporating differential evolution for the flow-shop scheduling problem with blocking. *Engineering Optimization*, 47(7):927–946.
- Kalczynski, P. J. and Kamburowski, J. (2007). On the NEH heuristic for minimizing the makespan in permutation flow shops. *Omega*, 35(1):53–60.
- Khorramizadeh, M. and Riahi, V. (2015). A bee colony optimization approach for mixed blocking constraints flow shop scheduling problems. *Mathematical Problems in Engineering*, 2015.
- Lin, Q., Gao, L., Li, X., and Zhang, C. (2015). A hybrid backtracking search algorithm for permutation flow-shop scheduling problem. *Computers & Industrial Engineering*, 85:437–446.

- Lin, S.-W. and Ying, K.-C. (2016). Optimization of makespan for no-wait flowshop scheduling problems using efficient matheuristics. *Omega*, 64:115–125.
- Lovner (1969). Optimal planning of parts machining on a number of machines. *Automation and Remote Control*, 12:1972–1978.
- Martinez, S., Dauzère-Pérès, S., Gueret, C., Mati, Y., and Sauer, N. (2006). Complexity of flowshop scheduling problems with a new blocking constraint. *European Journal of Operational Research*, 169(3):855–864.
- Martinez de La Piedra, S. (2005). *Ordonnancement de systèmes de production avec contraintes de blocage*. PhD thesis, Nantes.
- Montgomery, D. C. (2017). *Design and analysis of experiments*. John Wiley & Sons.
- Nawaz, M., Enscore, E. E., and Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95.
- Osman, I. H. and Potts, C. (1989). Simulated annealing for permutation flow-shop scheduling. *Omega*, 17(6):551–557.
- Pan, Q.-K. and Ruiz, R. (2012). Local search methods for the flowshop scheduling problem with flowtime minimization. *European Journal of Operational Research*, 222(1):31–43.
- Pan, Q.-K. and Ruiz, R. (2014). An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem. *Omega*, 44:41–50.
- Pan, Q.-K. and Wang, L. (2008). 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.
- Pan, Q.-K. and Wang, L. (2012). Effective heuristics for the blocking flowshop scheduling problem with makespan minimization. *Omega*, 40(2):218–229.
- Pinedo, M. L. (2016). *Scheduling: theory, algorithms, and systems*. Springer.

- Rad, S. F., Ruiz, R., and Boroojerdian, N. (2009). New high performing heuristics for minimizing makespan in permutation flowshops. *Omega*, 37(2):331–345.
- Rajendran, C. (1993). Heuristic algorithm for scheduling in a flowshop to minimize total flowtime. *International Journal of Production Economics*, 29(1):65–73.
- Riahi, V., Khorramizadeh, M., Newton, M. H., and Sattar, A. (2017). Scatter search for mixed blocking flowshop scheduling. *Expert Systems with Applications*, 79:20–32.
- Riahi, V., Newton, M. H., Su, K., and Sattar, A. (2018a). Local search for flowshops with setup times and blocking constraints. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 199–207.
- Riahi, V., Polash, M., Newton, M. H., and Sattar, A. (2018b). Mixed neighbourhood local search for customer order scheduling problem. In *Pacific Rim International Conference on Artificial Intelligence*, pages 296–309. Springer.
- Ribas, I., Companys, R., and Tort-Martorell, X. (2011). An iterated greedy algorithm for the flowshop scheduling problem with blocking. *Omega*, 39(3):293–301.
- Ruiz, R. and Maroto, C. (2005). A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479–494.
- Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049.
- Taillard, E. (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65–74.
- Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.

- Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons.
- Tasgetiren, M. F., Kizilay, D., Pan, Q.-K., and Suganthan, P. N. (2017). Iterated greedy algorithms for the blocking flowshop scheduling problem with makespan criterion. *Computers & Operations Research*, 77:111–126.
- Tasgetiren, M. F., Pan, Q.-K., Suganthan, P. N., and Buyukdagli, O. (2013). A variable iterated greedy algorithm with differential evolution for the no-idle permutation flowshop scheduling problem. *Computers & Operations Research*, 40(7):1729–1743.
- Trabelsi, W. (2012). *Ordonnancement des systèmes de production flexibles soumis à différents types de contraintes de blocage*. PhD thesis, Université de Lorraine.
- Trabelsi, W., Sauvey, C., and Sauer, N. (2010). Heuristic methods for problems with blocking constraints solving jobshop scheduling. In *Proceedings of the eighth International Conference on Modelling and Simulation, Hammamet, MOSIM*.
- Trabelsi, W., Sauvey, C., and Sauer, N. (2012). Heuristics and metaheuristics for mixed blocking constraints flowshop scheduling problems. *Computers & Operations Research*, 39(11):2520–2527.
- Vallada, E., Ruiz, R., and Framinan, J. M. (2015). New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677.
- Vasiljevic, D. and Danilovic, M. (2015). Handling ties in heuristics for the permutation flow shop scheduling problem. *Journal of Manufacturing Systems*, 35:1–9.
- Wang, C., Song, S., Gupta, J. N., and Wu, C. (2012). A three-phase algorithm for flowshop scheduling with blocking to minimize makespan. *Computers & Operations Research*, 39(11):2880–2887.
- Wang, L., Pan, Q.-K., and Tasgetiren, M. F. (2011). A hybrid harmony search algorithm for the blocking permutation flow shop scheduling problem. *Computers & Industrial Engineering*, 61(1):76–83.

- Ying, K.-C., Lin, S.-W., and Wu, W.-J. (2016). Self-adaptive ruin-and-recreate algorithm for minimizing total flow time in no-wait flowshops. *Computers & Industrial Engineering*, 101:167–176.
- Yuan, K. and Sauer (2007). Application of em algorithm to flowshop scheduling problems with a special blocking. In *Proceedings of the ISEM*.
- Yuan, K., Sauer, N., and Sauvey, C. (2009). 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*.