Heuristic Optimization
Implementation Exercise 1

Alberto Parravicini

1 How to compile and run the code

1.1 *Prerequisites*

- **Cmake**, **make**, a **C++11** compiler.
- **Armadillo**, a linear algebra library. How to install it:
  ```
  sudo apt-get install liblapack-dev
  sudo apt-get install libblas-dev
  sudo apt-get install libboost-dev

  sudo apt-get install libarmadillo-dev
  ```

1.2 *Compilation*

- Move to the *build* folder
  ```
  cd build
  ```
- Run *cmake*
  ```
  cmake ../src
  ```
- Run *make*
  ```
  make
  ```

The executable will be located in the *build* folder, and is called **flowshop**.

1.3 *How to run the program*

The generic syntax to run the program is

```
./flowshop
  --filename ../instances/instance_name
  --algorithm [ii|vnd]
  --random_seed 42
  --neighbourhood_function [t|e|i]
  --initial_state_function [random|rz]
  --use_best_improvement [0|1]
  --neigh_vector [tei|tie]
```

- `--filename`, `-f`: path to the instance file to be used.
- `--algorithm`, `-a`: algorithm to be used, either **ii**, *Iterative improvement* (default), or **vnd**, *Variable neighbour descent*.
- `--random_seed`, `-r`: integer number, used as seed for random number generation by the program. If omitted, the seed is randomized.
- `--neighbourhood_function`, `-n`: how the neighbours of a given candidate solution are computed, either **t**, *Transpose* (default), **e**, *Exchange*, or **i**, *Insert*.
- `--initial_state_function`, `-i`: how the initial candidate solution is computed, either **random**, *randomly*, or **rz**, using the *Simplified RZ heuristic* (default).
- `--use_best_improvement`, `-b`: either **0** or **1** (default), set if the *iterative improvement* algorithm should use *First improvement* or *Best improvement*. Ignored if the algorithm is set to **vnd**.
- `--neigh_vector`, `-v`: set the sequence of neighbourhood functions to be used by the **vnd** algorithm, either **tei**, *Transpose, Exchange, Insert* (default) or **tie**, *Transpose, Insert, Exchange*. Ignored if the algorithm is set to **ii**.

Example:

```
./flowshop -f ../instances/50_20_10 -a ii --use_best_improvement=1 -i rz
```

The algorithm will solve the instance **50_20_10** using *Best improvement iterative search*, with a random seed, *Transpose* neighbour function, and *RZ heuristic*.

2 Introduction to the PFSP

In the *Permutation Flow Shop Scheduling* (**PFSP**) we are given:

- A set of n jobs $J_1, \dots, J_n$.
- A set of m machines $M_1 \dots, M_m$.
- Each job J_i is composed of m different steps, $o_{i1}, \dots, o_{im}$, and each step must be executed on a different machine.
- Each step o_{ij} has a processing time p_{ij} associated to it.
- All jobs pass through the machines in the same order.
- A machine can process at most 1 job at a time.
- Each job has a weight w_i associated to it, which represents its importance.

Assume that the jobs run in the order $j_1, \dots, j_n$, and go through the machines in order $M_1, \dots, M_m$. The completion time of the ith operation of job j_k will be given by

$$C_{i,j_k} = \begin{cases} \sum_{l=i}^{i} p_{l,j_1} & if\ j_k = j_1 \text{ (first job)} \\ \sum_{l=1}^{k} p_{1,j_l} & if\ i = 1 \text{ (first machine)} \\ max(C_{i-1,j_k}, C_{i,j_{k-1}}) + p_{i,j_k} & \text{otherwise} \end{cases}$$

In short, the completion time of the first job will be given by the sum of the processing times of each of its steps (as j_1 will always find all the machines available).
Also, a new job can be processed by the first machine as soon as this is available, hence the second condition.

Our goal is to find the optimal scheduling for the jobs, such that the **weighted completion time** is minimized.

$$min\ WCT = \sum_{i=1}^{n} w_i \cdot C_{m,j_i}$$

3 Implementation

This section will focus on the **C++** implementation of algorithms to solve **PFSP**. For each class/file, it is provided a brief description.

- `flowshop.cpp`:
main access point to the implementation. The input arguments are handled using **Cxxopts** (https://github.com/jarro2783/cxxopts), which provides GNU-style syntax for the input arguments.
It will also measure the execution time, and write the output to a file.

The central idea of the optimization algorithms implementation is to create a structure which allows the algorithms to operate transparently to the problem they are solving.
In order to do so, the implementation of the problem instance and of the candidate solutions are wrapped into appropriate classes, which expose to the optimization algorithm a set of functions what work transparently to the underlying implementation of the algorithm.

Even though this architecture can look rather complex, and potentially slow down the optimization process, it is also very versatile and can be extended to any sort of optimization or search problem that can be solved by *local search* algorithms.

- `pfsp_state.h, pfsp_state.cpp`:
The *candidate solutions* (also referred to as *states*) are wrapped into a class which allows the optimization algorithms to process the candidates solutions independently from their actual implementation.
The class can also store the value of the state, given by some evaluation function. This could be useful, for example, if we had to store states in a priority queue sorted according to the state value.

- `pfps_problem.h, pfsp_problem.cpp`:
This class works as a wrapper to the actual problem instance.
A *problem* is modelled as an entity having an initial state, an evaluation function which can be applied to the states, and a function to compute the neighbourhood of a given state.
All these functions are actually function pointers, and their implementations can be set at runtime when the problem is instantiated, or even after, if one wants to dynamically update the evaluation function that is being used (as in *dynamic search*), or update how neighbours are generated (as in *variable neighbourhood descent*).

- `support_function.h, support_function.cpp`:
this file provides a series of functions that are used by the optimization algorithms. These functions are passed as input to the optimization algorithms by using

function pointers.
This provide higher flexibility, and makes the algorithms transparent to how candidates solutions are evaluated or generated.

The evaluation function is decomposed in 2 parts: one is exposed to the optimization algorithm, and has a signature which is independent to the problem which is being solved; the other takes as input some parameters specific to an instance; this was done so that the **RZ heuristic** can evaluate partial solutions by re-using the existing code.

Armadillo provides easy-to-use data-structures for matrices and vectors, with a syntax close to *MATLAB*.
This allows for easy manipulations of vectors, and very efficient vectorized operations.

The 3 functions to generate the neighbours of a given candidate solution will return a vector of candidate solutions, which can be inspected by the optimization algorithm.
The functions that compute the initial candidate solution are also provided here. They take a problem instance as input, and output a candidate solution.
It should be noted that all the neighbours of a given state are generated, and they are evaluated only later.
Evaluating them straight away would improve the performances of *Iterative Improvement*: however it will be shown later that the speed-up are very small, if the algorithm is combined with the *RZ heuristic* (which is what would be done in a real-case usage).

• `ii_engine.h, ii_engine.cpp`:
this class holds the implementation of the **iterative improvement** algorithm.
The constructor allows to choose between using *first* or *best* improvement, and to specify a *problem* to be solved, or optimized. The result of the search will be stored inside the class, and accessible to the outside.
It is important to note that the algorithm doesn't directly know how the problem instances or the candidate solutions are implemented, and as such the implementation is very flexible and easily extendible to other problems.

• `vnd_engine.h, vnd_engine.cpp`: this class holds the implementation of the **variable neighbourhood descent** algorithm (*VND*).
The structure of the class is similar to the one of iterative improvement.
In the case of *VND*, we use multiple neighbourhood functions, passed as a vector of function pointers to the class constructor.
As such, it is easily possible to use any combination of neighbourhood functions; moreover, if a single function is given, the algorithm becomes equivalent to *iterative search*. However, it was preferred to keep separate the 2 implementations, for the sake of clarity.

4 Inferential statistical analysis

4.1 *Introduction*

The implemented algorithms offer a wide choice of parameters to be set, in terms of how the initial state is computed, how new candidate solutions are generated, and so on.
As such, it is important to evaluate which combinations offer the best performances, both in terms of **results** (the *weighted completion time* defined above) and **execution time**.

The algorithms were tested on 60 different problem instances: 30 instances with 50 jobs and 30 instances with 100 jobs. All the instances had 20 machines.

2 types of tests were performed:

- Test the *iterative improvement* algorithm, by trying all different combinations of initial state generation (*random* and *RZ*), neighbourhood generation (*transpose*, *exchange*, *insert*) and first/best improvement, for a total of 12 different combinations of parameters.
- Test the *variable neighbourhood descent* algorithm, by trying different combinations of neighbourhood generator functions. The algorithm was tested with *Transpose, Exchange, Insert* and with *Transpose, Insert, Exchange*. In both cases, the algorithm was set to use *first improvement* and the *RZ* heuristic for the starting state.

To have comparable results across the tests on a single problem instance, the **seed** used by the *random number generator* was kept fixed across a single instance.

The tests were performed on the following machine:

- Computer: Microsoft Surface Pro 4
- CPU: Intel Core i5-6300U at 2.4 GHz (clocked at 2.95 Ghz)
- RAM: 4 GB at 1867Mhz

4.2 *Exploratory analysis*

Before starting any statistical test, it is a good idea to visualize the data, so to see if it's immediately possible to notice any interesting structure in the results.

First of all, it is required to separate the data relative to the instances with 50 jobs from the ones with 100 jobs, as the values from the 2 sets are not comparable with each other.

Then, it is possible to plot **boxplots** that display summary statistics of the **execution times** and of the **result values**. Relatively to the optimization results, it is also visualized the boxplot of the **ideal optimization results**, the values that would be achieved by an exact solver.

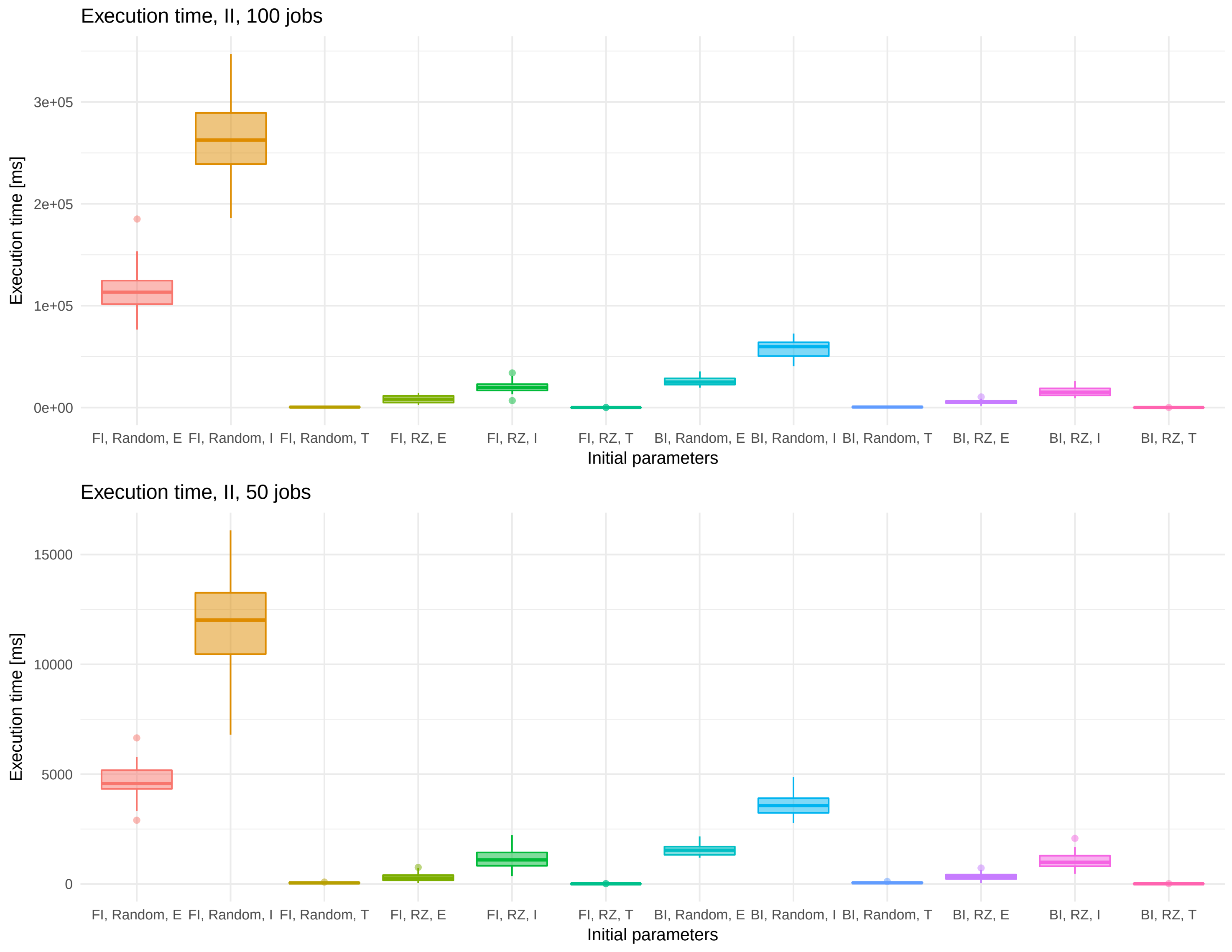

Figure 1: *Summary statistics of the execution time for* ***iterative improvement****, with different parameters.*

By looking at the execution times of **iterative improvement**, it is immediately possible to notice how *first improvement* with *random initial state* and *exchange* or *insert* neighbourhood functions is much slower than the other implementations.

Moreover, it is clear how the *transpose* function is much faster than the other, as the number of generated neighbours is linear with respect to the number of jobs, instead of quadratic.

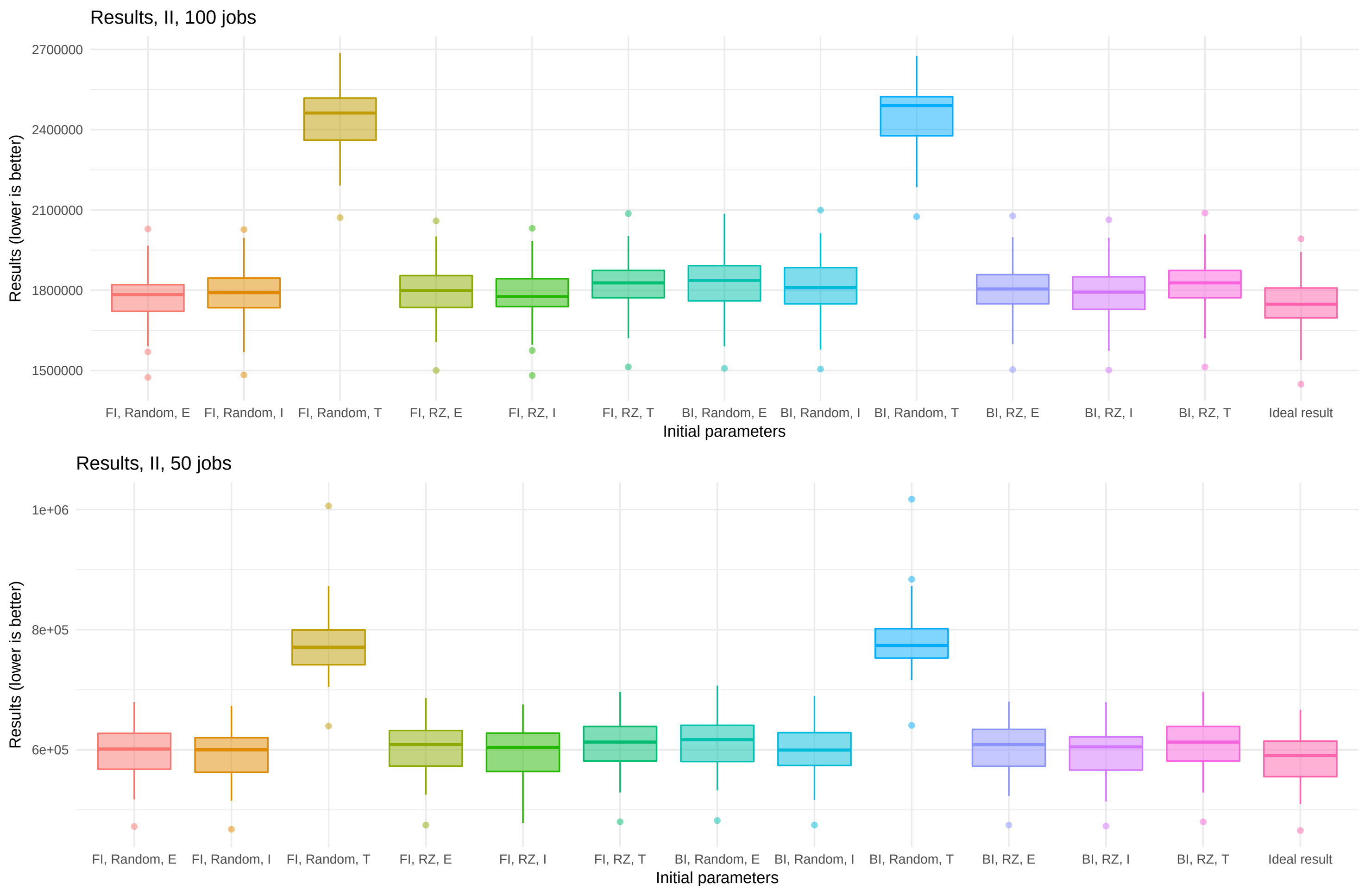

Figure 2: *Summary statistics of the optimization results for* ***iterative improvement****, with different parameters.*

The optimization results given by the algorithms seems to be all very close to each other regardless of the chosen parameters, with the obvious exception of the *transpose* function: despite being faster, it generates a lower amount of new candidate solutions, and it's not surprising to see it perform worse than the other options.
The fact that the results are all quite close to each other means that the choice of initial parameters will be based mostly on execution time, which greatly simplify the problem of picking the "best" algorithm.
Moreover, we can see how the results are seemingly very close to the best theoretical results. We will soon evaluate whether there is a statistical difference between the ideal results and the ones provided by our heuristic algorithms.

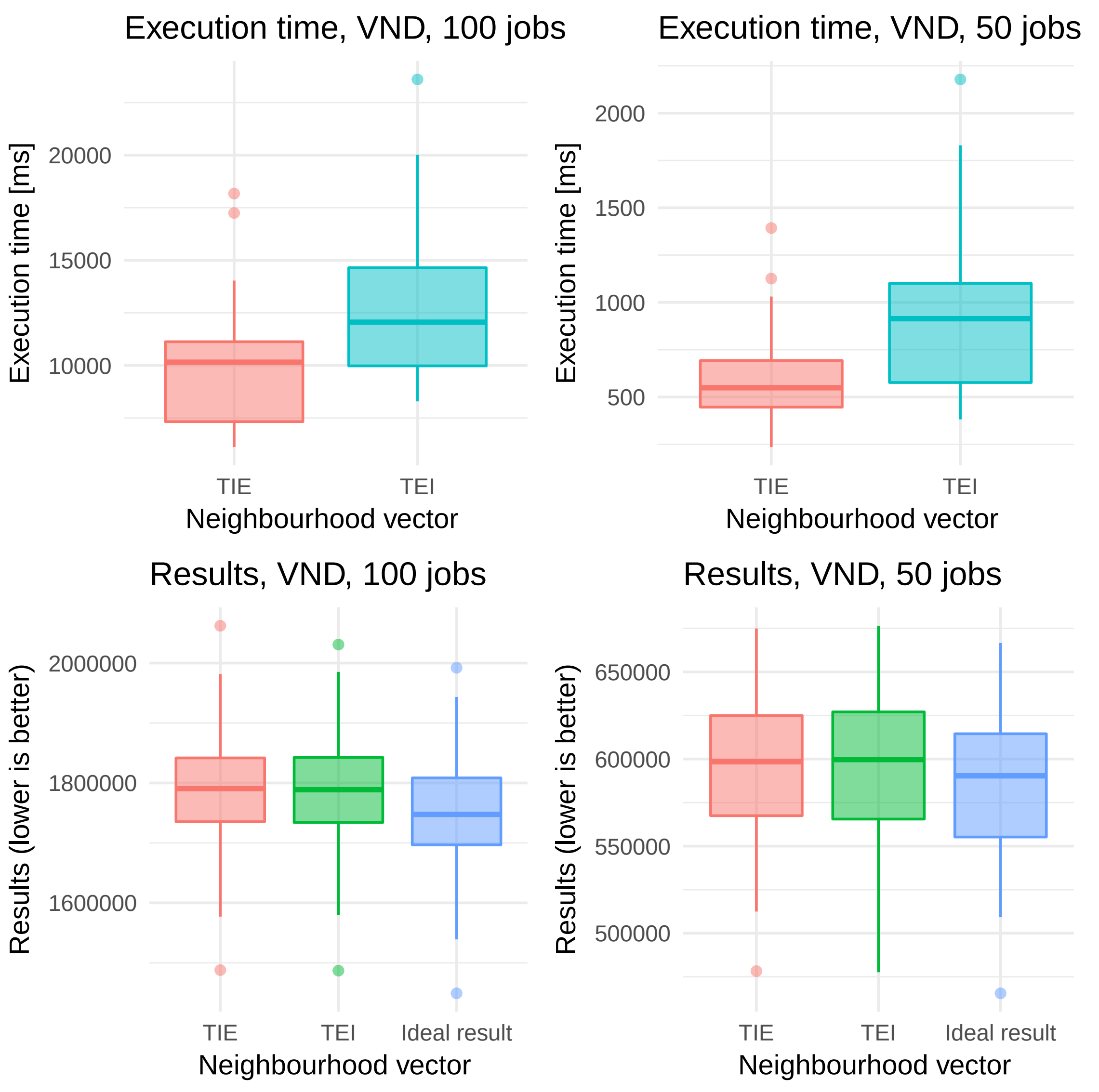

Figure 3: *Summary statistics of execution time and optimization results for* ***variable neighbourhood descent****, with different parameters.*

Relatively to *variable neighbourhood descent*, we can see how **TIE** (*Transpose, Insert, Exchange*) is generally faster than **TEI** (*Transpose, Exchange, Insert*). Both options are however very fast, if compared to the times given by some of the variants of *iterative improvement* algorithms seen above.
As for the results, both options seem to give results that are pretty much equivalent, and once again very close to the theoretical optimum.

From this initial analysis, it seems that the best algorithms could be **iterative search** with *RZ* heuristic and *Exchange* function (using *best* or *first* improvement doesn't seem to change much, with the previous parameters) and **VND** with *Transpose, Insert, Exchange*.
To get a definitive answer, however, more in-depth tests will be required.

4.3 *Summary statistics*

From the collected results, it is possible to compute a number of interesting statistics that will give some additional insight about the performances of the algorithms.
For both algorithms and for each combination of initial settings, it is possible to compute the *mean*, the *median*, the *standard deviation* (and other statistics) relatively to the execution time and to the optimization results.
It is also possible to evaluate how distant the algorithms are from the best possible result.
This can be done by using the **MAPE** (*mean absolute percentage error*), computed as

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{B_i - R_i}{B_i} \right|$$

where n is the number of instances (30 for each number of jobs), B_i is the best possible solution for a given instance, and R_i is the optimization result given by our algorithm.

Initial Parameters	Minimum	Median	Mean	Maximum	Standard deviation
FI, Random, E	76539	113272	112518	185083	22342
FI, Random, I	186265	262628	264573	347115	39546
FI, Random, T	224	438	479	911	169
FI, RZ, E	2461	8192	8037	14341	3642
FI, RZ, I	6834	19631	19934	34009	5576
FI, RZ, T	11	32	34	96	18
BI, Random, E	19544	25040	25956	35471	4474
BI, Random, I	40502	59755	58112	72690	8897
BI, Random, T	267	497	512	898	158
BI, RZ, E	1706	5626	5635	10429	1869
BI, RZ, I	9219	15135	15790	25937	4692
BI, RZ, T	11	42	42	106	17
VND, TIE	6122	10150	10120	18170	3143
VND, TEI	8292	12060	12860	23600	4006

Table 1: Summary statistics of the **execution times** (in milliseconds) of the **iterative improvement** and **VND** algorithms, with different initial parameters.

Initial Parameters	Minimum	Median	Mean	Maximum	Standard deviation
Ideal result	1449000	1754000	1745000	1992000	109155
FI, Random, E	1474000	1783000	1773000	2029000	110586
FI, Random, I	1484000	1791000	1782000	2027000	114707
FI, Random, T	2071000	2462000	2430000	2687000	145989
FI, RZ, E	1501000	1799000	1799000	2059000	111644
FI, RZ, I	1482000	1776000	1782000	2032000	111849
FI, RZ, T	1513000	1827000	1824000	2087000	114101
BI, Random, E	1508000	1837000	1829000	2086000	119963
BI, Random, I	1506000	1810000	1808000	2099000	120916
BI, Random, T	2075000	2490000	2449000	2675000	148046
BI, RZ, E	1504000	1805000	1805000	2078000	113938
BI, RZ, I	1502000	1793000	1788000	2063000	116300
BI, RZ, T	1513000	1828000	1825000	2088000	114601
VND, TIE	1488000	1791000	1784000	2062000	113651
VND, TEI	1487000	1789000	1782000	2031000	110927

Table 2: Summary statistics of the **optimization results** of the **iterative improvement** and **VND** algorithms, with different initial parameters, and instances with **100 jobs**.

Initial Parameters	Minimum	Median	Mean	Maximum	Standard deviation
FI, Random, E	2900	4571	4692	5177	779
FI, Random, I	6794	12020	11950	13260	2265
FI, Random, T	21.00	45.50	45.17	53.25	13
FI, RZ, E	40.0	265.0	310.5	400.5	200
FI, RZ, I	349.0	1095.0	1177.0	1433.0	505
FI, RZ, T	1.000	3.000	3.967	5.000	3
BI, Random, E	1191	1534	1554	1698	235
BI, Random, I	2772	3563	3630	3902	490
BI, Random, T	30.00	49.50	56.13	70.00	18
BI, RZ, E	37.0	329.0	333.8	425.2	182
BI, RZ, I	462.0	985.5	1051.0	1285.0	363
BI, RZ, T	1.000	4.500	5.167	6.750	3
VND, TIE	236.0	549.0	611.2	1393.0	269
VND, TEI	382.0	914.0	899.2	2178.0	408

Table 3: Summary statistics of the **execution times** (in milliseconds) of the **iterative improvement** and **VND** algorithms, with different initial parameters, and instances with **50 jobs**.

Initial Parameters	Minimum	Median	Mean	Maximum	Standard deviation
Ideal result	465500	590300	586200	666600	44161
FI, Random, E	472200	601300	597800	679600	46274
FI, Random, I	467700	599900	595500	673100	47417
FI, Random, T	639500	770900	779200	1006000	64347
FI, RZ, E	474600	608800	602900	686200	46776
FI, RZ, I	478200	603900	596100	675500	45197
FI, RZ, T	480100	612900	607700	696500	47827
BI, Random, E	482200	616800	610800	706900	47422
BI, Random, I	474700	599600	602600	689700	47814
BI, Random, T	640600	773700	784200	1017000	65121
BI, RZ, E	474600	608600	602500	680200	46325
BI, RZ, I	473000	604800	597900	679000	46515
BI, RZ, T	480100	612900	607900	696500	48075
VND, TIE	478200	598400	597300	674900	45947
VND, TEI	477600	599700	596700	676500	45846

Table 4: Summary statistics of the **optimization results** of the **iterative improvement** and **VND** algorithms, with different initial parameters, and instances with **50 jobs**.

Initial Parameters	100 Jobs	50 Jobs	Mean
FI, Random, E	1.607625	1.9744	1.791012
FI, Random, I	2.019402	1.578662	1.799032
FI, Random, T	39.3606	33.03201	36.1963
FI, RZ, E	3.099908	2.840713	2.97031
FI, RZ, I	2.137591	1.702044	1.919817
FI, RZ, T	4.53555	3.664744	4.100149
BI, Random, E	4.760639	4.20176	4.4812
BI, Random, I	3.573396	2.780781	3.177088
BI, Random, T	40.41302	33.88196	37.14749
BI, RZ, E	3.450056	2.787234	3.118645
BI, RZ, I	2.462845	1.994742	2.228794
BI, RZ, T	4.547392	3.686519	4.116955
VND, TIE	2.210744	1.9007523	2.055748
VND, TEI	2.11388	1.8048572	1.959371

Table 5: **Mean average percentage error** for the various algorithms, computed with respect to the best possible solution.

From the previous tables, it seems that the best results are given by **first improvement** with **random** start and **exchange** or **insert**.
These combinations are however the slowest ones, while **VND** with **transpose, exchange, insert** seems to give similar results at much higher speed. Moreover, a difference of 0.1% might not even be significant from a statistical point of view. We will analyse this more in details in the next section.

4.4 *Inferential statistical tests - Introduction*

From the previous plots and tables it seems quite clear that some algorithms perform better than others.
However, some of these differences might not be *significant* from a statistical point of view, and be instead caused by randomness in the test instances or in the algorithms.

In this section, the previous algorithms are compared to each other, to understand which ones are better in terms of speed and result quality.

Before performing any test, a few considerations about the testing methodology should be made.
First, the instances with 50 and 100 jobs should be treated separately, as they clearly compose different populations.
Moreover, the tests to be performed are heavily dependent on the distributions of the populations that are considered.
Most tests assume that the samples from a given population are *independent and identically distributed* (**I.I.D.**): the first condition can be considered true as all the instances are considered separately, without any of them having influences on the others; the second condition is harder to verify, but can be considered true as long as instances with 50 and 100 jobs are considered separately.

Paired t-test also assumes the populations to be *normally distributed*, and to have the same *variance*. The latter constraint can be removed by using **Welch's t-test**, which is provided by default by **R** [3].
The *normality* of the distribution can be checked in several ways, among which the **Shapiro-Wilk** test is one of the most common. [2] The test assumes normality of the distribution as *null hypothesis*.

If the normality condition isn't verified, it is necessary to resort to *non-parametric* tests, such as the **Wilcoxon signed-rank test** [4] for comparing the mean of two populations, or the **Kruskal-Wallis** test to compare more than two populations [1]. The latter test assumes as null hypothesis that all the populations have the same mean, similarly to **ANOVA**.

One last remark is that using *t-test* to compare more than two populations is generally ill-advised (http://www.stat.berkeley.edu/ mgoldman/Section0402.pdf). One way to overcome the issue is to employ the so-called **Bonferroni correction**, which can however give rather conservative estimates.
A better approach is to initially compare the populations with **ANOVA** or **Kruskal-Wallis**, and then proceed with pairwise tests if necessary.

4.5 *Tests on VND*

We can compare the results of **VND**, and see which combination of initial parameters is superior.

• First, we should check the normality of the distributions.

Algorithm	p-value, Execution time	p-value, Result
VND, TIE, 100 Jobs	0.03005	0.5337
VND, TIE, 50 Jobs	0.00832	0.5934
VND, TEI, 100 Jobs	0.005474	0.4946
VND, TEI, 50 Jobs	0.005752	0.6878
Ideal result, 100 Jobs		0.4312
Ideal result, 50 Jobs		0.6698

Table 6: **p-values** of the **Shapiro-Wilk** test for the various algorithms.

We can safely infer that the optimization results follow a normal distribution, while the execution times do not.
As such, they will be compared using the **Wilcoxon signed-rank test**.

Algorithm	p-value, Execution time	Mean TIE	Mean TEI
VND, 100 Jobs	0.00665	10120	12860
VND, 50 Jobs	0.002688	611.2	899.2

Table 7: **p-values** of the **Wilcoxon signed-rank test** test for the **execution time** of the various algorithms.

From the result we can infer that using **Transpose, Insert, Exchange** is the faster option.

• Then, we can compare the optimization results, keeping into account the optimal results too. As we have 3 populations (for each instance size), which we have verified to be gaussian, we can use the **ANOVA** test.

Algorithm	p-value, Optimization result	Mean TIE	Mean TEI	Mean Ideal
VND, 100 Jobs	0.3259	1784000	1782000	1745000
VND, 50 Jobs	0.5677	597300	596700	586200

Table 8: **p-values** of the **ANOVA test** test for the **optimization results** of the various algorithms.

It seems that there is no statistical difference not only between the 2 algorithms, but also with respect to the optimal solution.
As such, the choice of initial parameters will be based on the **execution speed**; as we have seen, the fastest option is **Transpose, Insert, Exchange**.

References

[1] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[2] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[3] Bernard L Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[4] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.