ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H-413 Heuristic Optimization

Implementation #1

Silberwasser Elliot 000518397

April 8, 2025

Contents

1 Abstract

This project aims to implement and test the Iterative Improvement algorithms and the Variable Neighborhood Descent algorithms on the **Permutation Flow-Shop Scheduling Problem with total Completion Time objective** (PFSP-CT). All the project is written in JAVA JDK 24.

This report covers the following points:

- Introduction
- Implementation of the algorithms
- Methodology of the tests
- Results
- Comparison of the results

2 Introduction

2.1 Problem description

The Permutation Flow-Shop Scheduling Problem with total Completion Time objective is an industrial optimization problem. In fact, we have several jobs composed by operations to be executed on several machines with some completion times. All jobs pass through the machines in the same order and are available at time zero. The problem consists in matching each job to each machine to minimize the total completion time of the solution.

2.2 Formal definition

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}.

Objective: Find a permutation π that minimizes the sum of the completion times $\sum_{i=1}^{n} C_i$.

2.3 Algorithms used to solve the problem

To solve the problem described above, we used the class of Iterative Improvement Algorithms. In fact, these algorithms are powerful to find an optimal solution of the problem. Their principle is to improve an initial solution to reach a solution close to the most optimal in an efficient way. To achieve this goal, there is a trade-off between optimality and efficiency.

For the first part of the project, we consider **12** variants of iterative improvement algorithms. We consider two pivoting rules, called **first** and **best Iterative Improvement**. We consider three neighborhood policies defined in the following way:

For two jobs $i, j \in [Jobs]$ with $i < j$:

1. **Transpose:** We swap i and j in the current permutation if there are adjacent.
2. **Exchange:** We simply swap i and j without constraints in the current permutation.
3. **Insert:** We insert the job i after the job j in the current permutation.

Finally, we consider two initializing methods to initialize the starting sequence of these variants. A random uniform permutation and a simplified RZ heuristic.

In the second part of the project, we implement the Variable Neighborhood Descent algorithms. In particular, two neighborhood orders, respectively **Transpose** -> **Exchange** -> **Insert** and **Transpose** -> **Insert** ->

Exchange. Those variants are based on the initial SRZ permutation heuristic and on the first iterative Improvement algorithm.

3 Implementation of the algorithms

To read the data from the benchmark files, a `readFile(String fileName)` function is executed. The data is stored in some variables as `processingTimesMatrix`, `numJobs` and `numMachines`.

3.1 Iterative Improvement Algorithms

To implement the algorithms, we start with the first and best improvement algorithm. Those functions are very similar in their signature. Their arguments are just an initial permutation and a neighborhood policy. Inside of the functions, we have a `switch (neighborhood) case "policy":` to manage the different policies wondered by the user. In that way, we combine the three neighborhood policies in the same function to avoid several algorithms. The two functions differ only in their process to find the best solution depending on the selected pivoting rule.

3.1.1 Random Uniform Permutation

To generate a Random Uniform Permutation, we create an initial sorted array initialized by the order index of the jobs. Then, we shuffle this initial array to get our random uniform permutation. To shuffle the array, we select an index of the array randomly and apply a swap between two jobs.

3.1.2 Simplified RZ Permutation

To generate an initial permutation following the simplified RZ heuristic, we compute the T_i array to sort the jobs in increasing order. Then, we give this initial solution to the `generateBestInitSolution()` function to enumerate (step by step by adding one job at a time) all the partial solutions and keep the best one which minimize the completion time of the initial permutation.

NOTE: If two partial solutions at a given step have the same minimum completion time, the choice policy will be based on the last solution found. Therefore, we keep the last permutation with the minimum completion time.

3.2 Variable Neighborhoods Descent Algorithms

Due to the fact that the VND algorithm is based on several variants described above, it implementation is very similar. We start by initialize the initial solution by calling the `initializePermutation()` function. In that way, we give to it the `"--srz"` argument. The differences appear in the progress of the algorithm. Indeed, we follow the neighborhood order given in argument to improve the permutation.

3.3 Launching the algorithms

To launch a specific algorithm k for a specific instance `taXXX`, you can do it in the following way:

```
$java IterativeImprovement ./Benchmarks/<taXXX> --<pivoting_rule> --<neighborhood_policy>
                                    --<initial_method>
```

OR

```
$java IterativeImprovement ./Benchmarks/<taXXX> --vnd --<neighborhood_order>
```

NOTE: Make sure that a benchmarks folder with all the instances to be tested is present in your ”src” folder. If you download the project from the github link below, normally the correct file tree will already be built.

4 Tests

To write readable tests, we decide to test all variants one by one in a specific function. In that way, we can easily test each algorithm independently. This choice creates duplicate codes. However, it simplifies the readability of tests and the understanding of results.

4.1 Methodology of the tests

To test our algorithms, we decide to launch them ten times on three different instance sizes (50, 100, and 200 jobs).

For each test, we compute the completion time of the solution found as well as its deviation from the best known value for the instance being tested. We also compute the computation time for instances of the same size.

At the end of the tests, we display in the terminal the average percentage deviation from the best known value for all instances and the sum of the total completion time. Therefore, the given percentage corresponds to the overall deviation for any instance size, not for a specific size.

4.2 Launching test

To launch a test on a specific algorithm k for each instance and 10 times, you need to execute the java class file with the following arguments:

```
$java IterativeImprovement --test --<pivoting_rule> --<neighborhood_policy>
                           --<initial_method>
```

OR

```
$java IterativeImprovement --test --vnd --<neighborhood_order>
```

where the neighborhood order can be either `--one` or `--two`.

5 Results

All results were computed from a virtual machine running on Linux Ubuntu version 24. The interpretation of the rapidity to obtain an optimal solution from the algorthims can be impacted by this setup.

Algorithms	Average relative % deviation	Sum of Completion Time (TCT)
BestExchangeRandomII	4.635022 %	180246093
BestTransposeRandomII	19.954693 %	207701659
BestInsertRandomII	8.589878 %	187414845
BestExchangeSrzII	4.6111436 %	180277420
BestTransposeSrzII	19.828869 %	207755130
BestInsertSrzII	8.767928 %	188337750
FirstExchangeRandomII	3.1579356 %	177483376
FirstTransposeRandomII	19.291025 %	206379407
FirstInsertRandomII	7.692353 %	185682557
FirstExchangeSrzII	3.066168 %	177602510
FirstTransposeSrzII	19.131846 %	206325848
FirstInsertSrzII	7.6131186 %	185839980
VNDFirstOrder	3.419995 %	178203070
VNDSecondOrder	4.049377 %	179497180

The following table is a presentation based on initial permutation and the instance size (i.e 50, 100 or 200 jobs). The results have obtained with the exchange neighborhood policy. It is obvious from the previous table that it produces the closer result from the best known value. To collect the computation time, we decide to test the algorithms with the most optimal variant of neighborhood.

Algorithms	Initial Solution	Instance Size	Total Compu. Time	Avg Comp.Time Per Test
BestII	Random	50	2321 ms	23.21 ms
BestII	Simplified RZ	50	2368 ms	23.68 ms
BestII	Random	100	19223 ms	192.29 ms
BestII	Simplified RZ	100	18656 ms	186.56 ms
BestII	Random	200	204594 ms	2045.94 ms
BestII	Simplified RZ	200	199302 ms	1993.02 ms
FirstII	Random	50	19801 ms	198.01 ms
FirstII	Simplified RZ	50	20557 ms	205.57 ms
FirstII	Random	100	364052 ms	3640.52 ms
FirstII	Simplified RZ	100	376446 ms	3764.46 ms
FirstII	Random	200	8746982 ms	87469.82 ms
FirstII	Simplified RZ	200	9068289 ms	90682.89 ms
VNDFirstOrder	Simplified RZ	50	12241 ms	122.41 ms
VNDFirstOrder	Simplified RZ	100	161919 ms	1619.19 ms
VNDFirstOrder	Simplified RZ	200	3320808 ms	33208.08 ms
VNDSecondOrder	Simplified RZ	50	13991 ms	139.91 ms
VNDSecondOrder	Simplified RZ	100	140187 ms	1401.87 ms
VNDSecondOrder	Simplified RZ	200	2964769 ms	29647.69 ms

6 Comparison of the results

To compare these results, we can answer the following questions.

6.1 Which initial solution is preferable ?

Based on the results, we can see that the SRZ initial solution is preferable. In fact, there is an advantage for this initializing method. It can be explained by the fact that this is a deterministic method and because we start from a more optimal initial sequence. Furthermore, in terms of efficiency, the SRZ heuristic is also faster than the random uniform permutation. Its deterministic side also makes it easier to monitor and reproduce the results.

6.2 Which pivoting rule generates better quality solutions and which is faster ?

We can not easily say that a pivoting rule is better than the other. However, each has its advantages and disadvantages. In fact, we can see from the results table above that the pivot rule "`first`" produces slightly better results in terms of optimality. Indeed, the average relative percentage deviation rate is in favor of this rule. The randomness of the random uniform permutation may not necessarily start with a sufficiently optimal initial sequence and thus increase the average rate of deviation from the optimal solution.

However, the `"first"` pivoting rule was particularly slow during the tests. This is surprising, knowing that in theory it should be faster. It can be explained by the structure of the algorithm. In fact, both algorithms differ in their progression. The pivoting rule `"first"` stops more quickly on a new partially optimal solution. Unlike the `"best"` pivoting rule, which retains only the best one among all the permutations (the neighborhood).

By this logic, first-improvement traverses the neighborhood more quickly, but once an improvement has been found, it is applied and the entire neighborhood is again explored from the beginning, starting with the new solution. However, to reach the same local optimum as the best-improvement, it might take more iterations for the first-improvement and this may explain the reason for its slowness. We don't think about a code optimization problem because we call a function called `computeCompletionTimeMatrixAfterMove()` to compute the cost matrix of the permutation. This function does not compute this matrix from the beginning but from job i between two permutations π and π'.

6.3 Which neighborhood generates better quality solution and what computation time is required to reach local optima ?

The exchange neighborhood policy generates better quality solution. In fact, we can easily see on the first table that for all iterative improvement methods, and for all initialize methods, the exchange neighborhood policy produces the most optimal solution and the closer from the best known value for each instance. Conversely, the transpose neighborhood policy is the worst of all with a difference from the best known value of almost 20 %. It can be explained by the fact that this policy swap two jobs in the new permutation only if their are adjacent. Therefore, improvement is rarer. However, in our tests, this policy was faster than the others. Again, because there is less improvement, there are fewer iterations.

6.4 For the VND algorithms, which neighborhood order is preferable ?

To answer this question, we can see in the first table above that the first neighborhood order produces better results than the second. Moreover, the first neighborhood order is also faster regardless of the size of the instances. Due to the fact that the both VND variants are based on the first-improvement algorithm, we do not have a large difference in their computation time, only that it is also quite long

The best results in terms of optimality of the first order can be explained by the fact that both orders start from the transpose policy. However, since we apply exchange first, this will significantly improve the solution and therefore find a better solution. Unlike the second order, which first uses the insert policy, which then creates too little improvement for the exchange policy.

Because the efficiency of the two neighborhood policies, namely exchange and insert, is so close, local search is too weak to find a better result. Applying exchange first allows us to directly achieve a better partial solution. This way, when arriving at the insert policy, first-improvement will have fewer iterations to produce, since exchange will have already done a good part of the work.

To summarize, since we stop at the first improvement, start with a strong neighborhood improve the quality of the solution.

Conversely, starting with the insert policy can lead to getting trapped more quickly in a less good local optimum.

7 Statistical tests

Normally, to reinforce our results, we should make a script in **R**, a statistical software. In that way, we can explore with the Wilcoxon signed-rank test if there is a statistically significant difference between the solution quality generated by the different algorithms. However, we did not have enough time to do this, which also explains the weak interpretation of our results. This comes from our implementation of the tests. In fact, in order to save more time, we should have written the results of our tests for each instance in a csv file. This would have saved us precious time for this part, especially given the number of test launches to achieve the results described above and the considerable time it took. The R file present in the project is the beginnings of a potential test but is in no way a file to be considered.

8 Project github repository

https://github.com/esilberw/Implementation1-H413.git