M-IRIFS:1 — 2020-2021

INFO-F413 - Heuristic Optimization

Implementation exercise 2

Author:
Nicolas WALLEMACQ

Professor:
Thomas Stützle

Assistant:
Federico PAGNOZZI

19th of May, 2021

Contents

1 Introduction

The overall goal of this year's INFO-F413's project is to is to implement Stochastic Local Search (SLS) algorithms for the Permutation Flow-shop Scheduling Problem (PFSP) with the weighted sum of competion times as objective.

1.1 PFSP

In the context of production lines, one can imagine having different types of a product requiring to go through different machines, with the processing time being specific to each type of product and machine it goes through.

Optimization can help speed up the production line by reducing the amount of time "lost" by finding an optimal sequence for the jobs to be completed. The PFSP tries to find this sequence with permutations.

More particularly, in the PFSP, the following will be assumed:

- The jobs pass through all the machines in the same order
- There are no constraints: infinite buffers between machines, no blocking, no wait requirements

2 Tasks and Results

This second implementation exercise is about implementing 2 different SLS algorithms for the PFSP with the weighted completion time objective (PSFP-WCT) building on top of the constructive and perturbative local search methods from the first implementation exercice.

The student is free to choose which SLS algorithm to implement amongst the following classes of methods:

- simple
- hybrid
- population-based

However, the two SLS methods chosen must belong to two different classes.

The two algorithms implemented should be evaluated on all instances provided for the first implementation exercise and be compared using statistical tests. The implementation should make use of appropriately selected iterative improvement algorithms from the first implementation exercise. The student needs top justify the choice of the iterative improvement algorithm used from the ones implemented for the first implementation exercise.

The experimental comparison should be run under same conditions (programming language, compiler and compiler flags, similar load on computer etc.) for the two algorithms. In other words, the observed differences should be attributable to differences in the algorithms and not to differences in the experimental conditions.

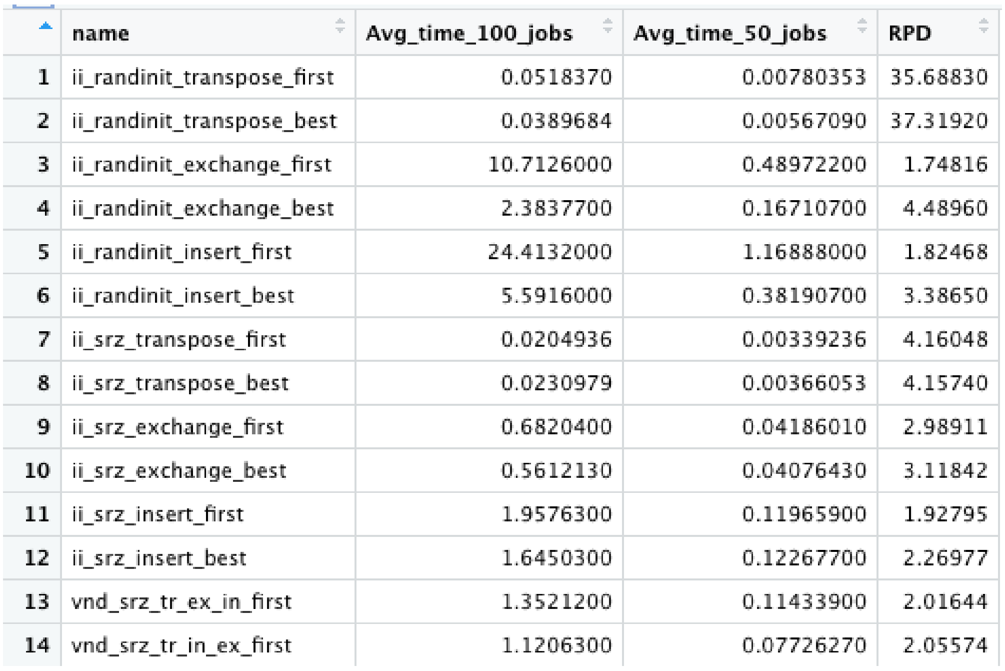
For each of the implemented SLS algorithms, it is asked to :

- describe how it works, which are the used algorithmic components, the main features
- interpret the computational results carefully
- justify the choice of the parameter settings used, the choice of the method for generating initial solutions and the choice of the iterative improvement algorithm

2.1 Summary of the results of Implementation 1

Those results of the first implementation's algorithms are summarized in the table presented in Figure 1. Note that:

- **ii** and **vnd** correspond to the iterative improvement algorithm corresponding to the exercise 1.1 and the variable neighborhood descent corresponding to the exercise 1.2, respectively.
- **randinit** and **srz** correspond to the random job permutation for the initial solution and the simplified RZ heuristic, respectively
- **transpose**, **exchange**, **insert** correspond to the neighborhood chosen.



	name	Avg_time_100_jobs	Avg_time_50_jobs	RPD
1	ii_randinit_transpose_first	0.0518370	0.00780353	35.68830
2	ii_randinit_transpose_best	0.0389684	0.00567090	37.31920
3	ii_randinit_exchange_first	10.7126000	0.48972200	1.74816
4	ii_randinit_exchange_best	2.3837700	0.16710700	4.48960
5	ii_randinit_insert_first	24.4132000	1.16888000	1.82468
6	ii_randinit_insert_best	5.5916000	0.38190700	3.38650
7	ii_srz_transpose_first	0.0204936	0.00339236	4.16048
8	ii_srz_transpose_best	0.0230979	0.00366053	4.15740
9	ii_srz_exchange_first	0.6820400	0.04186010	2.98911
10	ii_srz_exchange_best	0.5612130	0.04076430	3.11842
11	ii_srz_insert_first	1.9576300	0.11965900	1.92795
12	ii_srz_insert_best	1.6450300	0.12267700	2.26977
13	vnd_srz_tr_ex_in_first	1.3521200	0.11433900	2.01644
14	vnd_srz_tr_in_ex_first	1.1206300	0.07726270	2.05574

Figure 1: Performances of all the algorithms. The average times are expressed in seconds and the RPD in %

After conducting statistical tests and comparisons between the different algorithms, here are the conclusions I drew for the iterative improvement algorithms:

- To generate an initial solution, the Simplified RZ heuristic (SRZ) was preferable as it guaranteed to lead towards a relatively good solution.
- When using SRZ to generate an initial solution, the first-improvement pivoting rule lead to better results with the same speed than best-improvement.
- When using SRZ to generate an initial solution and first improvement as a pivoting rule, the insert neighborhood was best but took the most time to compute.

Here are the conclusions I drew for the Variable Neighborhood Descent algorithms:

- When using the SRZ to generate an initial solution and the first improvement as a pivoting rule, the VND algorithm performed better than a local search in a single neighborhood "transpose" or "exchange" as it got a better RPD. However , it lead to statistically similar solutions than a local search in a single neighborhood "insert".

- When using the SRZ to generate an initial solution and the first improvement as a pivoting rule, the ordering of the neighborhoods transpose-exchange-insert or transpose-insert-exchange lead to statistically similar results, so none was preferable above the other in terms of objective function. However, it is worth noting that the order transpose-insert-exchange was faster to compute.

2.2 Simple SLS Method: Random Iterative Improvement

The simple SLS method I chose to implement is the **Random Iterated Improvement** (RII). It is an extension of the iterative improvement implemented in the first implementation, except that sometimes we accept a worsening step by selecting a random neighbor. The interval at which we do a random move (Random Walk) is fixed by a parameter *wp* called *walk probability.* The initial solution can either be a random initialization or given by an heuristic.
Given the conclusions I drew from the first implementation exercise, I decided to apply a simple SLS method with:

- A simplified SRZ heuristic to generate a good candidate initial solution
- First improvement as pivoting rule
- Insert as neighborhood

The values for the parameters of my RII are discussed below but let's firs understand better my implementation. The flowchart representing how my RII algorithm works and its tuning I've made is presented on Figure 2.

I start with an initial solution given by SRZ heuristic. Since it is our only solution so far, I keep it as the best solution. I then enter in a while loop which will only stop once the elapsed time exceeds the stopping criterion. At the beginning of each "round", I save the solution in *Solution_prev.* I then draw a random number *p* between 0 and 100.
If this number is lower of equal to *walk_probability*, I do *nbPerturbations* times a Random Walk.One Random Walk consists in choosing randomly between transpose, exchange or insert neighborhood and applying a random move with this neighborhood. It is worth noting that one perturbation can consist in different local moves applied, e.g. a random exchange move followed by a random insertion

Otherwise, I apply a first improvement on the solution with the insert neighborhood.

Depending on whether we applied a random move or an improvement, there are 3 outcomes at this stage:

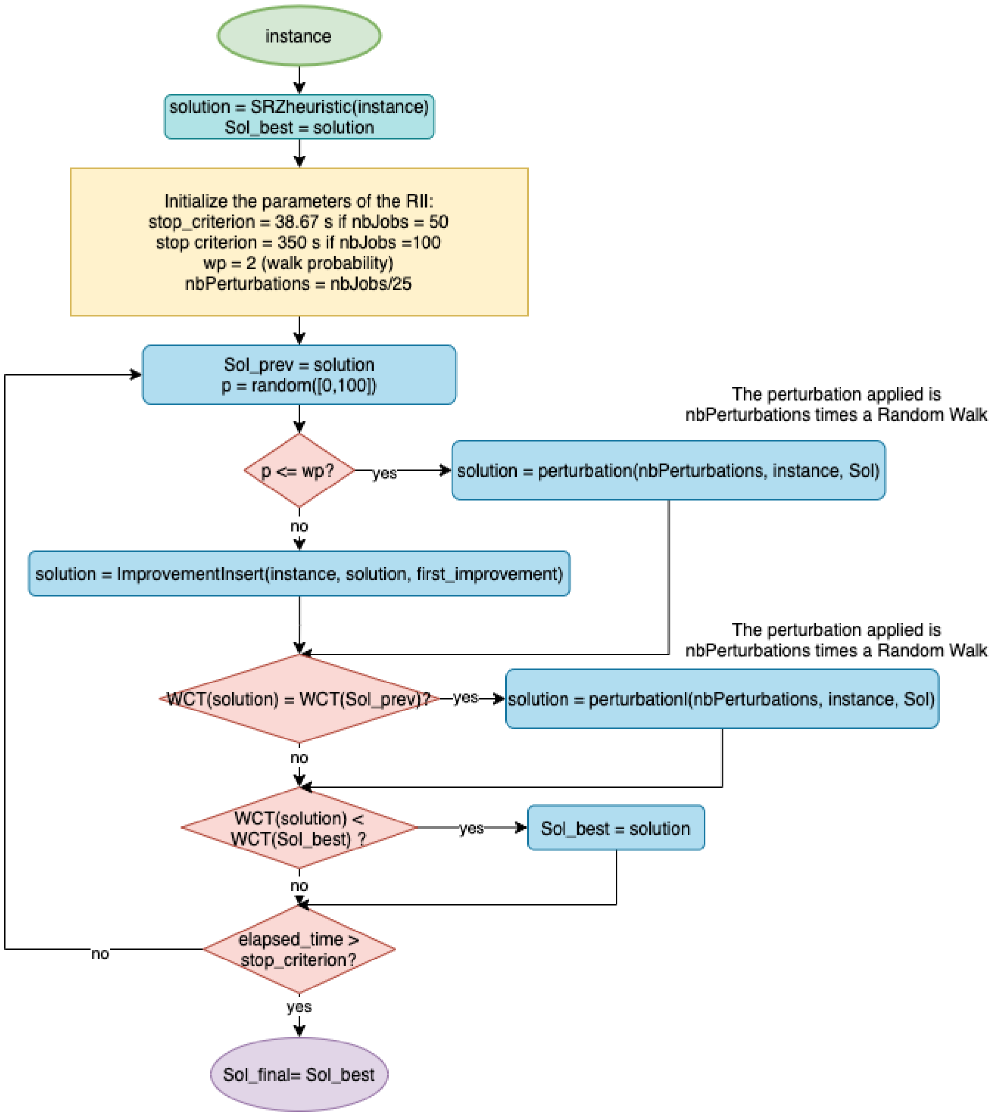
- either the solution remained the same because we're stuck in a local minima (detected by comparing *WCT(solution)* with *WCT(Sol_prev)*)
- or the solution has improved
- or the solution has got worse

The first outcome is tackled by forcing a perturbation on the solution with the hope to escape the local minima. Without forcing this, we endure the risk of staying stuck in the local minima for several rounds since *wp* is low. Forcing directly to apply a perturbation when we detect a local minima allows to explore faster the space of solutions.

For the second outcome, we save the solution as the best solution so far.

For the third outcome, we allow this to happen (it is meant to happen sometimes, by design) and we hope that the next round will improve on this worse solution and escape the potential local minima we were in.

Finally, we verify that the elapsed time doesn't exceed the stop criterion. If it does, we stop the search and the final solution *Sol_final* is our best solution so far *Sol_best.* Otherwise, we do another round.



instance
solution = SRZheuristic(instance)
Sol_best = solution
Initialize the parameters of the RII:
stop_criterion = 38.67 s if nbJobs = 50
stop criterion = 350 s if nbJobs =100
wp = 2 (walk probability)
nbPerturbations = nbJobs/25
Sol_prev = solution
p = random([0,100])
p <= wp?
yes
The perturbation applied is nbPerturbations times a Random Walk
solution = perturbation(nbPerturbations, instance, Sol)
no
solution = ImprovementInsert(instance, solution, first_improvement)
WCT(solution) = WCT(Sol_prev)?
yes
The perturbation applied is nbPerturbations times a Random Walk
solution = perturbationI(nbPerturbations, instance, Sol)
no
WCT(solution) < WCT(Sol_best) ?
yes
Sol_best = solution
no
elapsed_time > stop_criterion?
no
yes
Sol_final= Sol_best

Figure 2: Flowchart of my RII algorithm

Discussion of the parameters chosen for the RII:

stop_criterion: It was asked to stop the search after a certain amount of time, depending on the performance of our VND algorithm of the first implementation: As termination criterion, it was asked to use, for each instance, the average computation time it takes to run a full VND (implemented in the first exercise) on instances of the same size (50 or 100) and then multiply this time by 500 to allow for long enough runs of the SLS algorithms. However, the maximum termination criterion is 350s, so any exceeding criterion should be lowered to that value. Given the results of Figure 1, I have :

- $stop\_criterion\_50jobs = 500 * 0.07726 = 38.67s$
- $stop\_criterion\_100jobs = 500 * 1.12063 = 560.315s$ $350s$ so I take 350s as the stopping criterion for instances of 100 jobs

nbPerturbations: I decided to choose *nbPerturbations* proportionnal to the *nbJobs* of the instance. After several tests, I settled for $nbPerturbations = \frac{nbJobs}{25}$, which gave me a good diversification.

wp: After several tests, I settled for $wp = 2$ which, in conjunction with the value chosen for *nbPerturbations*, gave me a good intensification.

Note: *nbPerturbations* and *wp* had to be chosen in order to reach a good compromise between intensification (greedily increasing the solution quality) and diversification (preventing search stagnation by preventing the search process from getting trapped in confined regions). Low values for *wp* augment the intensification strategy since more proportionally more improvements are performed. Since the perturbation I apply when doing Random Walks is relatively strong (2 moves for instances of 50 jobs, and 4 moves for instances of 100 jobs), I had to choose a low *wp*. When choosing a higher *wp*, the search process didn't have enough time to improve sufficiently the solution before a new worsening solution was made, leading to poor results.

2.3 Hybrid SLS Method: Iterative Local Search

The hybrid SLS method I chose to implement is the **Iterative Local Search** (ILS). The initial solution can either be a random initialization or given by an heuristic.
Given the conclusions I drew from the first implementation exercise, I decided to apply a simple SLS method with:

- A simplified SRZ heuristic to generate a good candidate initial solution
- First improvement as pivoting rule
- Insert as neighborhood

Those are the same than the ones I chose for the simple SLS method, so we will be able to compare and assess better the performance of the Simple SLS method compared to the Hybrid SLS Method.

The values for the parameters of my ILS are discussed below but let's firs understand better my implementation. The flowchart representing how my ILS algorithm works and its tuning I've made is presented on Figure 3.

I start with an initial solution given by SRZ heuristic. Then, I apply a local search on this initial solution. Since the resulting solution is our only solution so far, I keep it as the best solution *Sol_best*. I then enter in a while loop which will only stop once the elapsed time exceeds the stopping criterion.

At the beginning of each "round", I save the solution in *Solution_prev* I apply several Random Walks on *Sol* (same perturbation function than for my RII algorithm in Section 2.2) to get *Sol_pert*. I then apply a local search on *Sol_pert* and reach a new optimal solution *Sol_current* which is a new local minima which can either be:

- the same than *Sol_prev*, meaning we're stuck in the same local minimum than previously. In this case, we increment the counter *countStuckLocalMin*. If this counter exceeds 4, then we increment the *adaptativeNbPerturbations* and reset *countStuckLocalMin* to 0. Otherwise, we go to the Metropolis condition but the solution *Sol_current* is sure to be accepted since WCT(sol_current) = WCT(Sol).
- better than *Sol_prev*, in which case we keep it as the best solution *Sol_best*. We then go to the Metropolis condition but the solution *Sol_current* is sure to be accepted since WCT(sol_current) = WCT(Sol).
- worse than *Sol_prev*, in which case we go to the Metropolis condition which will output a probability of accepting this worse solution *Sol_current* according to the following probability function:

$$p_{accept} = \begin{cases} 1, & \text{if } WCT(Sol_{current}) \leq WCT(Sol) \\ \exp\left(\frac{WCT(Sol)-WCT(Sol_{current})}{T}\right), & \text{otherwise} \end{cases}$$

where T = *Temperature*, who determines how likely it is to perform worsening search steps. At low temperatures correspond low probabilities of accepting a worsening step while at high temperatures correspond high probabilities of accepting a worsening step.

After that, if the retained solution *Sol* is different than the previous solution *Sol_prev*, it means we are not stuck in a local minima and we reset the value of *countStuckLocalMin* to 0 and the value of *adaptativeNbPerturbations* to *nbPerturbationStart*.

Finally, we verify that the elapsed time doesn't exceed the stop criterion. If it does, we stop the search and the final solution *Sol_final* is our best solution so far *Sol_best*. Otherwise, we do another round.

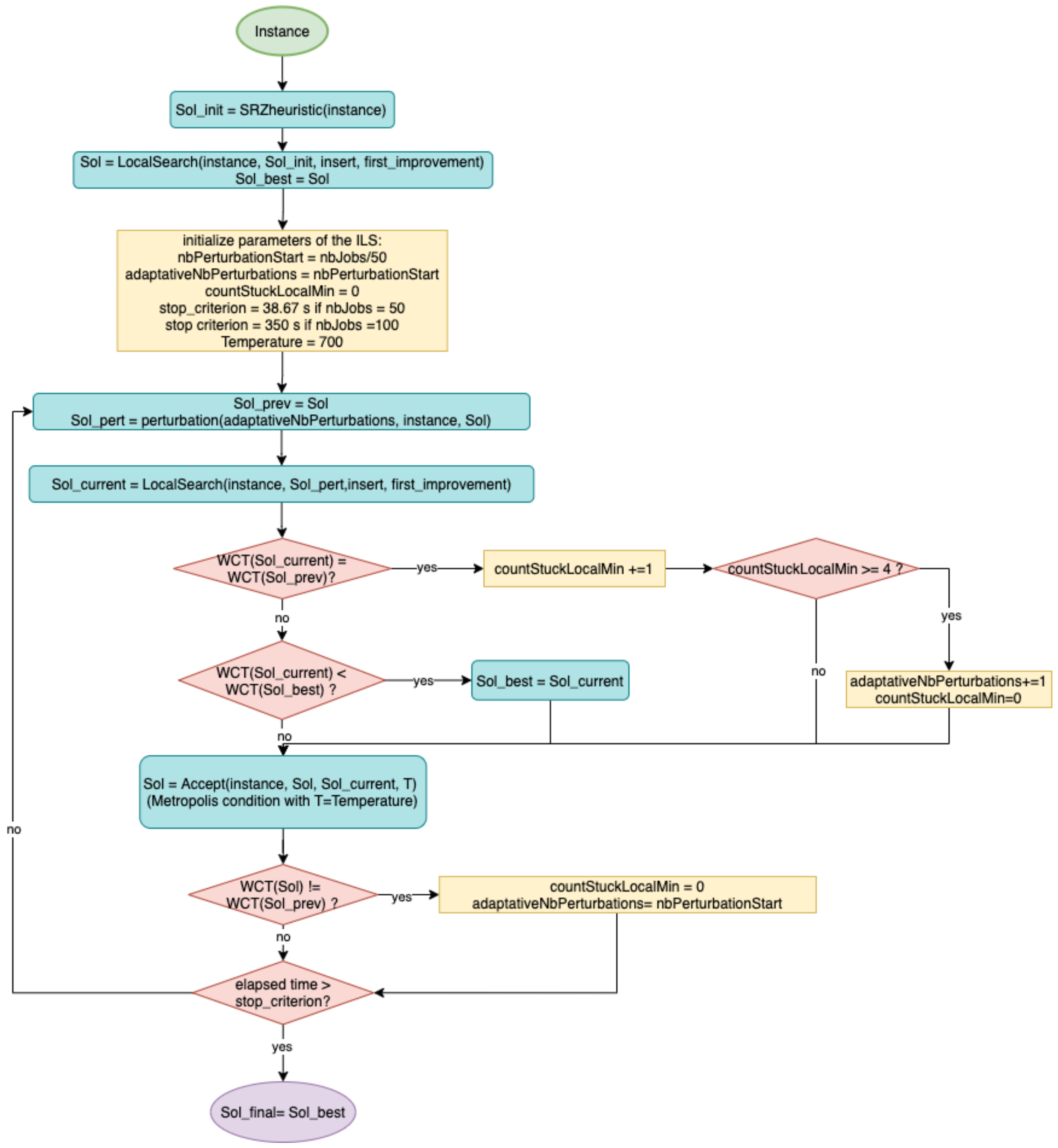

Instance
Sol_init = SRZheuristic(instance)
Sol = LocalSearch(instance, Sol_init, insert, first_improvement)
Sol_best = Sol
initialize parameters of the ILS:
nbPerturbationStart = nbJobs/50
adaptativeNbPerturbations = nbPerturbationStart
countStuckLocalMin = 0
stop_criterion = 38.67 s if nbJobs = 50
stop criterion = 350 s if nbJobs =100
Temperature = 700
Sol_prev = Sol
Sol_pert = perturbation(adaptativeNbPerturbations, instance, Sol)
Sol_current = LocalSearch(instance, Sol_pert,insert, first_improvement)
WCT(Sol_current) = WCT(Sol_prev)?
yes
countStuckLocalMin +=1
countStuckLocalMin >= 4 ?
yes
adaptativeNbPerturbations+=1
countStuckLocalMin=0
no
no
WCT(Sol_current) < WCT(Sol_best) ?
yes
Sol_best = Sol_current
no
Sol = Accept(instance, Sol, Sol_current, T)
(Metropolis condition with T=Temperature)
WCT(Sol) != WCT(Sol_prev) ?
yes
countStuckLocalMin = 0
adaptativeNbPerturbations= nbPerturbationStart
no
elapsed time > stop_criterion?
no
yes
Sol_final= Sol_best

Figure 3: Flowchart of my ILS algorithm

Discussion of the parameters chosen for the ILS:

stop_criterion: Same than for RII, so I have :

- $stop\_criterion\_50jobs = 500 * 0.07726 = 38.67s$
- $stop\_criterion\_100jobs = 500 * 1.12063 = 560.315s$ $350s$ so I take 350s as the stopping criterion for instances of 100 jobs

nbPerturbationStart: I decided to choose *nbPerturbationStart* proportionnal to the *nbJobs* of the instance. After several tests, I settled for $nbPerturbationStart = \frac{nbJobs}{50}$, which gave me a good diversification.

Temperature: After running several tests, I settled for T = 700 which gave me good results. I calculated this value because a difference of -1000 in the WCT gave me a $p_{accept} = \exp(\frac{-1000}{700}) = 0.2396 = 23.96\%$ while a difference of -3000 in the WCT gave me a $p_{accept} = \exp(\frac{-3000}{700}) = 0.0137 = 1.37\%$ which seemed reasonable to me.

2.4 Average RPD from the best known solutions and average RPD across all instances for both SLS methods

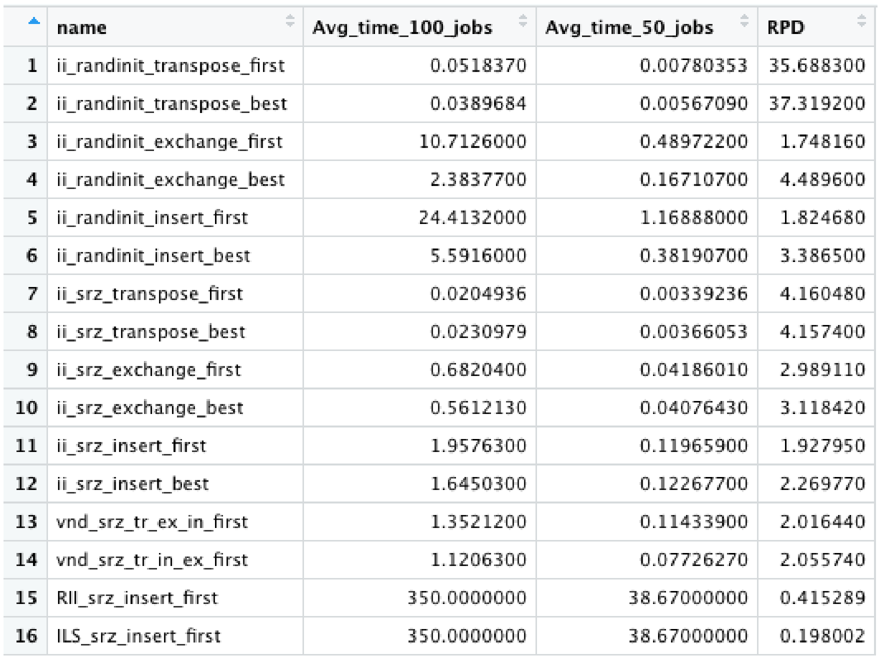
The .txt files containining the lists (in the CSV format) of all the RPD calculated for each instance, calculated 5 times each, and their mean can be found in my zip file of my project in the folder *resultsfiles*. Their names are "RII_srz_insert_firstNEW_FINAL_REPORT.txt", "RII_srz_insert_first_performance_FINAL_REPORT.txt" , "ILS_srz_insert_firstNEW_FINAL_REPORT.txt" , and "ILS_srz_insert_first_performance_FINAL_REPORT.txt". The table 1 resumes all the average RPD from the best known solutions for instances of the same size (50 jobs or 100 jobs) and across all instances for both SLS algorithms.

Instances	RII	ILS
50 jobs instances	0.25503%	0.137438%
100 jobs instances	0.575547%	0.258566%
All instances	0.415289%	0.198002%

Table 1: Average Relative Percentage Deviation from the best known solutions

As we can observe in Table 1, both the RII and the ILS perform well on all the instances. They perform even better on the smallest instances (50 jobs) than on the biggest instances (100 jobs).
Between the 2 SLS methods implemented, the ILS is the best and reaches an average RPD of 0.198% on all the instances. In fact, as can be seen in my results file "ILS_srz_insert_firstNEW_FINAL_REPORT.txt", I sometimes found better solutions than the ones provided in the file "bestSolutions.txt".

Figure 4 presents the updated table of all the performances for all algorithms.



	name	Avg_time_100_jobs	Avg_time_50_jobs	RPD
1	ii_randinit_transpose_first	0.0518370	0.00780353	35.688300
2	ii_randinit_transpose_best	0.0389684	0.00567090	37.319200
3	ii_randinit_exchange_first	10.7126000	0.48972200	1.748160
4	ii_randinit_exchange_best	2.3837700	0.16710700	4.489600
5	ii_randinit_insert_first	24.4132000	1.16888000	1.824680
6	ii_randinit_insert_best	5.5916000	0.38190700	3.386500
7	ii_srz_transpose_first	0.0204936	0.00339236	4.160480
8	ii_srz_transpose_best	0.0230979	0.00366053	4.157400
9	ii_srz_exchange_first	0.6820400	0.04186010	2.989110
10	ii_srz_exchange_best	0.5612130	0.04076430	3.118420
11	ii_srz_insert_first	1.9576300	0.11965900	1.927950
12	ii_srz_insert_best	1.6450300	0.12267700	2.269770
13	vnd_srz_tr_ex_in_first	1.3521200	0.11433900	2.016440
14	vnd_srz_tr_in_ex_first	1.1206300	0.07726270	2.055740
15	RII_srz_insert_first	350.0000000	38.67000000	0.415289
16	ILS_srz_insert_first	350.0000000	38.67000000	0.198002

Figure 4: Performances (updated with the SLS methods) of all the algorithms. The average times are expressed in seconds and the RPD in %

As we can observe, both SLS methods (RII and ILS) perform significantly better than the iterative improvement algorithms from implementation 1, at the cost of a higher time for the search.

2.5 Correlation plots of the average RPD

In this Section, I investigate whether the 2 algorithms are closely correlated. For that, I'll make a correlation plot, display the linear regression and compute the Spearman's correlation coefficient. It is a statistical measure of the strength of a monotonic relationship between paired data[1]. It is denoted by ρ and can take values between -1 and 1.

The closer ρ is to +1 or -1, the stronger the monotonic relationship is. Here is a guide to interpret the value of ρ[2]:

- 0.00 until 0.19: very weak
- 0.20 until 0.40: weak
- 0.40 until 0.59: moderate
- 0.60 until 0.79: strong
- 0.80 until 1.00: very strong

On Figure 5, we can observe the correlation plot of my 2 SLS algorithms on instances of 50 jobs.

[1]`https://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf`
[2]`https://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf`

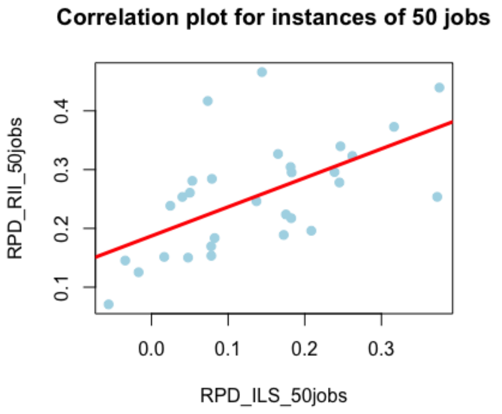


Figure 5: Correlation plot of the average RPD for the 2 algorithm on the 50 jobs instances: RII and ILS

For instances of 50 jobs, the Spearson's correlation coefficient $\rho_{50} = 0.6213571$, which indicates a strong correlation between the 2 SLS algorithms.

On Figure 6, we can observe the correlation plot of my 2 SLS algorithms on instances of 100 jobs.

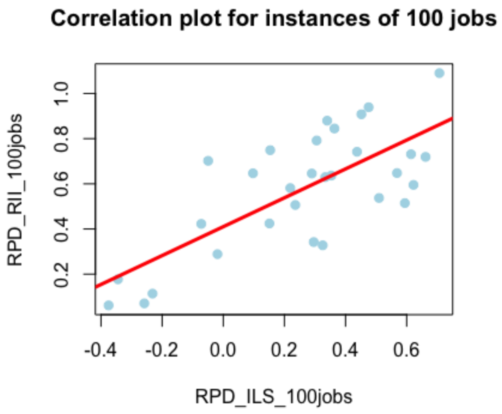


Figure 6: Correlation plot of the average RPD for the 2 algorithm on the 100 jobs instances: RII and ILS

For instances of 100 jobs, the Spearson's correlation coefficient $\rho_{100} = 0.0.6262514$, which also indicates a strong correlation between the 2 algorithms.

I can conclude that for both instance sizes (50 jobs and 100 jobs), the RPDs obtained by the 2 algorithms are strongly correlated, meaning that instances that are difficult for the RII tend to also be difficult for the ILS. However, as can be observed on Figure 5 and 6, but also with the mean values of the RPD found in Table 1, both perform well on all the instances. Indeed, for instances of 50 jobs, for both algorithms, we always get a RPD lower than 0.5% and for instances of 100 jobs, for both algorithms, we almost always get a RPD lower than 1%.

2.5.1 Statistical tests

It is then asked to determine by means of statistical tests whether there is a statistically significant difference between the RPD reached by the 2 algorithms for each instance size. If the two sets are statistically different, it will be relevant to compare the 2 performance of the 2 SLS methods between each other.

To assess it, the Wilcoxon test will done.

Wilcoxon test
The Wilcoxon test is a non-parametric statistical tests that allows to compare 2 groups. Given 2 sets, it computes the difference between the sets of pairs. With this results, it establishes whether the two sets are statistically different from each other by testing the null hypothesis.
The test outputs a p value which represents the probability that the null hypothesis is incorrectly rejected. If this value is lower than the the significance level α chosen, then the statistical test negated the null hypothesis and the two sets are said to be statistically different.
The value chosen for α is 0.05 so $p_{value} < \alpha$ is needed if I want to compare to results.
Statistical results

- **RII_srz_insert_first vs ILS_srz_insert_first for instances of 50 jobs**
  Result of the statistical test: $p_{value} = 4.712492e - 07$ which means that the 2 sets are statistically different.

- **RII_srz_insert_first vs ILS_srz_insert_first for instances of 100 jobs**
  Result of the statistical test: $p_{value} = 4.656613e - 08$ which means that the 2 sets are statistically different.

My statistical tests concluded that the 2 sets are statistically different for both instances sizes (50 jobs and 100 jobs). It makes thus sense to compare them. Reading the results of Table 1 and the correlation plots on Figure 5 and 6, I can conclude that the Iterative Local Search method is better than the Random Iterative Improvement method. Indeed, for instances of 50 jobs, the ILS outperforms on average the RII by a factor of $\frac{0.25503}{0.13744} = 1.85556$. For instances of 50 jobs, the ILS outperforms on average the RII by a factor of $\frac{0.575547}{0.258566} = 2.226$.

3 A few words on the implementation & how to compile

I implemented this project in C++, complementing the basic C++ files I received with the instructions. Although written in C++, I did not make use extensively of the object oriented programming.
Comments inside the code explain the behaviour of the different functions and instructions. Some function names are self-explanatory.

For the implementation 1, a README file ”README_imp1” is present and explains how to compile the program, and what to put in the command line in order to execute the program.
I insist on the fact that the user can either launch the program on ONE algorithm (on ALL the instances) or launch the program on ALL the algorithms (on ALL the instances).
For the implementation 2, a README file ”README_imp2” is present and explains how to compile the program, and what to put in the command line in order to execute the program.
I insist on the fact that the user can only launch an algorithm on 5 times on ALL the instances, and not only one one specific instance.

After launching the program, it will ouput results in the terminal but will also create results txt files written in the CSV format in the folder 'resultsfiles'. Please note that the results files of implementation 1 are not there anymore, with the exception of the results stored in ”all_performances.txt”. For them, please refer to my first implementation zip file.

Please note that:

- The txt file ’all_performances.txt’ inside the folder ’resultsfiles’ is not generated automatically. I manually gathered all the results from the results files (that are generated automatically).
- The R files ”wilcoxon_test.R” (for implementation 1) and ”statistical_tests_imp2.R” (for implementation 2) are not located in the src folder but in the folder ’resultsfiles’ instead in order to avoid path problems.

Please refer to the README file for the specific instructions on how to compile and for further details.

4 Conclusion

The **first implementation exercise** offered us a chance to get hands on with heuristic optimization with the task of implementing an iterative improvement and a variable neighborhood descent algorithm for the PFSP with the weighted sum of completion times objective. Instances of 50 jobs and 100 jobs were provided, as well as the best known solutions to those instances so that we could assess the quality of our solutions. I successfully implemented the those algorithms and, after conducting statistical tests and comparisons between the different algorithms, here are the conclusions I drew for the iterative improvement algorithms:

- To generate an initial solution, the Simplified RZ heuristic (SRZ) was preferable as it guaranteed to lead towards a relatively good solution.
- When using SRZ to generate an initial solution, the first-improvement pivoting rule lead to better results with the same speed than best-improvement.
- When using SRZ to generate an initial solution and first improvement as a pivoting rule, the insert neighborhood was best but took the most time to compute.

Here are the conclusions I drew for the Variable Neighborhood Descent algorithms:

- When using the SRZ to generate an initial solution and the first improvement as a pivoting rule, the VND algorithm performed better than a local search in a single neighborhood ”transpose” or ”exchange” as it got a better RPD. However , it lead to statistically similar solutions than a local search in a single neighborhood ”insert”.
- When using the SRZ to generate an initial solution and the first improvement as a pivoting rule, the ordering of the neighborhoods transpose-exchange-insert or transpose-insert-exchange lead to statistically similar results, so none was preferable above the other in terms of objective function. However, it is worth noting that the order transpose-insert-exchange was faster to compute.

This **second implementation exercise** was built on top of the constructive and perturbative local search methods from the first implementation exercise. I implemented two SLS algorithms: Random Iterative Improvement and Iterative Local Search. The main problem of the algorithms written in the first implementation is that, although some were fast to compute), they only allowed us to converge until the first local minima encountered. VND algorithms could already escape one local minima by trying another neighborhood improvement but once it was stuck with the 3 neighborhoods implemented, the algortihm stopped. Both SLS methods implemented in this second implementation exercise introduced perturbations to try to escape the local minimas and converge towards better solutions. The results were greatly improved, with ILS showing the best mean RPD (0.137% on instances of 50 jobs, 0.258% on instances of 100 jobs) of all algorithms.

Correlation plots allowed us to conclude that the performance of the 2 algorithms are strongly correlated, meaning that when an instance is "difficult" for the RII, it tends to also be "difficult" for the ILS.
Last but not least, the statistical tests confirms that the performance (in terms of RPD) of both algorithms for both instances sizes (50 jobs and 100 jobs) are are statistically different, meaning that it makes sense to compare the 2 SLS algorithms between each other.

The Figure 4 resumes all the performances of all the algorithms in terms of RPD and average time to compute.

As a whole, we discovered that heuristics can help speed up the local search to get a great result (leading to a solution with a low RPD) by starting from a relatively good initial solution and not a random one (which most likely is a bad one), that different neighborhoods/pivoting rules lead to different final solutions. However, if no perturbations are introduced, an algorithm quickly converges until a local minima. The 2 SLS methods I implemented introduced perturbations in order to escape those local minimas and search for a new local minima, potentially better. Some local minima sometimes form large "basins of attraction" and it is sometimes difficult to escape from them. An optimization engineer should thus make good compromises between diversification and intensification. Finally, an engineer should make intelligent decisions to get great results but can also compromise between quality (good RPD) and efficiency (computation time), depending on the constraints he faces.