INFO-H413

Université Libre de Bruxelles

École Polytechnique de Bruxelles

Heuristic Optimisation Implementation Exercise 1

Author:
Jauffret Oscar

Professor:
Stützle Thomas

April 8th, 2025

Abstract

The permutation flowshop scheduling problem is an NP-hard problem, and many different heuristics have been proposed to tackle it. In this study, we analyze the performance of several stochastic local search algorithms for the PFSP in terms of solution quality and computation time. Twelve variants of Iterative Improvement and two variants of Variable Neighbourhood Descent were considered, using different initialization methods, pivoting rules and neighbourhood relations. We applied Wilcoxon and t-tests on 30 benchmark instances ranging from 50 to 200 jobs to identify the most effective configurations. The best configurations achieved average deviations within 4% of the best-known solutions. The simplified RZ heuristic outperformed random initialization in both solution quality and computation time. First-improvement generally led to better solutions, although it was slower than best-improvement. The insert neighbourhood produced the best-quality solutions, while the transpose neighbourhood was the fastest. Variable Neighbourhood Descent significantly improved performance - over 30% improvement in some cases - compared to single-neighborhood strategies, and was often faster. Between the two VND strategies studied, applying the exchange neighbourhood before the insert yielded slightly better results. Overall, this study highlights how tuning even simple algorithmic components can drastically improve the performance of stochastic local search methods for the PFSP.

Project repository available at: `https://github.com/OscarJauffret/FlowshopSLS`

Contents

1 Introduction

The *Permutation Flowshop Scheduling Problem* (PFSP) is one of the most studied optimization problems due to its relevance in manufacturing, logistics and production planning. It is known to be an NP-hard problem, and finding optimal solutions becomes computationally infeasible for very large instances. For this reason, heuristics have been developed to find near-optimal solutions in reasonable time.

The objective of this study is to evaluate the performance of different variants of iterative improvement algorithms on the PFSP. Several variants of the initialization method, the neighbourhood relation, and the pivot rule are considered. In addition to standard iterative improvement, we also explore a Variable Neighbourhood Descent (VND) algorithm in an effort to escape local optima during the search.

This study was conducted as part of a project for the *Heuristic Optimisation* course.

2 Problem description

In this section, we describe the *Permutation Flowshop Scheduling Problem* (PFSP) in further detail, including the objective and constraints.

In the PFSP, all jobs must pass through every machine in the same order. We assume that there are infinite buffers between machines and no blocking or no-wait constraints are considered. Given a set of n jobs $J_1, ..., J_n$, where each job J_i consists of m operations $o_{i1}, ..., o_{im}$ performed on $M_1, ..., M_m$ machines in that order, with processing time p_{ij} for operation o_{ij}, the objective is to find a permutation π that minimizes the sum of completion times $\sum_{i=1}^{n} C_i$, where $C_i = C_{im}$, the completion time of job i on the last machine.

Let $\pi(k)$ denote the k^{th} job in the permutation and C_{ij} the completion time of job i on machine j. The completion time $C_{\pi(k)j}$ is computed recursively based on the position of the job and machine, as follows.

- The first job on each machine is directly processed as it does not need to wait for another job to free the machine. The completion time of the first job on machine j is the sum of processing times of this job on all previous machines.

$$C_{\pi(1)j} = \sum_{h=1}^{j} p_{\pi(1)h} \quad \forall j \in \{1, ..., m\} \tag{1}$$

- All jobs on the first machine only need to wait for the previous job in the permutation to finish being processed on the machine. They are directly available as they are not stuck in a previous machine.

$$C_{\pi(k)1} = \sum_{h=1}^{k} p_{\pi(h)1} \quad \forall k \in \{1, ..., n\} \tag{2}$$

- For all other cases, the completion time of a job k on a machine j depends on the maximum of two constraints. First, the machine needs to be free, so the previous job needs to be done on this machine. Second, the job needs to be done with the previous machine as it cannot enter a new machine before finishing the previous. Then, we add the processing time of k on j.

$$C_{\pi(k)j} = \max\{C_{\pi(k-1)j}, C_{\pi(k)j-1}\} + p_{\pi(k)j} \quad \forall k \in \{2, ..., n\}, \forall j \in \{2, ..., m\}, \tag{3}$$

3 Material and methods

In this section, we describe how the algorithms are implemented and the experimental setup that was used to conduct the statistical tests. The results of these tests are discussed in Section 4.

3.1 Algorithm implementation

The algorithms were implemented in C++. Detailed documentation can be found in the `include/` source files, we briefly describe the overall structure here.

The `Instance` class holds the data relevant to a given instance, such as the number of jobs and machines, and the processing times. The `Solution` class represents a candidate solution for the given instance. It maintains a permutation of the jobs and provides methods to compute the objective function (sum of completion times), and to generate neighbouring solutions.

Two main solver classes were implemented: `FlowShopII`, which applies Iterative Improvement (II), and `FlowShopVND`, which applies Variable Neighbourhood Descent (VND). The classes use configurable neighborhood iterators, initialization strategies and pivoting rules to explore the solution space and iteratively improve a candidate solution.

3.1.1 Algorithm variants

Multiple variants were developed to study the impact of different algorithmic choices. These include:

- **Initialization**: either a random permutation or a permutation generated by a simplified RZ heuristic.
- **Neighbourhood structure**: *transpose*, *exchange*, or *insert*.
- **Pivoting rule**: *first-improvement* or *best-improvement*.

For the VND algorithm, there are two variants that differ in the order in which the neighbourhoods are visited:

- *Transpose* $\longrightarrow$ *Exchange* $\longrightarrow$ *Insert* (*TEI*)
- *Transpose* $\longrightarrow$ *Insert* $\longrightarrow$ *Exchange* (*TIE*)

All possible combinations of these parameters were tested to identify statistically significant differences in performance (see subsection 3.3).

3.2 Instances

We used 30 instances, which all involved 20 machines. They were divided into three groups according to the number of jobs to schedule (50, 100, and 200). The best-known total completion times are also given to assess the quality of the solutions found by our algorithms.

3.3 Experimental setup

There are 12 configurations[1] of the Iterative Improvement algorithm. Each configuration was executed 10 times on each instance, resulting in a total of 3600 runs of the Iterative Improvement for the 30 benchmark instances.

We used a bash script to automate the experiments, and saved the results to a CSV file. Then, we used R scripts to perform paired Wilcoxon and t-tests to determine if there are statistically significant differences in performance. Comparisons were grouped by algorithmic component — initialization method, neighborhood structure, and pivoting rule — to evaluate the relative performance of each variant independently.

The evaluation of the Variable Neighbourhood Descent algorithms followed a similar methodology. Each of the two neighbourhood orderings was tested 10 times on every instance, leading to a single paired test comparison.

In addition to solution quality, we also recorded the execution time of each run. The same statistical analysis methods were used to identify the fastest pivoting rules and neighborhood structures.

4 Results

In this section, we present and analyze the results obtained by the paired Wilcoxon and t-tests. These tests help determine whether differences in performance metrics are systematic across the tested instances. We use the results of the tests to determine the best variant of each algorithm.

For all tests, a significance level (α) of 0.05 was used. In our case, using another α such as 0.01 would not change anything about the conclusion[2]. Also, the statistical tests were performed on the percentage deviation from best-known solutions.

For clarity, when both tests agree, the results of the t-test are not shown in the tables. Full detailed results, including R scripts that were used to gather the statistics, are available in the `results/` directory of the repository.

Note. In all tables, whenever the p-value indicates that there is a significant difference between the two components tested, the cell corresponding to the better-performing variant is highlighted in gray.

4.1 Summary

We begin by providing an overall summary of solution quality and execution time for the Iterative Improvement algorithm using each initialization method, grouped by instance size, shown in Table 1. Since there are 10 instances for each instance size, 6 configurations, and each configuration is ran 10 times, each row in the table aggregates results from 600 runs.

This overview already suggests that the simplified RZ heuristic outperform the random initialization, both in terms of solution quality and execution time, across all instance sizes.

[1] 2 Initialization methods $\times$ 2 Pivoting rules $\times$ 3 Neighbourhood relations
[2] There were no p-values between 0.05 and 0.01.

Initialization	**Jobs**	**Avg. % Deviation**	**Total Time (ms)**
Random	50	9.027	68,042.2
Random	100	9.511	1,228,812.8
Random	200	9.951	24,790,052.1
SRZ	50	4.341	25,786.7
SRZ	100	5.023	449,621.1
SRZ	200	4.441	7,166,216.2

Table 1: Average percentage deviation from best-known solutions and total execution time for each initialization method and instance size.

4.2 Solution quality analysis

In this subsection, we present and analyze the results of the statistical tests on the quality of the solutions obtained by the Iterative Improvement algorithm. Quality is characterized by a deviation from the best-known solution.

4.2.1 Initialization methods

First, we compare the two initialization methods: random initialization and the simplified RZ heuristic. The results of the comparison are presented in Table 2. In all configurations, both statistical tests agreed on the p-values, which are systematically below the chosen significance level.

Pivot Rule	**Neighborhood**	**Random (%)**	**SRZ (%)**	**p-value$_{\textbf{Wilcoxon}}$**
First	Transpose	19.279	6.225	$< 2.2 \times 10^{-16}$
First	Exchange	3.102	3.892	$< 2.2 \times 10^{-16}$
First	Insert	3.937	3.142	$< 2.2 \times 10^{-16}$
Best	Transpose	20.028	6.218	$< 2.2 \times 10^{-16}$
Best	Exchange	5.626	4.542	$< 2.2 \times 10^{-16}$
Best	Insert	5.007	3.591	$< 2.2 \times 10^{-16}$

Table 2: Comparison between random and SRZ initialization across different configurations. Values are average percentage deviations from best-known solutions.

In most configurations, the simplified RZ heuristic consistently outperforms random initialization, with significant improvements in solution quality. However, when using a first-improvement pivoting rule combined with an exchange neighbourhood, random initialization unexpectedly yields better results. This could be due to the simplified RZ bringing the initial candidate in a hard-to-escape local optimum. The local search could also be more effective from a less structured starting point in this specific setting

These results are not entirely surprising: using a constructive heuristic that steers the initial candidate towards a better solution like RZ often produces better starting points, which reduces the expectation on the local search to reach high-quality solutions.

4.2.2 Pivoting rule

Next, we compare the two pivoting rules: first-improvement and best-improvement. The results are presented in Table 3. Here, the p-values of the t-test are included.

Initialization	**Neighborhood**	**First (%)**	**Best (%)**	**p-value$_{\text{Wilcoxon}}$**	**p-value$_{\text{t-test}}$**
Random	Transpose	19.279	20.028	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Exchange	3.102	5.626	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Insert	3.937	5.007	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Transpose	6.225	6.218	0.077	0.106
SRZ	Exchange	3.892	4.542	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Insert	3.142	3.591	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$

Table 3: Comparison between first- and best-improvement pivoting rules across different configurations. Values are average percentage deviations from best-known solutions.

Across all configurations except one, first-improvement outperforms best-improvement. This only exception occurs when the initialization method is the simplified RZ heuristic and the neighbourhood is transpose: in this specific configuration, both p-values are larger than our significance level, indicating that the difference is not statistically significant.

These results might seem counterintuitive at first, as we would expect the best-improvement to yield better results. However, the greedy best-improvement may cause it to quickly settle in a local optimum. Since no perturbation step is considered in the algorithm, escaping from these local optima is not possible, making first-improvement a better option in practice.

4.2.3 Neighbourhood relation

Finally, we compare the neighbourhood relations: transpose, exchange, and insert pairwise and analyze the results.

Transpose vs Exchange: Let's start by comparing the transpose and exchange neighbourhood structures. The results are presented in Table 4. Once again, both statistical tests agreed on a p-value smaller than the significance level, indicating a clear performance difference.

Initialization	**Pivot Rule**	**Transpose (%)**	**Exchange (%)**	**p-value$_{\text{Wilcoxon}}$**
Random	First	19.279	3.102	$< 2.2 \times 10^{-16}$
Random	Best	20.028	5.626	$< 2.2 \times 10^{-16}$
SRZ	First	6.225	3.892	$< 2.2 \times 10^{-16}$
SRZ	Best	6.218	4.542	$< 2.2 \times 10^{-16}$

Table 4: Comparison between transpose and exchange neighborhood structures across different configurations. Values are average percentage deviations from best-known solutions.

The conclusion is clear: in every configuration tested, the exchange neighbourhood significantly outperforms the transpose neighbourhood. This result aligns with expectations, as the exchange neighbourhood is more general. It includes all the transitions offered by the transpose neighbourhood, with some others that can help escape local minima.

Transpose vs Insert: Next, we compare the transpose and insert neighbourhood relations. The results are presented in Table 5. Similarly to the previous test, we see a clear performance difference.

Initialization	**Pivot Rule**	**Transpose (%)**	**Insert (%)**	**p-value$_{\text{Wilcoxon}}$**
Random	First	19.279	3.937	$< 2.2 \times 10^{-16}$
Random	Best	20.028	5.007	$< 2.2 \times 10^{-16}$
SRZ	First	6.225	3.142	$< 2.2 \times 10^{-16}$
SRZ	Best	6.218	3.591	$< 2.2 \times 10^{-16}$

Table 5: Comparison between transpose and insert neighborhood structures across different configurations. Values are average percentage deviations from best-known solutions.

As with the previous comparison, the insert neighbourhood consistently outperforms the transpose neighbourhood. The explanation is the same; the insert neighbourhood can simulate the transpose neighbourhood, and has extra transitions that can be useful to escape from local optima.

Exchange vs Insert: Finally, we compare the exchange and insert neighbourhood strategies. The results are shown in Table 6. Both tests indicate a performance difference on each configuration.

Initialization	**Pivot Rule**	**Exchange (%)**	**Insert (%)**	**p-value$_{\text{Wilcoxon}}$**
Random	First	3.102	3.937	$< 2.2 \times 10^{-16}$
Random	Best	5.626	5.007	$< 2.2 \times 10^{-16}$
SRZ	First	3.892	3.142	$< 2.2 \times 10^{-16}$
SRZ	Best	4.542	3.591	$< 2.2 \times 10^{-16}$

Table 6: Comparison between exchange and insert neighborhood structures across different configurations. Values are average percentage deviations from best-known solutions.

The results show that the insert neighbourhood performs better than exchange neighbourhood in three out of four configurations, with all differences being statistically significant. In the case of random initialization with first-improvement, exchange yields slightly better solutions, but overall insert seems to be the more effective neighbourhood.

This observation aligns with the idea that the insert neighbourhood is more expressive than the exchange neighbourhood. In general, it can simulate an exchange transition through two consecutive insertions. However, the additional flexibility of the insertion neighbourhood could make it more dependent on the pivoting strategy used, which could explain why it is not consistently better than the exchange neighbourhood.

Conclusion of the neighbourhood quality analysis: The experiments showed that the insert neighbourhood usually provided the best solution quality, followed closely by exchange, while transpose yielded significantly worse results. This confirms the intuition that a larger neighbourhood size provides better results.

4.3 Execution time analysis

In this subsection, we present and analyze the results obtained from the statistical tests for the execution times of the different variants of the Iterative Improvement algorithm. All experiments were run on an 11th Gen Intel Core i7-11700k CPU operating at 3.6 GHz.

4.3.1 Pivoting rule

To determine which pivoting rule is faster, we performed Wilcoxon and t-tests on the execution times of each configuration for the 3 instance sizes (50, 100, and 200 jobs). We compare the performances of the first- and best- improvement strategies in Table 7.

Init.	**Neighborhood**	**Jobs**	**First (ms)**	**Best (ms)**	**p-value$_{\text{Wilcoxon}}$**	**p-value$_{\text{t-test}}$**
Random	Transpose	50	4.096	4.196	0.0078	0.0040
Random	Transpose	100	25.428	25.184	0.986	0.313
Random	Transpose	200	243.295	222.106	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Exchange	50	132.567	80.158	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Exchange	100	2563.38	1258.187	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Exchange	200	61996.3	20893.0	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Insert	50	281.179	178.227	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Insert	100	5701.72	2714.23	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
Random	Insert	200	122218	42327.7	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Transpose	50	7.578	7.738	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Transpose	100	47.451	48.207	$< 2.2 \times 10^{-16}$	1.85×10^{-11}
SRZ	Transpose	200	366.436	367.237	0.00029	0.066
SRZ	Exchange	50	33.322	31.519	7.25×10^{-8}	0.00027
SRZ	Exchange	100	773.893	508.617	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Exchange	200	16770.4	6805.57	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Insert	50	94.924	82.787	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Insert	100	1949.58	1168.46	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
SRZ	Insert	200	30966.1	16386.4	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$

Table 7: Comparison of execution times (in milliseconds) between first- and best-improvement pivoting rules, across different configurations and instance size.

One trend clearly emerges, we see that best-improvement is generally faster than first improvement. This mirrors our results from the quality analysis. One explanation is that best-improvement more quickly converges to local optima, thus terminating earlier.

There are, however, two configurations where no significant difference in execution time was observed. First, when using random initialization and the transpose neighbourhood for 100 jobs, both tests confirm that there is no significant statistical difference. Second, with SRZ initialization and transpose neighbourhood for 200 jobs, the Wilcoxon test suggests a difference, while the t-test does not.

4.3.2 Neighbourhood relation

Transpose vs Exchange: First, let's compare the transpose and exchange neighbourhood relations, to see which one is faster. Again, the same statistical tests were conducted, and the results are presented in Table 8.

Init	**Pivot**	**Jobs**	**Transpose (ms)**	**Exchange (ms)**	**p-value$_{\textbf{Wilcoxon}}$**
Random	First	50	4.096	132.567	$< 2.2 \times 10^{-16}$
Random	First	100	25.428	2563.38	$< 2.2 \times 10^{-16}$
Random	First	200	243.295	61996.319	$< 2.2 \times 10^{-16}$
Random	Best	50	4.196	80.158	$< 2.2 \times 10^{-16}$
Random	Best	100	25.184	1258.187	$< 2.2 \times 10^{-16}$
Random	Best	200	222.106	20893.002	$< 2.2 \times 10^{-16}$
SRZ	First	50	7.578	33.322	$< 2.2 \times 10^{-16}$
SRZ	First	100	47.451	773.893	$< 2.2 \times 10^{-16}$
SRZ	First	200	366.436	16770.441	$< 2.2 \times 10^{-16}$
SRZ	Best	50	7.738	31.519	$< 2.2 \times 10^{-16}$
SRZ	Best	100	48.207	508.617	$< 2.2 \times 10^{-16}$
SRZ	Best	200	367.237	6805.574	$< 2.2 \times 10^{-16}$

Table 8: Comparison of execution times (in milliseconds) between transpose and exchange neighbourhoods, across different configurations and instance size.

The results are unsurprising; the transpose neighbourhood is significantly faster in every configuration. This is expected, as the transpose operator examines fewer possible moves than exchange. This is also consistent with our findings from the quality analysis. While the exchange neighbourhood provides better-quality results, it is computationally more expensive.

Transpose vs Insert: This comparison is very similar to the previous and the results are almost identical to those of transpose vs exchange: the transpose neighbourhood is significantly faster in all cases. Since the conclusion is the same and the performance gap is even wider, we omit the full table here. Full data is available in the `/results` folder of the project repository.

Exchange vs Insert: Finally, let's compare exchange and insert neighbourhoods execution times. The results are presented in Table 9.

Init	**Pivot**	**Jobs**	**Exchange (ms)**	**Insert (ms)**	**p-value$_{\textbf{Wilcoxon}}$**
Random	First	50	132.567	281.179	$< 2.2 \times 10^{-16}$
Random	First	100	2563.380	5701.717	$< 2.2 \times 10^{-16}$
Random	First	200	61996.319	122218.084	$< 2.2 \times 10^{-16}$
Random	Best	50	80.158	178.227	$< 2.2 \times 10^{-16}$
Random	Best	100	1258.187	2714.232	$< 2.2 \times 10^{-16}$
Random	Best	200	20893.002	42327.715	$< 2.2 \times 10^{-16}$
SRZ	First	50	33.322	94.924	$< 2.2 \times 10^{-16}$
SRZ	First	100	773.893	1949.582	$< 2.2 \times 10^{-16}$
SRZ	First	200	16770.441	30966.071	$< 2.2 \times 10^{-16}$
SRZ	Best	50	31.519	82.787	$< 2.2 \times 10^{-16}$
SRZ	Best	100	508.617	1168.460	$< 2.2 \times 10^{-16}$
SRZ	Best	200	6805.574	16386.403	$< 2.2 \times 10^{-16}$

Table 9: Comparison of execution times (in milliseconds) between exchange and insert neighbourhoods, across different configurations and instance size.

These findings are consistent with the theoretical sizes of the neighbourhood relations. Since the neighbourhood size for the insert neighbourhood is larger than the one of exchange neighbourhood, insert takes more time to complete.

Conclusion of the neighbourhood time analysis: Overall, the experiments confirm that more expressive neighbourhood structures increase computation time. The transpose neighbourhood clearly is the fastest neighbourhood, but it is consistently less effective in guiding the search towards better solutions, as we saw in the quality analysis. However, for time-constrained applications, it can be a good fit.

4.4 Variable Neighbourhood Descent

4.4.1 Summary of VND vs single neighbourhoods

First, we provide an overall summary of the solution quality and execution time of the VND algorithm using each neighbourhood ordering, grouped by instance size. The results are shown in Table 10. Since there are 10 instances for each instance size, and each configuration is ran 10 times, each row in the table aggregates results from 100 runs.

Strategy	**Jobs**	**Avg. % Deviation**	**Avg. Time (ms)**
TEI	50	3.010	103.08
TEI	100	3.492	1239.35
TEI	200	3.526	18825.50
TIE	50	2.764	88.81
TIE	100	3.839	1280.90
TIE	200	3.657	19512.16

Table 10: Comparison of TEI and TIE VND strategies by average deviation from best-known solutions and execution time per instance size.

We can already observe that the solutions found by the Variable Neighbourhood Descent algorithm consistently achieve lower deviation values than those produced by the Iterative Improvement algorithm. Execution times fall between the extremes observed with single-neighbourhood strategies, such as purely transpose (fastest) and insert (slowest).

Then, we evaluate the percentage improvement brought by VND over the usage of a single neighborhood. The results are presented in Table 11. The formula used to compute the quality improvement is the following:

$$\text{Improvement} = 100 \cdot \frac{\%\text{dev}_{\text{single}} - \%\text{dev}_{\text{VND}}}{\%\text{dev}_{\text{single}}} \tag{4}$$

We can observe that the Variable Neighborhood Descent provides better-quality solutions than local search using a single neighbourhood. In some cases, the improvement exceeds 30%, although most values fall between 10% and 20%.

Regarding execution time, VND is sometimes significantly faster (up to 64%), especially compared to the insert neighbourhood. However, it is notably slower than the exchange neighbourhood in a few configurations, particularly for small instances.

Strategy	**Jobs**	**Compared To**	**Quality improvement (%)**	**Time improvement (%)**
TEI	50	Exchange	30.26	-48.55
TEI	50	Insert	11.02	35.28
TEI	100	Exchange	21.38	2.87
TEI	100	Insert	14.05	57.02
TEI	200	Exchange	14.30	29.27
TEI	200	Insert	18.23	64.46
TIE	50	Exchange	35.96	-27.98
TIE	50	Insert	18.29	44.24
TIE	100	Exchange	13.55	-0.38
TIE	100	Insert	5.49	55.58
TIE	200	Exchange	11.13	26.69
TIE	200	Insert	15.21	63.17

Table 11: Improvement of VND strategies over single neighbourhood local search. Positive values mean VND is better. Negative time improvement means VND is slower.

4.4.2 Comparison of neighbourhood orderings in VND

Now, let's compare the two neighbourhood orderings considered in the VND: transpose → exchange → insert (TEI) and transpose → insert → exchange (TIE). Since this is the only difference in the configurations, a single test is necessary.

TEI (%)	**TIE (%)**	**p-value$_{\textbf{Wilcoxon}}$**	**p-value$_{\textbf{t-test}}$**
3.343	3.420	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$

Table 12: Comparison of VND strategies TEI and TIE on all instances.
Values are average percentage deviations from best-known solutions.

Both statistical tests show a significant difference in performance. The TEI strategy outperforms TIE with a slightly better average deviation from optimal solutions. While the difference is small in absolute terms, it is consistent enough across instances to be statistically significant. This suggests that prioritizing the exchange neighborhood over insert may lead to slightly more effective descent strategies in VND.

5 Conclusion

In this report, we conducted a experimental analysis of the Iterative Improvement algorithm and Variable Neighbourhood Descent algorithms for solving the Permutation Flowshop Problem. We evaluated the impact of the initialization methods, pivoting rules, neighbourhood relations and neighbourhood orderings on the solution quality and the execution time.

We saw that the proposed solutions were within 4% of the best-known solutions for our best-performing configurations.

The results showed that the best initilization method was the simplified RZ heuristic in most cases, both in terms of solution quality and execution time. It outperformed random initialization on most configurations. First-improvement, although slower most of the time, generally led to better solutions than best-improvement. Among the neighbourhood relations, transpose was the fastest to terminate, while insert was the most effective for finding high quality solutions. Exchange neighbourhood was in the middle in terms of solution quality and execution time.

We also confirmed the superiority of VND over single-neighbourhood strategies, with improvements of up to 30% in solution quality. It was also generally faster, except compared to exchange neighbourhood on small instance sizes. The TEI strategy slightly outperformed TIE, with the difference being statistically significant.

Overall, this study highlights how fine-tuning even simple local search components can lead to significant performance gains, and how VND strategies can further enhance solution quality.