MIP-Based Local Search for Permutation Flowshop Scheduling with Makespan Objective

Sebastian Cáceres-Gelvez[1,2], Thu Huong Dang[1] and Adam N. Letchford[1]

[1]Department of Management Science, Lancaster University,, Lancaster LA1 4YX, UK. E-mail:, {s.caceresgelvez,t.h.dang,a.n.letchford}@lancaster.ac.uk

[2]Universidad de Santander, Facultad de Ingenierías, Grupo de Investigación EUREKA UDES, Cúcuta, Colombia

To appear in *Computers & Operations Research*

Abstract

The permutation flowshop scheduling problem with makespan objective, or PFM for short, is a classic NP-hard scheduling problem. At present, the most promising heuristics for the PFM are based on variations of local search. This led us to consider five new neighbourhoods for the PFM. Each neighbourhood is of exponential size, but can be explored quite quickly by solving a small mixed-integer program. We propose a *matheuristic* framework that incorporates our proposed neighbourhoods to evaluate and compare their effectiveness. Extensive computational experiments show that integrating our best neighbourhood to the proposed matheuristic reduces the makespan by over 60% on average, compared to the variant without it, on both the classical Taillard benchmark instances and the more recent instances proposed by Vallada, Ruiz and Framinan

Keywords: flowshop scheduling; permutation flowshop; very large-scale neighbourhood search; matheuristics

1 Introduction

Machine scheduling problems are an important family of combinatorial optimisation problems, and they have received a great deal of attention from the Operational Research and Optimisation communities. In this paper, we consider the *permutation flowshop scheduling problem with makespan objective*, or PFM for short.

In the PFM, m machines are arranged in a flow line, numbered from 1 to m, and n jobs must pass through each machine in the flow line one after the other, i.e., each job must be processed on machine 1, then machine 2, and so on. Although the job order is identical, each machine represents a different processing stage with its own processing times. Specifically, each machine can process only one job at a time, and the processing time of job j on machine i is p_{ij}. The aim is to determine a *sequence* of jobs to be processed on machines that minimises the *makespan*, i.e., the time taken to finish processing the last job on the last machine. The PFM is often denoted as $F_m|prmu|C_{max}$, following the three-field notation in [14].

Johnson [17] showed that the PFM with $m = 2$ can be solved in polynomial time. For general m, however, the PFM is NP-hard in the strong sense [18]. A wide range of approaches are available, including exact methods (e.g., [12, 16, 33]), heuristics (e.g., [8, 10, 26]) and approximation algorithms [2, 20, 28].

At present, the most promising heuristics for the PFM are based on variations of local search (e.g., [5, 8, 10]). Local search procedures aim to improve a solution by exploring its neighbours, but existing approaches in the PFM literature often rely on evaluating a large number of candidate moves, which can be time-consuming. In this study, we propose five new neighbourhoods for the PFM and demonstrate how to formulate the neighbourhood exploration as a small *mixed-integer program* (MIP), enabling us to identify the best possible neighbour by using a MIP solver as a black box. (The idea of using a MIP solver to explore a neighbourhood is in line with the 'MIPping' philosophy introduced by Fischetti *et al.* [9].)

While heuristic and metaheuristic approaches dominate the literature on the PFM, exact methods have rarely been incorporated into neighborhood search procedures, largely due to concerns over computational overhead. To the best of our knowledge, the only exception is the work of Haouari and Ladhari [15], who proposed a branch-and-bound-based local search algorithm. However, their experiments were limited to Taillard instances (see [31]) with $m \leq 10$.

As a second contribution of this study, we present a hybrid heuristic for the PFM, which combines the 'NEH' heuristic [21] with our proposed neighbourhood search procedures. This hybrid heuristic enables us to evaluate the relative performance of our proposed neighborhoods. (It is also an example of a *matheuristic*; see Subsection 2.4.) After explaining the hybrid heuristic, we give extensive computational results on the benchmark instances of Taillard [31] and Vallada *et al.* [34].

The paper has a simple structure. Section 2 gives a brief overview of the relevant literature. Section 3 describes our new neighbourhoods. Section 4 presents our matheuristic algorithm. Section 5 contains the computational results, and some concluding remarks are made in Section 6.

We assume throughout that the reader is familiar with the basics of

integer programming [36]. We also assume, without loss of generality, that the processing times are non-negative integers.

2 Literature Review

Since the literature on flowshop scheduling is vast, we cover here only works of direct relevance. Subsection 2.1 recalls a MIP formulation of the PFM. Subsections 2.2 and 2.3 cover constructive heuristics and local search, and Subsection 2.4 covers matheuristics. For more on flowshop scheduling in general, see the book [6].

2.1 The Stafford *et al.* formulation

Stafford *et al.* [27] surveyed MIP formulations of the PFM, and proposed a new one that proved to be quite effective. The new formulation uses the following variables:

- x_{jk} $(j, k = 1, \ldots, n)$: 1 if job j is assigned to the k-th position in the sequence and 0 otherwise.
- f_{ik} $(i = 1 \ldots, m; k = 1, \ldots, n)$: the time at which machine i finishes processing the k-th job in the sequence.

The formulation is then:

$$\begin{aligned}
\min \quad & f_{mn} && && (1)\\
\text{s.t.} \quad & \textstyle\sum_{k=1}^{n} x_{jk} = 1 && (j = 1, \ldots, n) && (2)\\
& \textstyle\sum_{j=1}^{n} x_{jk} = 1 && (k = 1, \ldots, n) && (3)\\
& f_{11} = \textstyle\sum_{j=1}^{n} p_{1j} x_{j1} && && (4)\\
& f_{i,k+1} \geq f_{ik} + \textstyle\sum_{j=1}^{n} p_{ij} x_{j,k+1} && (i = 1, \ldots, m;\ k = 1, \ldots, n-1) && (5)\\
& f_{i+1,k} \geq f_{ik} + \textstyle\sum_{j=1}^{n} p_{i+1,j} x_{jk} && (i = 1, \ldots, m-1;\ k = 1, \ldots, n) && (6)\\
& x_{jk} \in \{0, 1\} && (i = 1, \ldots, m;\ k = 1, \ldots, n) && (7)\\
& f_{ik} \geq 0 && (i = 1, \ldots, m;\ k = 1, \ldots, n). && (8)
\end{aligned}$$

The constraints (2) and (3) are standard assignment constraints. The constraint (4) states that the finishing time of the first job on the first machine is equal to the processing time of that job. The constraints (5), (6), and (7) ensure that the other finishing times take the correct values.

Stafford *et al.* reported that they could solve instances with up to 40 jobs and 5 machines, in reasonable computing times, with this formulation. This accords with our own experience.

2.2 Constructive heuristics

The earliest heuristics for the PFM were 'constructive', in the sense that solutions were iteratively built from scratch. For detailed reviews of these heuristics, up to 2004 or so, see [10, 26]. It is generally agreed that, among those heuristics, the 'NEH' heuristic by Nawaz *et al.* [21] is the winner.

NEH can be described in the following steps:

- Sorting phase: Jobs are sorted in a non-increasing order based on their total processing time on m machines;
- Initialisation: The first job is used to initialise a 'partial' sequence;
- Insertion phase: The remaining jobs are then iteratively inserted into the partial sequence, each being placed in the position that minimises the partial makespan at that stage.

If implemented in a naive manner, NEH runs in $O(n^3m)$ time. Taillard [30] showed that this can be reduced to $O(n^2m)$ using appropriate data structures.

Since the publication of [21], a large number of improvements have been proposed to the NEH heuristic, with a comprehensive efficiency comparison between popular ones available, e.g., in [4, 25, 8]. For brevity, we do not go into details.

2.3 Meta-heuristics based on local search

As one might expect, local search has been applied extensively to the PFM. The following two neighbourhoods have proved particularly popular (see, e.g., [10, 26]):

- 'swap' or 'interchange', which exchanges the positions of two jobs in the current sequence;
- 'shift' or 'insertion', which selects a job and changes its position in the sequence;

Note that the 'swap' neighbourhood is a subset of the 'shift' neighbourhood. Moreover, the argument of Taillard [30], mentioned in the previous subsection, shows that one can find the best insertion position for any given job in $O(nm)$ time. For these reasons, 'shift' is normally preferred to 'swap'. Indeed, the shift neighbourhood has been incorporated into both simulated annealing and tabu search metaheuristics for the PFM [13, 22].

Some more complicated neighbourhoods include 'block moves', which relocate an entire block of consecutive jobs to another position [15, 32], and 'general shift', which involves removing a specific number of jobs from the

sequence and then inserting them one at a time in the best position [5, 7]. For brevity, we do not go into details.

Existing local-search metaheuristics in the PFM literature often rely on evaluating a large number of candidate moves, which can be time-consuming. Conversely, evaluating only a small number of moves may fail to fully exploit the neighbourhood. In this study, we demonstrate how to use small-sized MIP formulations to effectively and extensively explore the neighbourhood.

2.4 Matheuristics

In recent years, there has been increasing interest in heuristics that employ mathematical programming techniques, such as branch-and-bound, dynamic programming or integer programming. These hybrid solution methods have come to be known as *matheuristics* [19].

We are aware of only one matheuristic for the PFM: the truncated branch-and-bound algorithm of Haouari & Ladhari [15]. Matheuristics have however been proposed for flowshop scheduling with other objectives, such as minimising tardiness [29] or minimising the total completion time [3]. Matheuristics have also been proposed for the no-wait variant of the PFM, the PFM with batch processing machines, and other scheduling problems such as job shop scheduling [1]. We focus on the PFM itself, introducing simple yet effective matheuristic approaches for the problem.

3 Five MIP-Based Neighbourhoods

In this section, we present five new neighbourhoods for the PFM. They are described in Subsection 3.1 to 3.5. Each neighbourhood can be explored by solving a restricted version of the Stafford *et al.* MIP (1)-(8), in which the majority of the x variables have been fixed to either zero or one.

Throughout this section, we assume that we already have an initial feasible PFM solution that we hope to improve. Such a solution can be created, for example, by using the NEH heuristic. It is then converted into a feasible solution for the Stafford *et al.* MIP. We denote this solution by $(\bar{x}, \bar{f})$.

3.1 Neighbourhood 1: position block

Let us call a set of consecutive positions in the sequence a *position block*. For example, if $n = 20$, then $\{4, 5, \ldots, 8\}$ is a position block of size 5, starting at position 4 (i.e., positions 4 through 8 in order). A simple neighbourhood for the PFM is obtained as follows: select a position block of a given size, find out which jobs currently occupy the positions in that block, and then allow those jobs to change their position while the positions of the other jobs remain fixed. To give each job a chance of being moved, one can select a block of fixed length at random. For instance, if the current job sequence

is $(j_1, j_2, \ldots, j_{20})$ and the selected block is $\{4, 5, 6, 7, 8\}$, then the neighbourhood consists of all permutations of the jobs $(j_4, j_5, j_6, j_7, j_8)$ within those positions.

A disadvantage of the given neighbourhood is that the jobs in the first few positions and the last few positions have a low probability of being able to move. To address this, we use a more general kind of position block, in which we use arithmetic modulo n to permit blocks to wrap around from position n back to position 1. For example, if $n = 20$ and we seek a block of size 5, then we permit a block like $\{18, 19, 20, 1, 2\}$.

The details are given in Algorithm 1. The input parameter `block_size` represents the cardinality of the blocks, and the random integer t represents the first position in the block. Note that the Stafford *et al.* MIP is regarded as part of the input. In other words, we assume that the MIP is already stored in the computer's memory.

Algorithm 1: Position-block neighbourhood

input : number of machines m, number of jobs n,
integer `block_size` (between 2 and n),
feasible solution $(\bar{x}, \bar{f})$, Stafford *et al.* MIP

Let t be a random integer between 1 and n;
if $t + \texttt{block\_size} - 1 \leq n$ **then**
  for $k \in \{1, \ldots, t-1\} \cup \{t + \texttt{block\_size}, \ldots, n\}$ **do**
    for $j = 1, \ldots, n$ **do**
      Add the equation $x_{jk} = \bar{x}_{jk}$ to the MIP;
    end
  end
else
  for $k \in \{t + \texttt{block\_size} - n, \ldots, t-1\}$ **do**
    for $j = 1, \ldots, n$ **do**
      Add the equation $x_{jk} = \bar{x}_{jk}$ to the MIP;
    end
  end
end
Solve the restricted MIP by cut-and-branch;
Let (x^*, f^*) be the optimal solution to the restricted MIP;
Restore the MIP to its original form;

output: New MIP solution (x^*, f^*)

We remark that the number of x variables that are free in the restricted MIP is n times `block_size`. Thus, as long as `block_size` is significantly smaller than n, the restricted MIP is likely to be much easier to solve than the original MIP. Larger values of `block_size` will lead to an increased probability of finding an improved solution, but this will come at the cost of a higher running time.

3.2 Neighbourhood 2: generalised swap

Our second neighbourhood is similar to the first, but we no longer require positions to be consecutive. We choose an integer parameter `set_size` that lies between 2 and n. We then select a set of positions, say S, of cardinality `set_size`. We then allow the jobs whose positions are currently in S to change their positions while forcing the other jobs to remain in their current position.

For example, if $n = 20$ and `set_size` $= 4$, we might choose $S = \{2, 7, 14, 18\}$. If these positions hold jobs $(j_2, j_7, j_{14}, j_{18})$, the neighbourhood includes all permutations of these jobs across the selected positions.

In order to give every job a chance of being moved, we select the set S at random. The details are given in Algorithm 2. We call this neighbourhood the 'generalised swap' neighbourhood because it reduces to the classical swap neighbourhood when $|S| = 2$.

Algorithm 2: Generalised swap neighbourhood

input : number of machines m, number of jobs n,
integer `set_size` (between 2 and n),
feasible solution $(\bar{x}, \bar{f})$, Stafford *et al.* MIP

```
Let S be a random subset of {1,...,n} of cardinality set_size.;
for k ∈ {1,...,n} \ S do
    for j = 1,...,n do
        Add the equation x_jk = x̄_jk to the MIP;
    end
end
Solve the restricted MIP by cut-and-branch;
Let (x*, f*) be the optimal solution to the restricted MIP;
Restore the MIP to its original form;
```

output: New MIP solution (x^*, f^*)

Note that, for this neighbourhood, the number of x variables that are free in the restricted MIP is n times `set_size`.

3.3 Neighbourhood 3: delta-shake

In our third neighbourhood, which we call the 'delta-shake' neighbourhood, we choose a positive integer parameter δ, and then permit each job to change its position by no more than δ.

As in the position-blocks neighbourhood, we apply arithmetic modulo n to the jobs that are in the first few positions or in the last few positions. For example, if $n = 20$ and $\delta = 3$, then the job that is currently in position 2 will be permitted to move to positions 19 and 20 as well as positions 1, 3, 4 and 5 (or it could stay in position 2). This ensures that every job has exactly $2\delta + 1$ possible positions. See Algorithm 3 for details.

Algorithm 3: Delta-shake neighbourhood

input : number of machines m, number of jobs n,
integer δ (between 1 and $\lfloor n/2 \rfloor$),
feasible solution $(\bar{x}, \bar{f})$, Stafford *et al.* MIP,
current positions $P[j]$ for each job $j = 1, \ldots, n$

for $j = 1, \ldots, n$ **do**
  if $P[j] + \delta > n$ **then**
    for $k \in \{P[j] + \delta - n + 1, \ldots, P[j] - \delta - 1\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  else if $P[j] - \delta \leq 0$ **then**
    for $k \in \{P[j] + \delta + 1, \ldots, P[j] - \delta + n - 1\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  else
    for $k \in \{1, \ldots, P[j] - \delta - 1\} \cup \{P[j] + \delta + 1, \ldots, n\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  end
end
Solve the restricted MIP by cut-and-branch;
Let (x^*, f^*) be the optimal solution to the restricted MIP;
Restore the MIP to its original form;

output: New MIP solution (x^*, f^*)

Note that, for this neighbourhood, the number of x variables that are free in the restricted MIP is exactly $n(2\delta+1)$. Note also that this neighbourhood is *deterministic*, whereas the previous two neighbourhoods are randomised.

3.4 Neighbourhood 4: randomised delta-shake

A disadvantage of the delta-shake neighbourhood is that most jobs can only change their position by a small amount. It is conceivable however that some jobs might need to move further than others. For this reason, we consider a randomised version of delta-shake, in which the amount by which a job may move is itself random.

Note that we allow each job j to change its position by a maximum of Δ_j, where Δ_j is a random number between 1 and $2\delta - 1$. This means that, for any given job, the expected value of Δ_j is δ. This, in turn ensures that the expected number of x variables that are free in the restricted MIP is $n(2\delta + 1)$. This is the same as the number of x variables that are free in the delta-shake neighbourhood. See Algorithm 4 for details.

For example, suppose $n = 20$ and $\delta = 3$. The allowed displacement is sampled between 1 and 5 for each job in the selected positions. If the sampled value Δ_8 for job j_8 is 5, then it may move to any position from 3

Algorithm 4: Randomised delta-shake neighbourhood

input : number of machines m, number of jobs n,
positive integer δ (between 1 and $\lfloor n/4 \rfloor$),
feasible solution $(\bar{x}, \bar{f})$ for the Stafford *et al.* MIP,
current positions $P[j]$ for each job $j = 1, \ldots, n$

for $j = 1, \ldots, n$ **do**
  Let Δ_j be a random number between 1 and $2\delta - 1$;
  if $P[j] + \Delta_j > n$ **then**
    for $k \in \{P[j] + \Delta_j - n + 1, \ldots, P[j] - \Delta_j - 1\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  else if $P[j] - \Delta_j \leq 0$ **then**
    for $k \in \{P[j] + \Delta_j + 1, \ldots, P[j] - \Delta_j + n - 1\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  else
    for $k \in \{1, \ldots, P[j] - \Delta_j - 1\} \cup \{P[j] + \Delta_j + 1, \ldots, n\}$ **do**
      Add the equation $x_{jk} = 0$ to the MIP;
    end
  end
end
Solve the restricted MIP by cut-and-branch;
Let (x^*, f^*) be the optimal solution to the restricted MIP;
Restore the MIP to its original form;

output: New MIP solution (x^*, f^*)

to 13.

3.5 Neighbourhood 5: extended shift

Our fifth and final neighbourhood is called the 'extended shift' neighbourhood. We pick an integer parameter `set_size` and select a random set S of jobs of cardinality `set_size`. We then allow the jobs in S to be placed in any desired position while maintaining the order of the jobs that are not in S.

Suppose, for example, that $n = 8$, `set_size` $= 2$ and $S = \{4, 6\}$. Suppose also that the initial sequence is $(3, 1, 8, 5, 7, 4, 2, 6)$. A possible move would be to move job 4 to the 3rd position and move job 6 to the 4th position. The resulting sequence would be $(3, 1, 4, 6, 8, 5, 7, 2)$.

Note that, in this example, we had to change the position of some of the jobs that were *not* in S. In particular, we had to move jobs 8, 5 and 7 two steps later in the sequence, and move job 2 one step later. We ensure that our restricted MIP is flexible enough to handle this possibility. See Algorithm 5 for details.

One can check that the new neighbourhood is a generalisation, not only

Algorithm 5: Extended shift neighbourhood

input : number of machines m, number of jobs n,
integer parameter `set_size` (between 1 and n),
feasible MIP solution $(\bar{x}, \bar{f})$, Stafford *et al.* MIP,
current positions $P[j]$ for each job $j = 1, \ldots, n$

Let S be a random subset of $\{1, \ldots, n\}$ of cardinality `set_size`;
Set T to $\{P[j] : j \in S\}$;
for $j \in \{1, \ldots, n\} \setminus S$ **do**
    Set h to $|\{i \in T | i < P[j]\}|$;
    for $k \in \{1, \ldots, (P[j] - h - 1)\} \cup \{(P[j] + \texttt{set\_size} - h + 1), \ldots, n\}$ **do**
        Add the equation $x_{jk} = 0$ to the MIP;
    end
end
Solve the restricted MIP by cut-and-branch;
Let (x^*, f^*) be the optimal solution to the restricted MIP;
Restore the MIP to its original form;

output: New MIP solution (x^*, f^*)

of the 'shift' neighbourhood, but also of the neighbourhoods mentioned in 3.1 and 3.2. Moreover, if `set_size` is even, then the new neighbourhood contains the delta-shake neighbourhood with $\delta = (\texttt{set\_size}/2 + 1)$.

Note that, for this neighbourhood, the number of x variables that are free in the restricted MIP is exactly:

$$n \times \texttt{set\_size} + (n - \texttt{set\_size}) \times (\texttt{set\_size} + 1).$$

This is due to the fact that the jobs in S can occupy any of the n positions, while each of the remaining jobs has $\texttt{set\_size} + 1$ potential positions.

4 A Matheuristic for the PFM

In this section, we present a matheuristic for the PFM which uses the neighbourhoods that were presented in the previous section. Subsection 4.1 gives an overview of the algorithm in its basic form, and Subsection 4.2 describes a variant of the algorithm. Subsection 4.3 provides some implementation details.

4.1 Overview of the matheuristic

Algorithm 6 outlines our matheuristic. Before going into the details of the algorithm, we make some remarks about the inputs to the algorithm.

One input is the 'MIP-based neighbourhood type'. By this we mean one of the five neighbourhoods mentioned in the previous section. Another input is 'MIP-based neighbourhood size'. By this we mean `block_size` for

position blocks, `set_size` for generalised swap, δ for delta-shake, Δ for randomised delta-shake, and `set_size` for extended shift. Finally, the parameter '`max_tries`' is used to give the randomised MIP-based neighbourhoods more than one chance to find an improved feasible solution. (For the delta-shake neighbourhood, which is deterministic, one should set `max_tries` to 1.)

Algorithm 6: Matheuristic for the PFM

input : number of machines m, number of jobs n, processing times p_{ij}, MIP-based neighbourhood type, MIP-based neighbourhood size, positive integer parameter `max_tries`

```
Run the NEH heuristic to get initial PFM solution;
Set up the Stafford et al. MIP;
Set improved to true;
while improved is true do
    Set improved to false;
    Apply the shift procedure;
    if the solution has been improved then
        Set improved to true;
    end
    if improved is false then
        Set counter to 1;
        while counter ≤ max_tries and improved is false do
            for j = 1, ..., n do
                Let P[j] be the current position of job j;
            end
            Modify the MIP according to the given neighbourhood;
            Solve the modified MIP by cut-and-branch;
            if the solution has not been improved then
                Increment counter;
            else
                Update (x̄, f̄);
                Set improved to true;
            end
        end
    end
end
```

output: PFM solution $(\bar{x}, \bar{f})$ and upper bound $\bar{f}_{mn}$

Now we discuss the algorithm itself. First, the NEH heuristic is called to generate an initial feasible solution. After that, a local search procedure is applied based on the shift neighbourhood. When that procedure is unable to improve the solution further, a more complex local search procedure is applied, based on one of our five MIP-based neighbourhoods. If the chosen MIP-based neighbourhood manages to find an improved solution, we revert to using the shift neighbourhood again. Otherwise, the process terminates.

The reason that we run NEH and shift first is that they generally produce a reasonably good PFM solution quickly [22, 30]. This tends to reduce the total number of MIPs that are solved, which in turn reduces the overall computing time. The desire to reduce the number of MIPs is also why we call shift whenever the MIP-based procedure finds an improved solution.

4.2 A variant of the matheuristic

In our preliminary computational experiments, we experimented with several variants of Algorithm 6, but none of them gave significantly better results. For brevity, we do not describe all variants, but there is one that was reasonably promising, which can be found in Algorithm 7. We call this variant 'dynamic', since it adjusts the size of the given MIP-based neighbourhood in a dynamic manner.

Algorithm 7: Dynamic local search procedure

input : current PFM solution $(\bar{x}, \bar{f})$, MIP-based neighbourhood type, positive integer parameters `min_size`, `max_size`

```
Run NEH and set up the MIP as in Algorithm 6;
Set improved to true;
while improved is true do
    Set improved to false;
    Apply the shift procedure;
    if the solution has been improved then
        Set improved to true;
    end
    if improved is false then
        Set θ to min_size;
        while θ ≤ max_size and improved is false do
            for j = 1, ..., n do
                Let P[j] be the current position of job j;
            end
            Modify the MIP so that the given MIP-based neighbourhood
             has size θ;
            Solve the modified MIP by cut-and-branch;
            if the solution has not been improved then
                Set θ to θ + 1;
            else
                Set improved to true;
                Update (x̄, f̄);
            end
        end
    end
end
```

output: PFM solution $(\bar{x}, \bar{f})$ and upper bound $\bar{f}_{mn}$

In Algorithm 7, the input parameters `min_size` and `max_size` determine the minimum and maximum sizes of the MIP-based neighbourhood, and the variable θ determines the size of that neighbourhood in each major iteration. As before, the 'size' of the neighbourhood means `block_size` for position blocks, `set_size` for generalised swap, and so on.

4.3 Implementation details

To solve the restricted MIPs, we use the 'Mixed Integer Optimizer' of IBM ILOG CPLEX v. 22.1.1. As far as we can tell, the solver uses 'cut-and-branch' by default (i.e., cutting planes are generated at the root node of the branch-and-bound tree). In our preliminary experiments, we found that the restricted MIPs could usually be solved very quickly, but they were sometimes rather challenging for the larger PFM instances. To alleviate this problem, we use several 'implementation tricks':

- Whenever we feed a reduced MIP to the MIP solver, we provide the current incumbent solution to the solver as well. This gives the solver a strong initial upper bound, which typically leads to a smaller branch-and-bound tree. It also gives the solver a good initial primal solution, which it may be able to improve using its own internal heuristics.
- Solvers such as `CPLEX` and `Gurobi` have a parameter which enables one to control the trade-off between finding good primal solutions and proving optimality. The parameter is set to HIDDENFEAS, which directs the MIP optimiser to prioritise the identification of high-quality feasible solutions that are typically challenging to find.
- Such solvers also enable one to abort the cut-and-branch run after a certain number of improved primal solutions has been found. We set this number to 2. (Note that, if we set it to 1 instead, we would be following a 'first-improvement' strategy, which stops searching the neighbourhood as soon as an improved solution is encountered.)
- We impose a run-time limit of 900 seconds for solving each reduced MIP.

We found that, with these improvements, we were able to reduce the MIP solution times so much that we could set the parameter `max_tries` as large as 10, while still keeping the total computing times reasonable.

5 Computational Results

In this section, we present the results of some computational experiments that we conducted using the benchmark PFM instances of Taillard [31] and Vallada *et al.* [34]. The instance data, along with our detailed results, will

be made available at the Lancaster University Data Repository, under the heading 'Permutation Flowshop Problem MIP Local Search'[1]. The implementation code is also available in the GitHub repository[2]

The MIP-based local search was implemented in Python 3.10 and run on a heterogeneous cluster[3] on nodes with 16 processing cores and 64 GB of RAM. The MIP neighbourhoods were solved using the IBM ILOG CPLEX 22.1.1 solver and the Decision Optimization CPLEX Modeling (`DOCplex`) library for Python. We set the parallel feature in the MIP optimiser to use up to 16 threads when running on the nodes.

Now, recall that four of our five neighbourhoods involve randomness. This means that the final PFM solution obtained can vary from one run of the matheuristic to another. So, for each of the randomised neighbourhoods and each test instance, we run our matheuristic ten times, with different random seed numbers each time. We then record the best makespan and the time spent to obtain this makespan over the ten solutions found for each instance.

Interestingly, we found that the obtained solution could vary from one run to another even when we used the delta-shake neighbourhood, which is deterministic. As far as we can tell, this variation was caused by the fact that our MIP solver was running several parallel threads in asynchronous mode, meaning that slight variations in speed could lead to different improved solutions being found when the reduced MIPs were being solved. To address this, we perform ten runs even for the delta-shake neighbourhood.

5.1 Parameter settings

For a fair comparison between our five MIP-based neighbourhoods, we ensure that the reduced MIP sizes are as similar as possible. So, once we have chosen a specific value of δ for the delta-shake neighbourhood or its randomised version, we set the parameters `block_size` and `set_size` to $2\delta + 1$ in the position-block and generalised swap neighbourhoods, respectively.

Obtaining a fair comparison with the extended shift neighbourhood is a bit more tricky. With a bit of work, it can be shown that one would need to set the parameter `set_size` to around $(n - 1/2) - \sqrt{(n - 1/2)^2 - 2\delta n}$. Fortunately, this expression tends to δ when n approaches ∞. So we just set the parameter to δ, for simplicity. We have performed some tests to define the size of δ in Section 5.2.

[1] `http://www.research.lancs.ac.uk/portal/en/datasets/search.html`
[2] `https://github.com/scaceresg/pfm-mip-based-local-search`
[3] `https://lancaster-hec.readthedocs.io/en/latest/index.html`

5.2 Size parameter tests

To identify appropriate parameter values for our neighbourhoods, we tested various settings on a subset of the Taillard instances [31]. The Taillard instances have $n \in \{20, 50, 100, 200, 500\}$ and $m \in \{5, 10, 20\}$. There are ten instances for each combination of n and m with $n \in \{20, 50, 100\}$ and $m \in \{5, 10, 20\}$. There are also ten instances for each of the following combinations of n and m: $(200, 10)$, $(200, 20)$ and $(500, 10)$. This makes 120 instances in total. The optimal values for the instances with $m \in \{5, 10\}$ can be found in [31]. At the time of writing, the best-known lower and upper bounds for the other instances were available on Taillard's personal website.[4].

For this preliminary test, we used all small-scale instances with $m = 5$, one medium-scale instance with $(m, n) = (10, 50)$, and three large instances with $(m, n) \in \{(20, 100), (20, 200), (20, 500)\}$. We tested the two following approaches:

- The *'static'* approach, where the size parameter remains constant throughout the algorithm. We tested the parameter values in the set $\{10, 12, \lceil n/10 + m/10 \rceil\}$.
- The *'dynamic'* approach, where the size parameter is dynamically adjusted throughout the search, as described in Algorithm 7. We tested $(\texttt{min\_size}, \texttt{max\_size})$ from the set $\{(5, 10), (6, 12)\}$.

For each neighbourhood and each parameter test setting, we present the average results across several instances (denoted by '#') selected for each combination of m and n. Table 1 shows the average percentage gaps between our upper bounds and the best-known ones, while Table 2 reports the average computing times in seconds.

The results show that our matheuristic, using the position block, generalised swap and randomised delta neighbourhoods, achieved significantly better performance in terms of average percentage gaps when using static approaches with $\delta \in \{10, 12\}$, compared to the static approach with $\delta = \lceil n/10 + m/10 \rceil$ and the dynamic approaches. However, this outperformance comes with slower running times. A closer look at the output suggests that this is due to an increased chance of finding improved solutions, which leads to a higher number of executed 'while' loops in our matheuristic.

The dynamic approaches perform significantly better than the static approaches when applied to delta-shake, in terms of solution quality. However, among all neighbourhoods, delta-shake produces the worst solution quality, especially for larger instances. With static approaches, delta-shake rarely finds improved solutions, leading to very short computing times. In contrast, the dynamic approaches result in significantly longer running times.

[4] `http://mistic.heig-vd.ch/taillard/`(accessed 10/02/25)

This is because the dynamic approaches do not immediately terminate when delta-shake fails to find an improvement, as static approaches do. Instead, they continue attempting up to (`max_size` - `min_size`) times, increasing the size parameter by 1 after each failed attempt and only exiting after all these attempts fail to improve the solution.

For the extended shift neighbourhood, the static and dynamic approaches perform similarly. However, the dynamic approach with $\delta \in [5, 10]$ yields slightly better solutions than the static one while also requiring less computing time.

Based on these observations, static approaches are preferable because they can find better solutions with the two most effective neighbourhoods in this preliminary test: position block and generalised swap.

Among the static approaches, the setting $\delta = \lceil n/10 + m/10 \rceil$ is clearly less effective compared to other values of δ. For the two largest instances, increasing δ from 10 to 22 or 52 does not lead to better solutions. This suggests that the search space is too narrow when δ is set to only around 10% of the number of jobs. Both $\delta = 10$ and $\delta = 12$ perform similarly in terms of solution quality, but we prefer $\delta = 10$ because it results in shorter computing times for the position block and generalised swap neighbourhoods. Therefore, for the rest of the experiments, we used the static approach with $\delta = 10$.

5.3 Taillard instances

As mentioned in the previous subsection, we used the static approach with $\delta = 10$ for all neighbourhoods. However, for instances with $n = 20$, we set $\delta = 9$ because in the position block, generalised swap, and delta-shake neighbourhoods, each job has exactly $2\delta + 1$ possible positions to move.

Similar to the preliminary tests, we present the results in two tables: Table 3 shows the average percentage gaps of upper bounds, while Table 4 presents the average computing times in seconds. Both tables follow the same structure as Table 1 and 2, with the exception that the size parameter δ is fixed to 10 and the results from a simple 'NEH and shift' approach (our matheuristic without implementing any neighbourhoods) are also presented. These results serve as the baseline for measuring the improvement potential of each MIP-based neighbourhood, with the average improvements shown in the last row.

It is clear that our MIP-based neighbourhoods generally improve the initial solutions obtained with the simple NEH and shift approaches. For small-scale instances ($m = 5$), delta-shake and randomised delta help to close the optimum gap from the NEH and shift approach. The other neighbourhoods also effectively reduce the initial optimum gap by more than 50% for all instances. For medium-scale instances ($m = 10$), the extended shift neighbourhood performs exceptionally well, reducing the average initial op-

Table 1: Average percentage gaps for the size parameter tests

	δ	m	n	#	Position Block	Generalised Swap	Delta-shake	Randomised Delta	Extended Shift
		5	20	10	0.00	0.00	0.00	0.00	0.00
		5	50	10	0.01	0.01	0.00	0.00	0.00
		5	100	10	0.00	0.11	0.00	0.00	0.03
Static	10	10	50	1	1.50	1.10	4.50	1.50	1.10
		20	100	1	3.80	4.70	6.90	6.90	5.60
		20	200	1	2.60	3.10	4.00	4.00	4.00
		20	500	1	1.70	1.70	1.70	1.70	1.50
		average:			1.37	1.53	2.44	2.01	1.75
		5	20	10	0.00	0.00	0.00	0.00	0.00
		5	50	10	0.00	0.01	0.00	0.00	0.00
		5	100	10	0.00	0.10	0.00	0.00	0.01
Static	12	10	50	1	1.10	1.10	4.50	1.10	1.10
		20	100	1	4.30	4.80	6.90	6.90	5.90
		20	200	1	2.50	2.90	4.00	4.00	4.00
		20	500	1	1.70	1.70	1.70	1.70	1.70
		average:			1.37	1.52	2.44	1.96	1.82
	3	5	20	10	1.51	1.46	0.73	0.28	0.37
	6	5	50	10	0.05	0.21	0.03	0.00	0.01
	11	5	100	10	0.01	0.08	0.00	0.00	0.05
Static	6	10	50	1	1.80	4.50	4.50	1.10	1.10
	12	20	100	1	4.00	4.80	6.90	6.90	5.80
	22	20	200	1	3.00	3.50	4.00	4.00	4.00
	52	20	500	1	1.70	1.70	1.70	1.70	1.70
		average:			1.72	2.32	2.55	2.00	1.86
		5	20	10	0.00	0.00	0.00	0.00	0.00
		5	50	10	0.02	0.09	0.00	0.00	0.01
		5	100	10	0.10	0.19	0.00	0.00	0.04
Dynamic	[5,10]	10	50	1	2.10	1.90	2.80	1.70	1.10
		20	100	1	4.50	5.30	6.90	6.90	5.70
		20	200	1	3.00	3.50	4.00	4.00	3.70
		20	500	1	1.70	1.70	1.70	1.70	1.60
		average:			1.63	1.81	2.20	2.04	1.74
		5	20	10	0.00	0.00	0.00	0.00	0.00
		5	50	10	0.00	0.02	0.00	0.00	0.00
		5	100	10	0.01	0.16	0.00	0.00	0.02
Dynamic	[6,12]	10	50	1	1.30	1.50	2.80	2.80	1.30
		20	100	1	5.10	5.40	6.90	6.90	5.30
		20	200	1	3.30	2.90	4.00	4.00	3.90
		20	500	1	1.70	1.70	1.70	1.70	1.60
		average:			1.63	1.67	2.20	2.20	1.73

timum gap from 2.26% to 0.3% on average. Following closely are the position block and randomised delta neighbourhoods, which on average reduce this

Table 2: Average times in seconds for the size parameter tests

	δ	m	n	#	Position Block	Generalised Swap	Delta-shake	Randomised Delta	Extended Shift
		5	20	10	9.14	8.55	1.82	6.38	4.52
		5	50	10	3.60	4.21	3.15	15.61	8.22
		5	100	10	8.50	5.81	15.73	58.24	21.60
Static	10	10	50	1	246.13	1550.89	901.01	27322.35	16693.80
		20	100	1	7528.29	15459.91	908.11	9009.77	12376.09
		20	200	1	1342.50	2127.67	923.93	9030.70	9031.18
		20	500	1	546.66	2377.04	1323.43	9496.80	13318.70
		average:			1383.55	3076.30	582.45	7848.55	7350.59
		5	20	10	8.48	8.62	1.69	6.24	4.61
		5	50	10	4.92	5.76	5.75	18.48	8.61
		5	100	10	8.42	7.60	18.20	110.74	35.33
Static	12	10	50	1	486.81	3237.54	901.22	16725.41	7204.12
		20	100	1	6760.31	18653.28	909.85	9011.78	17696.60
		20	200	1	3650.75	5633.67	928.07	9035.45	9035.76
		20	500	1	638.26	6657.49	1402.04	9575.81	9564.39
		average:			1651.14	4886.28	595.26	6354.84	6221.34
	3	5	20	10	0.34	0.46	0.58	1.95	1.34
	6	5	50	10	2.03	1.85	3.75	13.45	7.69
	11	5	100	10	7.94	5.41	15.21	69.97	18.02
Static	6	10	50	1	14.66	2.66	901.19	8802.32	1905.16
	12	20	100	1	4476.27	18992.88	909.48	9011.49	9770.88
	22	20	200	1	4691.35	6140.14	928.05	9036.35	9036.10
	52	20	500	1	669.43	9444.65	1316.81	9489.91	9489.03
		average:			1408.86	4941.15	582.15	5203.63	4318.32
		5	20	10	2.28	2.25	4.47	3.26	2.06
		5	50	10	2.49	2.17	11.95	9.93	5.04
		5	100	10	4.31	3.73	40.22	30.78	13.01
Dynamic	[5,10]	10	50	1	54.54	148.95	8178.51	8664.52	4758.25
		20	100	1	130.46	1108.81	5409.41	5409.39	8869.94
		20	200	1	108.06	1780.24	5428.32	5428.18	8776.51
		20	500	1	592.03	876.47	5953.07	5949.94	7087.62
		average:			127.74	560.37	3575.14	3642.28	4216.06
		5	20	10	2.50	2.36	4.69	3.18	2.29
		5	50	10	2.84	4.17	14.88	13.05	7.29
		5	100	10	7.07	5.01	61.36	45.06	16.28
Dynamic	[6,12]	10	50	1	196.71	1036.79	17043.65	14659.82	20288.90
		20	100	1	545.54	3876.76	6310.71	6310.78	14145.35
		20	200	1	75.52	3805.69	6333.02	6334.01	7780.33
		20	500	1	627.39	1621.97	6853.96	6850.22	12525.72
		average:			208.22	1478.96	5231.75	4888.02	7823.74

gap to 0.39% and 0.43%, respectively. Unsurprisingly, delta-shake is the least effective neighbourhood in improving the initial solution, due to its deterministic nature. For large-scale instances, the position block performs

Table 3: Average percentage gaps of upper bounds for Taillard instances

m	n	NEH + Shift	Position Block	Generalised Swap	Delta-shake	Randomised Delta	Extended Shift
5	20	2.00	0.00	0.00	0.00	0.00	0.00
	50	0.56	0.01	0.01	0.00	0.00	0.00
	100	0.29	0.00	0.11	0.00	0.00	0.03
	avg:	0.95	0.00	0.04	0.00	0.00	0.01
10	20	2.76	0.00	0.00	0.09	0.00	0.00
	50	3.76	0.65	1.04	2.07	0.79	0.50
	100	1.59	0.45	0.84	1.03	0.43	0.25
	200	0.91	0.46	0.59	0.77	0.51	0.46
	avg:	2.26	0.39	0.62	0.99	0.43	0.30
20	20	2.48	0.13	0.15	0.55	0.12	0.03
	50	6.25	2.66	4.05	5.93	5.16	5.14
	100	4.96	2.92	3.85	4.96	4.84	4.52
	200	3.65	2.73	3.02	3.65	3.59	3.43
	500	1.66	1.43	1.39	1.66	1.66	1.65
	avg:	3.80	1.97	2.49	3.35	3.07	2.95
	avg:	2.34	0.79	1.05	1.45	1.17	1.09
	% improvement:		66.2	55.0	38.0	49.9	53.4

the best among the neighbourhoods, reducing the average initial optimum gap by nearly half. Generalised swap follows closely behind, surprisingly outperforming extended shift in terms of solution improvement on average. This could be due to the fact that the size parameter in extended shift is set to δ, which is rounded down from its decimal value for a fair comparison, as described in Subsection 5.1. Delta-shake remains the least effective neighbourhood, failing to improve initial solutions from the NEH and shift approach for instances with at least 100 jobs. This is because the size of $\delta = 10$ is too small relative to the number of jobs, limiting the search space and preventing the identification of improved solutions.

Regarding computing times, one can see that the delta-shake neighbourhood runs the fastest. This result can be explained by our static parameter setting and its deterministic nature, where the `max_tries` parameter is set to 1. This means that our matheuristic terminates if it fails to find an improved solution, without making further attempts as other neighbourhoods do. Among the non-deterministic neighbourhoods, position block achieved the best solution gaps in the shortest time on average, followed by generalised swap. These two neighbourhoods were slower than the others for instances with $n = 20$, but were much faster when $n \geq 50$. A closer look at the results shows that position block and generalised swap make signif-

icant improvements early on, which requires more time to solve each MIP, but quickly reach a point where no further improvements can be found. In contrast, randomised delta and extended shift steadily improve the solution with each iteration, requiring more 'while' loops to continue improving the solution.

Table 4: Average times in seconds for Taillard instances

m	n	NEH + Shift	Position Block	Generalised Swap	Delta-shake	Randomised Delta	Extended Shift
5	20	0.03	9.14	8.55	1.82	6.38	4.52
	50	0.17	3.60	4.21	3.15	15.61	8.22
	100	0.60	8.50	5.81	15.73	58.24	21.60
	avg:	0.27	7.08	6.19	6.90	26.74	11.45
10	20	0.07	3215.48	3167.01	915.32	1223.46	318.51
	50	0.57	340.62	1043.68	1423.69	12921.55	10459.62
	100	1.77	121.86	89.75	1221.83	15121.96	8626.43
	200	6.51	105.70	57.77	1079.13	11957.56	8724.78
	avg:	2.23	945.91	1089.56	1159.99	10306.13	7032.34
20	20	0.13	12799.67	13030.00	1278.46	14724.33	10836.84
	50	1.19	21028.28	17816.40	1056.82	14339.75	15492.69
	100	5.80	4206.91	10191.70	910.81	11963.93	15546.65
	200	21.29	1331.64	4401.81	941.52	10402.73	10684.28
	500	158.53	773.86	4001.48	1192.13	9371.75	10017.34
	avg:	37.39	8028.07	9888.28	1075.95	12160.50	12515.56
	avg:	13.30	2993.69	3661.34	747.61	7497.79	6519.78

5.4 Vallada *et al.* instances

Vallada *et al.* [34] gave some evidence that the Taillard instances are relatively easy for their size. They created some instances that were designed to be hard for the exact techniques that existed at the time. Their 'smaller' instances have $n \in \{10, 20, 30, 40, 50, 60\}$ and $m \in \{5, 10, 15, 20\}$, whereas their 'larger' instances have $n \in \{100, 200, 300, 400, 500, 600, 700, 800\}$ and $m \in \{20, 40, 60\}$. There are ten instances for each combination of n and m, making 480 instances in total. Vallada *et al.* [34] also computed upper bounds for these instances.

Similarly to the Taillard instances, we set $\delta = 9$ for the Vallada *et al.* instances when $n = 20$. We also excluded the very small instances with $n = 10$. This left a total of 440 instances for our experiment, with 200 from the smaller set and 240 from the larger set. Given the large number of instances, we chose to use only one neighbourhood. Since position block was the most

effective one in the Taillard instances, we selected it for this experiment. As before, we recorded the best upper bounds and corresponding computational times over 10 runs with different seed values for each instance.

5.4.1 Smaller Vallada *et al.* results

Tables 5 and 6 show the average percentage gaps of upper bounds and the average computing times (in seconds) for the smaller instances. For each combination of n and m, we present results for the solution obtained using the NEH and shift method ('NS') and the solution from our matheuristic with the position block neighbourhood ('PB'), with the percentage improvement provided in the final row.

Table 5: Average % gaps for the smaller Vallada *et al.* instances

	$m = 5$		$m = 10$		$m = 15$		$m = 20$	
n	NS	PB	NS	PB	NS	PB	NS	PB
20	1.01	0.00	3.02	0.02	3.25	0.06	2.53	0.08
30	0.92	0.00	3.88	0.05	4.60	0.28	3.61	0.56
40	0.26	0.00	3.66	0.22	4.60	0.51	3.76	0.96
50	0.26	0.00	2.91	0.59	4.41	0.74	4.15	1.04
60	0.26	0.00	3.12	0.56	4.38	0.87	4.84	1.24
avg:	0.54	0.00	3.32	0.29	4.25	0.49	3.78	0.78
%impr:		100.0		91.3		88.4		79.5

The position block neighbourhood still remains highly effective in improving the initial solutions from the NEH and shift approach for these instances. It closes the gap completely for the instances with $m = 5$, and reduces the average gap by at least 74% across all combinations of m and n. As expected, as the instance size increases, our neighbourhood tends to become less effective, due to our static approach for setting the size parameter in the neighbourhood, and it requires longer running times.

5.4.2 Larger Vallada *et al.* results

Finally, we present the results for the larger Vallada *et al.* instances. Tables 7 and 8 are similar to Tables 5 and 6, respectively.

We encountered some memory problems when solving the MIPs for instances with $m \geq 40$. As a result, we decided to increase the capacity of the nodes to 40 processing cores and 192GB of RAM for these instances.

We see that 'position block' is still capable of improving the solutions found with 'NEH and shift'. However, the improvement is modest, averaging

Table 6: Average times in seconds for the smaller Vallada *et al.* instances

	$m = 5$		$m = 10$		$m = 15$		$m = 20$	
n	NS	PB	NS	PB	NS	PB	NS	PB
20	0.03	21.57	0.08	5702.50	0.10	13050.63	0.15	12586.65
30	0.07	7.22	0.19	5465.54	0.28	16104.80	0.36	17796.82
40	0.14	2.90	0.26	1595.71	0.55	13386.73	0.83	20659.96
50	0.14	2.90	0.63	297.33	1.19	20024.39	1.34	22299.92
60	0.14	2.90	0.86	316.30	1.26	5438.10	2.03	25312.51
avg:	0.11	7.50	0.40	2675.48	0.67	13600.93	0.94	19731.17

Table 7: Average percentage gaps for the larger Vallada *et al.* instances

	$m = 20$		$m = 40$		$m = 60$	
n	NS	PB	NS	PB	NS	PB
100	4.30	1.74	3.96	2.56	4.07	3.01
200	3.45	2.68	3.64	2.34	3.18	2.67
300	2.31	1.70	3.19	2.38	3.07	2.33
400	1.67	1.41	2.78	2.15	2.76	2.38
500	1.49	1.27	2.47	2.30	2.57	2.15
600	1.18	1.04	2.31	2.06	2.37	2.07
700	0.97	0.85	2.07	1.94	2.19	2.09
800	0.79	0.65	1.95	1.84	2.31	2.07
avg:	2.02	1.42	2.80	2.20	2.82	2.35
%impr:		29.8		21.5		16.7

around $16 - 30\%$. As in the case of the smaller Vallada *et al.* instances, the neighbourhood becomes less effective as the instance size increases. This may be due to the static setting of the 'size' parameter, which results in a rather limited search space. Interestingly, the average percentage gap tends to *decrease* as n increases. We do not have a clear explanation for this.

Table 8 shows that our matheuristic runs faster as n increases. A possible explanation for this is that, as n increases, the smaller value of δ relative to n causes the matheuristic to execute fewer 'while' loops. This happens because it fails to find improved solutions after just a few iterations, leading to shorter computating times. However, this phenomenon does not seem to apply as m increases, perhaps because the search space becomes more dependent on the value of δ relative to n.

Table 8: Average times in seconds for the larger Vallada *et al.* instances

	$m = 20$		$m = 40$		$m = 60$	
n	NS	PB	NS	PB	NS	PB
100	6.47	9708.38	16.08	18907.87	13.96	15535.09
200	27.45	875.04	52.98	12714.07	105.42	14475.86
300	51.00	902.48	148.12	5566.40	221.94	14425.30
400	118.02	764.01	286.23	3915.10	447.02	4146.64
500	146.84	676.80	483.62	1959.01	581.57	4514.92
600	247.02	928.31	721.23	2297.74	984.73	3580.87
700	329.05	1097.59	1220.63	1900.77	1481.86	2912.58
800	457.48	1848.90	1264.02	2056.46	1717.92	3639.65
avg:	172.91	2100.19	524.11	6164.68	694.30	7903.86

5.5 Statistical tests

To determine if incorporating the position block neighbourhood into our matheuristic leads to a statistically significant improvement, and to compare its performance against other neighbourhoods on the Taillard and Vallada *et al.* instances, we apply the non-parametric *Wilcoxon signed-rank tests* for paired data. We conduct a one-sided test at a significance level of 0.05. The test is performed using Python's `SciPy wilcoxon` function. We remark that 10 independent replications of each comparison were run using the same set of problem instances.

The hypotheses for the test are as follows:

- *Null hypothesis* H_0: the distribution of differences between pairs is symmetric about zero.
- *Alternative hypothesis* H_a: position block neighbourhood leads to an improvement over other neighbourhoods or improves the performance of the 'NEH+shift' (NS).

Table 9 shows the results of the Wilcoxon signed-rank test on all 120 Taillard instances. The comparison are made in terms of both the percentage optimality gap and the running time metrics. For each pair and each metric, we report the Wilcoxon statistic (W), the corresponding one-sided p-value and an effect size Cliff's δ.

For the %Gap metric, all p-values are well below the significance threshold of 0.05, and all effect sizes are negative, indicating strong statistical evidence that the position block neighbourhood significantly outperforms all other considered neighbourhoods. Notably, the comparison with the 'NEH + shift' returns a W value of 0.0 and an effect size of Cliff's $\delta = -0.57$,

Table 9: Results from the Wilcoxon signed-rank test for Taillard instances

Rules	%Gap			Time		
	W	*p-value*	Cliff's δ	*W*	*p-value*	Cliff's δ
Matheuristic vs. NS	0.0	1.40e-20	-0.57	7260.0	1.00e+00	0.85
PB vs. Generalised Swap	267.5	4.85e-10	-0.10	3036.0	5.99e-02	-0.06
PB vs. Delta-shake	1.5	5.99e-14	-0.17	4287.0	9.57e-01	-0.02
PB vs. Randomised Delta	232.5	2.00e-08	-0.04	1383.0	2.00e-09	-0.37
PB vs. Extended Shift	547.0	2.34e-05	-0.02	1520.0	1.64e-08	-0.30

suggesting consistent and substantial improvements when incorporating the position block neighbourhood into the matheuristic across all instances.

For the Time metric, adding the position block neighbourhood to the 'NEH + shift' method consistently increases computational time, as shown by a large effect size (Cliff's $\delta = 0.85$) and our previous computational results. However, the Wilcoxon signed-rank test yields very high p-values of 1, indicating that this increase is not statistically significant. This is likely because the position block neighbourhood slows down the algorithm in a consistent way each time, resulting in little variation in the data. Since the Wilcoxon signed-rank test relies on variability to detect significance, the lack of variation prevents it from identifying the increase as statistically significant.

On the other hand, the position block is significantly faster than the randomised delta and extended shift neighbourhoods, with p-values well below 0.05 and negative effect sizes. However, when compared to other neighbourhoods, the differences in computational time are not statistically significant at the 0.05 level.

Table 10 presents the Wilcoxon signed-rank test results for all 200 small and 240 large Vallada et al. instances. Due to their volume and complexity, only the position block neighbourhood is applied, as described in Section 5.4. This test evaluates whether it leads to a statistically significant improvement in our matheuristic.

Table 10: Results from the Wilcoxon signed-rank test for Vallada *et al.* instances

Rules	Size	%Gaps			Time		
		W	*p-value*	Cliff's δ	*W*	*p-value*	Cliff's δ
Matheuristic vs. NS	*small*	0.0	9.32e-32	-0.76	20100.0	1.00e+00	0.98
	large	0.0	2.15e-34	-0.35	28840.0	1.00e+00	0.86

The results are consistent with those observed for the Taillard instances.

The position block neighbourhood yields statistically significant improvements in solution quality, as shown by p-values approaching zero and negative effect sizes (Cliff's $\delta = -0.76$ for small instances and Cliff's $\delta = -0.35$ for large instances). However, this improvement comes at the cost of increased running time, although this increase is not statistically significant according to the Wilcoxon signed-rank test.

6 Concluding Remarks

The permutation flowshop problem with the makespan objective is a classic problem in machine scheduling. We have developed five neighbourhoods for the problem and shown how to explore them by solving small-size MIPs. Additionally, we proposed a matheuristic using our proposed neighbourhoods. The computational results indicate that our matheuristic, though conceptually simple, performs quite well in improving initial solutions obtained by 'NEH and shift'. It achieves improvements of up to 66.2% on average for the well-known Taillard instances, and up to 100% and 29.8% on average for the small and large instances of Vallada *et al.*, respectively. Among the neighbourhoods, the 'position block' neighbourhood stands out, particularly for large instances.

We believe that our matheuristic can be easily adapted to other variants of the PFM, including different objectives, such as total tardiness or total flow time, or various constraints like no-wait and no-idle conditions (see, e.g., [24, 11, 26, 35]). An interesting question for future research is whether one could improve the running time of our approach by calling some faster neighbourhood search routines (such as those in [7, 23, 5]) before calling our MIP-based routines.

Acknowledgement

The first author gratefully acknowledges funding from the Engineering and Physical Sciences Research Council (EPSRC) under grant EP/V520214/1, and from the Ministerio de Ciencia, Tecnología e Innovación of Colombia (MINCIENCIAS) through the call '885 de 2020—Doctorados en el Exterior'. We also thank the IT team at Lancaster University, who provided guidance and facilitated the use of the cluster on which we ran our experiments.

References

[1] P. Brandimarte and E. Fadda. A reduced variable neighborhood search for the just in time job shop scheduling problem with sequence dependent setup times. *Comput. Oper. Res.*, 167, 2024. Article 106634.

[2] S. Cáceres Gelvez, T.H. Dang, and A.N. Letchford. On some lower bounds for the permutation flowshop problem. *Comput. Oper. Res.*, 159, 2023. Article 106320.

[3] F. Della Croce, A. Grosso, and F. Salassa. A matheuristic approach for the total completion time two-machines permutation flow shop problem. *Ann. Oper. Res.*, 213:67–78, 2014.

[4] X. Dong, H. Huang, and P. Chen. An improved NEH-based heuristic for the permutation flowshop problem. *Comput. Oper. Res.*, 35:3962–3968, 2009.

[5] J. Dubois-Lacoste, F. Pagnozzi, and T. Stützle. An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Comput. Oper. Res.*, 81:160–166, 2017.

[6] H. Emmons and G. Vairaktarakis. *Flow Shop Scheduling: Theoretical Results, Algorithms, and Applications*. Springer, New York, 2013.

[7] V. Fernandez-Viagas and J.M. Framinan. A best-of-breed iterated greedy for the permutation flowshop scheduling problem with makespan objective. *Comput. Ind. Eng.*, 112, 2019. Article 104767.

[8] V. Fernandez-Viagas, R. Ruiz, and J.M. Framinan. A new vision of approximate methods for the permutation flowshop to minimise makespan: state-of-the-art and computational evaluation. *Eur. J. Oper. Res.*, 257:707–721, 2017.

[9] M. Fischetti, A. Lodi, and D. Salvagnin. Just MIP it! In V. Maniezzo, T. Stützle, and S. Voß, editors, *Matheuristics, Hybridizing Metaheuristics and Mathematical Programming*. Springer, Boston, MA, 2009.

[10] J.M. Framinan, J.N. Gupta, and R. Leisten. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *J. Oper. Res. Soc.*, 55:1243–1255, 2004.

[11] J.M. Framinan, R. Leisten, and R. Ruiz-Usano. Comparison of heuristics for flowtime minimisation in permutation flowshops. *Comput. Oper. Res.*, 32:1237–1254, 2005.

[12] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. *Eur. J. Oper. Res.*, 284:814–833, 2020.

[13] J. Grabowski and M. Wodecki. A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Comput. Oper. Res.*, 31:1891–1909, 2004.

[14] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discr. Math.*, 5:287–326, 1979.

[15] M. Haouari and T. Ladhari. A branch-and-bound-based local search method for the flow shop problem. *J. Oper. Res. Soc.*, 54:1076–1084, 2003.

[16] T.A. Jessin, S. Madankumar, and C. Rajendran. Permutation flowshop scheduling to obtain the optimal solution/a lower bound with the makespan objective. *Sādhanā*, 45, 2020. Article 228.

[17] S.M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Nav. Res. Logist. Q.*, 1:61–68, 1954.

[18] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Ann. Discr. Math.*, 1:343–362, 1977.

[19] V. Maniezzo, M.A. Boschetti, and T. Stützle. *Matheuristics: Algorithms and Implementations*. Springer, Cham, 2021.

[20] V. Nagarajan and M. Sviridenko. Tight bounds for permutation flow shop scheduling. *Math. Oper. Res.*, 34:417–427, 2009.

[21] M. Nawaz, E. Enscore, and I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11:91–95, 1983.

[22] I.H. Osman and C.N. Potts. Simulated annealing for permutation flowshop scheduling. *Omega*, 17:551–557, 1989.

[23] F. Pagnozzi and T. Stützle. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *Eur. J. Oper. Res.*, 276:409–421, 2019.

[24] B.d.A. Prata, L.R. Abreu, and V. Fernandez-Viagas. A systematic review of permutation flow shop scheduling with due-date related objectives. *Comput. Oper. Res.*, 177, 2025. Article 106989.

[25] F. Rossi, M. Nagano, and R.F.T. Neto. Evaluation of high performance constructive heuristics for the flow shop with makespan minimization. *Int. J. Adv. Manuf. Technol.*, 87:125–136, 2016.

[26] R. Ruiz and C. Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *Eur. J. Oper. Res.*, 165:479–494, 2005.

[27] E.F. Stafford, F.T. Tseng, and J.N. Gupta. Comparative evaluation of MILP flowshop models. *J. Oper. Res. Soc*, 56:88–101, 2005.

[28] M.I. Sviridenko. A note on permutation flow shop problem. *Ann. Oper. Res.*, 129:247–252, 2004.

[29] Q.C. Ta, J.C. Billaut, and J.L. Bouquard. Matheuristic algorithms for minimizing total tardiness in the m-machine flow-shop scheduling problem. *J. Intell. Manuf.*, 29:617–628, 2018.

[30] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *Eur. J. Oper. Res*, 47:65–74, 1990.

[31] E. Taillard. Benchmarks for basic scheduling problems. *Eur. J. Oper. Res.*, 64:278–285, 1993.

[32] M.F. Tasgetiren, Q.K. Pan, D. Kizilay, and M.C. Vélez-Gallego. A variable block insertion heuristic for permutation flowshops with makespan criterion. In *Proc. CEC '17*, pages 726–733, Donostia, Spain, 2017. IEEE.

[33] C.P. Tomazella and M.S. Nagano. A comprehensive review of branch-and-bound algorithms: guidelines and directions for further research on the flowshop scheduling problem. *Exp. Syst. Appl.*, 158, 2020. Article 113556.

[34] E. Vallada, R. Ruiz, and J.M. Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *Eur. J. Oper. Res.*, 240:666–677, 2015.

[35] E. Vallada, R. Ruiz, and G. Minella. Minimising total tardiness in the m-machine flowshop problem: a review and evaluation of heuristics and metaheuristics. *Comput. Oper. Res.*, 35:1350–1373, 2008.

[36] L.A. Wolsey. *Integer Programming.* Wiley, New York, 2nd edition, 2020.