Heuristic Optimization Implementation Exercise 2

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 (delete its content if it's not empty)

```
cd build
```

- Run *cmake*

```
cmake ../src
```

- Run *make*

```
make
```

The executables will be located in the *build* folder, and are called **flowshop_iga** (for **Iterated Greedy**) and **flowshop_gen** (for the **Memetic Algorithm**).

1.3 *How to run the programs*

The generic syntax to run the **Iterated Greedy Algorithm** is

```
./flowshop_iga
  --filename ../instances/instance_name
  --random_seed 42
  --distr_vec_size 6
  --lambda 2
  --max_time [msec]
  --write_exec_trace [0|1]
```

- `--filename`, `-f`: path to the instance file to be used.
- `--random_seed`, `-r`: integer number, used as seed for random number generation by the program. If omitted, the seed is randomized.
- `--distr_vec_size`, `-d`: how many elements in the candidate solutions should be considered in the *Destruction/Construction* procedure. Should be a value between 0 and the number of jobs (3 is the default).
- `--lambda`, `-l`: parameter that influences the temperature of the algorithm. Lower values of λ will slow the convergence.
- `--max_time`, `-t`: maximum amount of time for which the algorithm should run. It is expressed in *milliseconds*. By default it runs for 300 seconds on 50 jobs instances and for 5000 seconds on 100 jobs instances.
- `--write_exec_trace`, `-e`: if 1, the execution trace of the algorithm (i.e. the best current results at a given amount of time) is written to the file *./results/results_details_iga.csv* (default 0).

Example:

```
./flowshop_iga -f ../instances/50_20_10 -l 2 -d 5
```

IGA will solve the instance **50_20_10** using $\lambda = 2$, with a random seed, maximum time of 300 seconds, and *destruction vector size* of 5; no execution trace is written.

The generic syntax to run the **Memetic Algorithm** is

```
./flowshop_iga
  --filename ../instances/instance_name
  --random_seed 42
  --population_size 10
  --crossover_prob 0.99
  --mutation_prob 0.05
  --weights_type [uni|uni10|sm]
  --mutationtype [tr]
  --max_time [msec]
  --write_exec_trace [0|1]
```

- `--filename`, `-f`: path to the instance file to be used.
- `--random_seed`, `-r`: integer number, used as seed for random number generation by the program. If omitted, the seed is randomized.
- `--population_size`, `-p`: positive integer, it controls the population size of the algorithm (default 10).
- `--crossover_prob`, `-c`: value between 0 and 1. It controls the probability that two candidate solutions are combined together instead of just copying them (default 0.99).
- `--mutation_prob`, `-m`: value between 0 and 1. It controls the probability that a candidate solution is randomly mutated (default 0.05)).
- `--weights_type`, `-w`: it controls the probability that each candidate solution is selected by the algorithm (default `sm`, *Softmax*).
- `--mutationtype`, `-n`: it controls the type of mutation done by the algorithm (default `tr`, *Transpose*).
- `--max_time`, `-t`: same as **IGA**.
- `--write_exec_trace`, `-e`: same as **IGA**. Results are written to the file *./results/results_details_gen.csv* (default 0).

Example:

```
./flowshop_gen -f ../instances/50_20_10 -p 20 -c 0.8
```

The **Memetic Algorithm** will solve the instance **50_20_10** using *population size* $= 2$, with a random seed, maximum time of 300 seconds, *Transpose mutation* with chance 0.05, *crossover rate* of 0.8, and *uniform* weights; no execution trace is written.

2 Overview of the implementation

This section will focus on the **C++** implementation of algorithms to solve **PFSP**. For each class/file, it is provided a brief description. Note that the 2 new algorithms were built on top of the existing structure used in the previous assignment. In this section, the focus will be on new parts that have been added. For a complete overview of the code structure, please refer to the first report.

• `flowshop_iga.cpp`:
main access point to **IGA**.

• `flowshop_gen.cpp`:
main access point to the **Memetic Algorithm**. Using 2 separate files (and so 2 executables) was considered the best option as the 2 algorithm have very different structure and parameters, and keeping a single executable would make things more confusing.

• `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.
The file was updated to contain functions that are used by **IGA** and by the **Memetic Algorithm**. Operators for *Destruction/Construction*, *Crossover* and *Mutation* are contained here. Please refer to the algorithm description for more details about them.

• `iga_engine.h, iga_engine.cpp, gen_engine.h, gen_engine.cpp`:
this class holds the implementation of the **iterated greedy algorithm** and of the **memetic algorithm**.
Their constructors allows to choose the different parameters, 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.
The algorithms don't directly know how the problem instances or the candidate solutions are implemented, and as such the implementations are very flexible and easily extendible to other problems.
The details of the implementations and of the parameters will be given in the next section.

3 Details of the implementation

3.1 *Iterated greedy algorithm*

The Iterated Greedy Algorithm **(IGA)** applies *local search* to states that are iteratively generated through *heuristic procedures*.
At each step, the previous candidate solution is modified through an heuristic procedure called *Destruction/Construction*, and then *Local search* is applied to the result.
The proposed implementation is based on the work of *Pan* and *Ruiz* [2].

The initial state is generated through the **RZ heuristic**, already seen in the first assignment.
The subsidiary local search is **Best Improvement** with Transpose as neighbourhood function: as seen in the first assignment, this algorithm is able to find good solutions very quickly, as long as the initial state is acceptably good.

The **Destruction/Construction** procedure that is adopted is the same that is described in the article: a certain number of elements is randomly removed from the solution, and then they are re-added one-by-one in the position that gives the best partial result.
As for the number of elements that are removed, **10**, **8** and **6** were tested on instances of size 50, with 10 being sightly better in terms of solution quality (refer to the specific section for more details).
Instances of size 100 use a value of 8, as suggested by *Pan* and *Ruiz*.

As explained in *Pan*, *Ruiz* [2], candidate solutions that aren't improving are accepted with a probability that is proportional to their quality. Moreover, this probability is a function of the instance size, and of the parameter λ.
Higher values of λ will increase the acceptance probability, and make the algorithm more prone to *diversification*. Convergence will be slowed down, however.
It was shown that the value of λ doesn't have any significant impact on the performances, and as such a value of $\boldsymbol{\lambda = 2}$ was picked, as suggested by the article.

The algorithm stops after a certain amount of time has elapsed; this value was set equal to 500 times the average execution time of a *Variable Neighbour Descent* algorithm, on similarly sized instances. Concretely, the maximum amount of time on 50 jobs instances is set to **100** seconds, while for 100 jobs instances it is **500** seconds.
However, the algorithm stops if no improvement has been found for a given amount of time, equal to the maximum allowed time divided by 50. Using a longer window doesn't seem to provide significant improvements in terms of

quality, and the chosen value is a good compromise between execution time and result quality.

3.2 *Memetic algorithm*

Memetic algorithms combine *genetic algorithms* with *local search*. First, a **crossover** operator is applied to the initial random population. Then, local search can be applied to the resulting population, followed by a **mutation** operator. This second population is then improved by using again *local search*, and the resulting candidate solutions will be used as population for the next iteration.

This type of algorithm has a large number of parameters that can be considered. As a starting point, some of the parameters were picked according to **Bonarini** lecture notes, slide 45.
The population size was tested with values of 10 and 20, but no significant difference was observed.

Crossover and **Mutation** are done by sampling candidate solutions from the population. Each candidate solution is picked with a probability that is a function of its fitness.
Three different approaches have been tested. In the first, probabilities are simply inversely functional to the fitness; in the second, referred as `uni10`, we consider the inverse of the 10-th power of the scores. The idea is to increase the probabilities relative to good solutions, and penalize the others.
As extreme case, the third approach is to use the *softmax* function, which is the inverse of the exponential of the scores.
Overall, using a simple uniform probability seems to give the best results, probably because it ensures the highest diversification.

Crossover was applied by using the **PMX** operator [4], which makes sure that children candidate solutions are valid permutations. Pairs of candidate solutions are picked from the population accordingly to their fitness, and are combined through crossover with a probability of 0.99; lower values were tested, and they gave significantly worse results.

Mutation is done by swapping 2 random elements in the candidate solution, with a probability of 0.05; higher probabilities seems to excessively disrupt good solutions.

In a single iteration of the *memetic algorithm*, **Best improvement** is applied twice as subsidiary local search procedure: in the first case, after *crossover*, it is used with the **Transpose** operator, to quickly improve the quality of the population without wasting too much time.

After mutation it is applied again with the **Exchange** operator, which is slower but significantly improves the quality of the population.

The termination criterion is implemented in the same way as **IGA**.

4 Inferential statistical analysis

4.1 *Introduction*

The implemented algorithms offer a wide choice of parameters to be set.
Trying all possible combinations would take a huge amount of time, but it is still possible to perform some tests to check how different parameters can impact the performances, both in terms of **results** (*weighted completion time*) 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.
In some tests, where indicated, a smaller subset of instances was used.
For each combination of parameters, the same instance has been tested 5 times, then the *median* execution time and result have been considered.
Compared to the *mean*, the *median* is more robust to outliers, and will give more meaningful results.

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.

Note: contrary to the previous assignment, this time the code has been compiled by using `-O3` level optimization. To provide results that are comparable, the previous algorithms have also been recompiled with the same optimization level; then, **VND** has been tested again, with the same procedure used for the 2 new algorithms, in order to have a meaningful comparison.

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

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. The results obtained by **Variable Neighbour Descent** with Transpose, Insert, Exchange have been also included in the comparisons, as a reference point. In the case of 50 jobs instances, the plots shows also the values for different parameters for **IGA**.

In the first 2 plots we can compare the execution times of **IGA**, **Memetic algorithm** and **VND** on instances of size 50 and 100. On paper, all the algorithms are given the same maximum execution time, but, as said before, an *early stopping* procedure has been adopted, in order to reduce computation time. As such, it is meaningful to compare the execution times, to see how long it takes to the algorithms to reach a stagnation point.

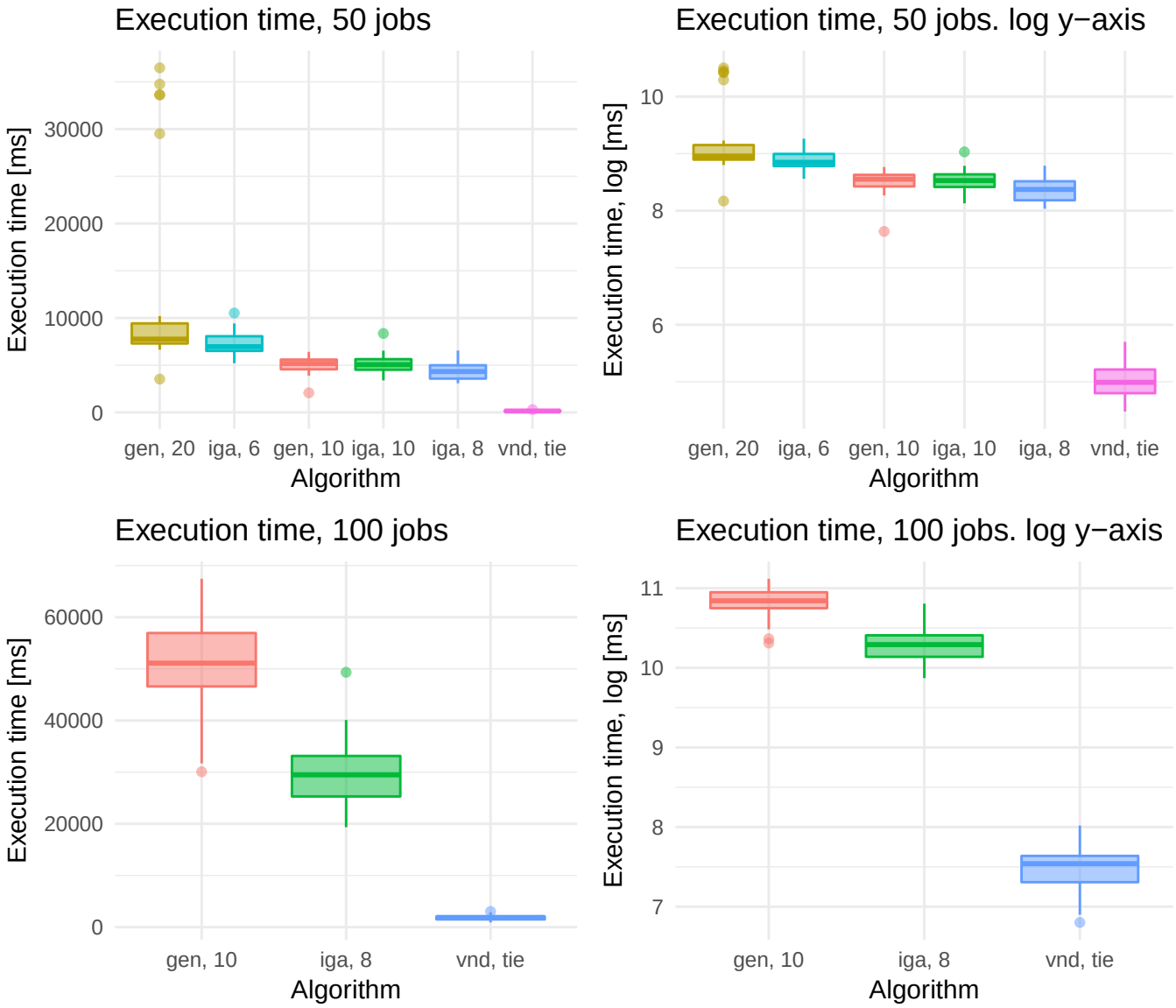


Figure 1: *Box plots of the execution times for the various algorithms.*

We can see how the new algorithms are significantly slower that **VND**, and that modifying 8 or 10 elements in the *destruction/construction* procedure leads a faster convergence to a local optimum. This can be explained considering that the *destruction/construction* procedure is very fast, and modifying more elements doesn't slow down the algorithm; on the other hand, it improves more the

current solution by using the *destruction/construction* heuristic, and causes a faster convergence.

The optimization results are compared by looking at the **ARPD**, *average percentage relative deviation*, defined as:

$$ARPD = \frac{1}{R} \sum_{i=1}^{R} \left(\frac{100 \cdot (S_i - S_{best})}{S_{best}} \right)$$

where R is the number of repetitions of each instance, S_i is the score of a given iteration, and S_{best} is the best known score for the instance (in our case, the optimal solution). Using **ARPD** prevents the results from being biased by instances with unusually high (or low) scores, and gives more reliable results.

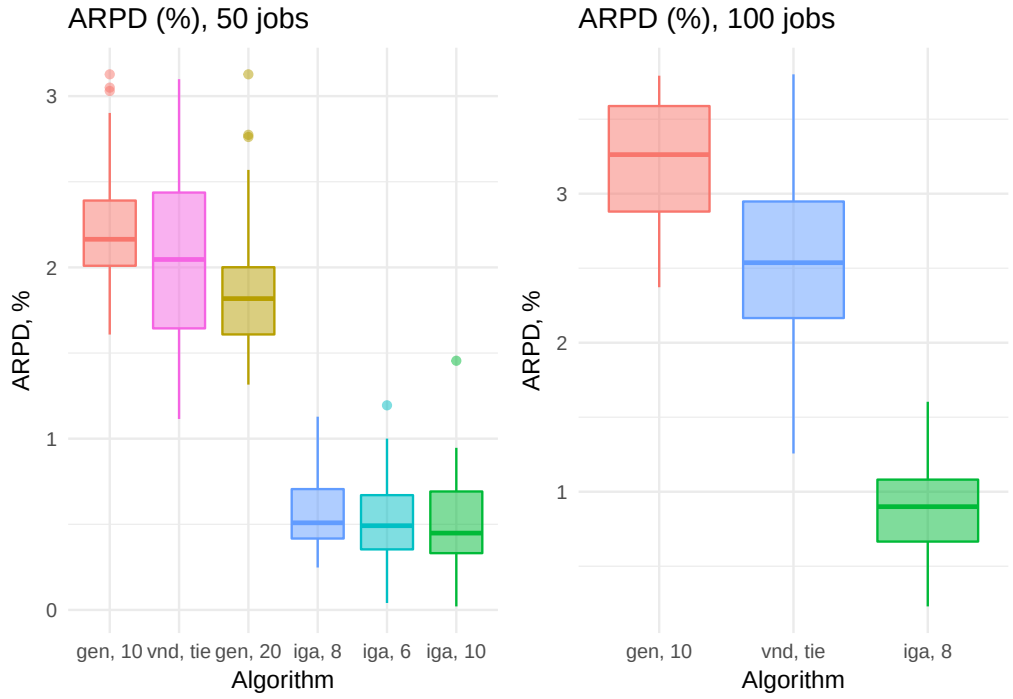


Figure 2: *Summary statistics of the optimization results for the various algorithms.*

IGA shows some improvements over **VND**, at the cost of slower execution time. Note that the population size of the **Memetic algorithm** is different between instances with 50 and 100 jobs, due to computational constraints. This is probably the reason why the algorithm is seemingly better than **VND** for smaller instances, but worse on the bigger ones.

We can also look more in details at the results of **IGA**, when 6, 8 or 10 elements are modified in the *destruction/construction* procedure. These tests were done on instances of size 50.

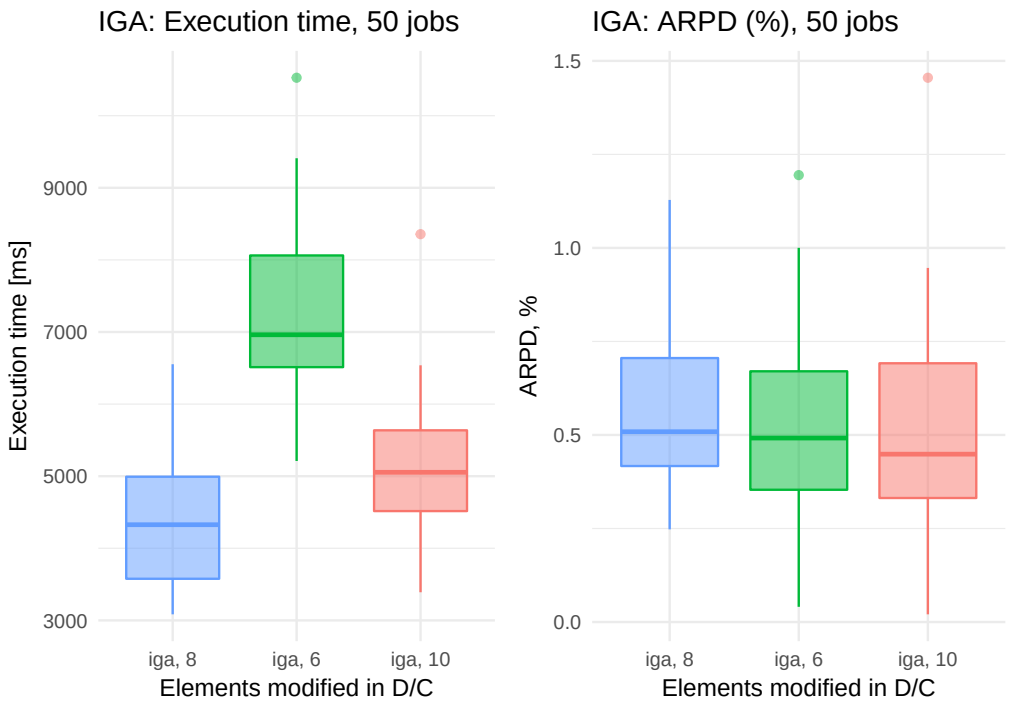


Figure 3: *Results for **IGA** where a different number of elements is modified in destruction/construction.*

A *Kruskal-Wallis* test was performed to check if there is a statistically significant difference between the 3 values.

Type	p.value	Median, 6	Median, 8	Median, 10
Execution time	3.142e-05	6891.00	4615.50	5268.00
ARPD	0.14	0.59	0.52	0.38

As it is possible to reject the *null hypothesis* that the 3 algorithms have the same execution time, we can perform a further test on the fastest 2, and see if they are different.

Type	p.value	Median, 8	Median, 10
Execution time	0.3147	4615.50	5268.00

It is possible to say with good confidence that using a value of 8 gives a faster algorithm, but in terms of solution quality the 3 options are pretty much equivalent. It is also not possible to say that using 10 instead of 8 will give a faster algorithm, in spite of what the plots might suggest. As such, for the remaining

part of the report it will be used a value of 8, accordingly to what was suggested by the article.

We can also compare the results of the *Memetic algorithm* with population size 10 or 20.

Type	p.value	Median, 10	Median, 20
Execution time	3.894e-13	5174.50	7760.50
ARPD	0.0008	2.16	1.82

It can be seen how using a smaller population will give a faster algorithm, but worse results. For instances size of 100 it was preferred to use a population of 10, while on instances of 50 jobs a value of 20 was used, in order to get the best quality/execution-time *tradeoff*.

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 each algorithm, 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 (using **ARPD**).

Algorithm	Min	Mean	Median	Max	Standard Deviation
GEN, 20	3528.00	12028.42	7760.50	36484.50	9926.38
IGA, 8	3081.00	9562.20	4615.50	39951.00	11517.58
VND, tie	88.00	152.93	147.00	299.00	47.64

Table 1: Summary statistics of the **execution time** of the various algorithms over instances with **50 jobs**.

Algorithm	Min	Mean	Median	Max	Standard Deviation
GEN, 10	30079.00	51450.10	51113.00	67448.00	9058.44
IGA, 8	18293.00	31509.20	30382.00	50518.00	6801.02
VND, tie	899.00	1834.03	1882.00	3038.00	534.49

Table 2: Summary statistics of the **execution time** of the various algorithms over instances with **100 jobs**.

Algorithm	Min	Mean	Median	Max	Standard Deviation
GEN, 20	1.32	1.91	1.82	3.13	0.42
IGA, 8	0.07	0.49	0.52	1.09	0.22
VND, tie	1.11	2.07	2.05	3.10	0.50

Table 3: Summary statistics of the **ARPD** of the various algorithms over instances with **50 jobs**.

Algorithm	Min	Mean	Median	Max	Standard Deviation
GEN, 10	2.37	3.20	3.26	3.79	0.41
IGA, 8	0.07	0.91	0.87	1.66	0.43
VND, tie	1.26	2.52	2.54	3.80	0.63

Table 4: Summary statistics of the **ARPD** of the various algorithms over instances with **100 jobs**.

The previous tables confirm what was observed with the plots: **IGA** is the best algorithms in terms of results, but it is a bit slower that **VND**. The **Memetic** algorithm is consistently than the other 2 on instances of size 100.

4.4 *Inferential tests*

It seems obvious that there exist significant differences among the various algorithms; to see if this is actually the case, we will perform statistical tests on the observed results.

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.

If the population aren't normally distributed, it is necessary to resort to *non-parametric* tests, such as the **Wilcoxon signed-rank test** [6] 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**.

Type	p.value	Mean, IGA 8	Mean, GEN 20
Execution time	1.931e-05	9562.20	12028.42
ARPD	2.2e-16	0.49	1.91

Table 5: **p-values** of the **Wilcoxon** test for the various algorithms, relatively to *execution time* and $ARPD$ of 50 jobs instances.

Type	p.value	Mean, IGA 8	Mean, GEN 10
Execution time	1.292e-11	31509.20	51450.10
ARPD	2.2e-16	0.91	3.20

Table 6: **p-values** of the **Wilcoxon** test for the various algorithms, relatively to *execution time* and $ARPD$ of 100 jobs instances.

Unsurprisingly, it is confirmed by the statistical tests that **IGA** is better than the **Memetic algorithm** both in *execution time* and in *result quality*.

4.5 *Correlation plot of solution quality*

A useful way to visualize how the 2 algorithms perform with respect to each other is to draw a **correlation plot**.
On the axes, the **ARPD** (or the **execution time**), of the 2 algorithms.
Each point represents the **ARPD** (or the **execution time**) of the 2 algorithms on the same instance. To get more robust results, each point is computed at the mean of 5 different runs on the same instance.

Them, we can compute the correlation coefficient of the 2 algorithms. The correlation is measured as **Pearson Correlation**.
Then, the correlation value is tested to see if it is statistically significant.

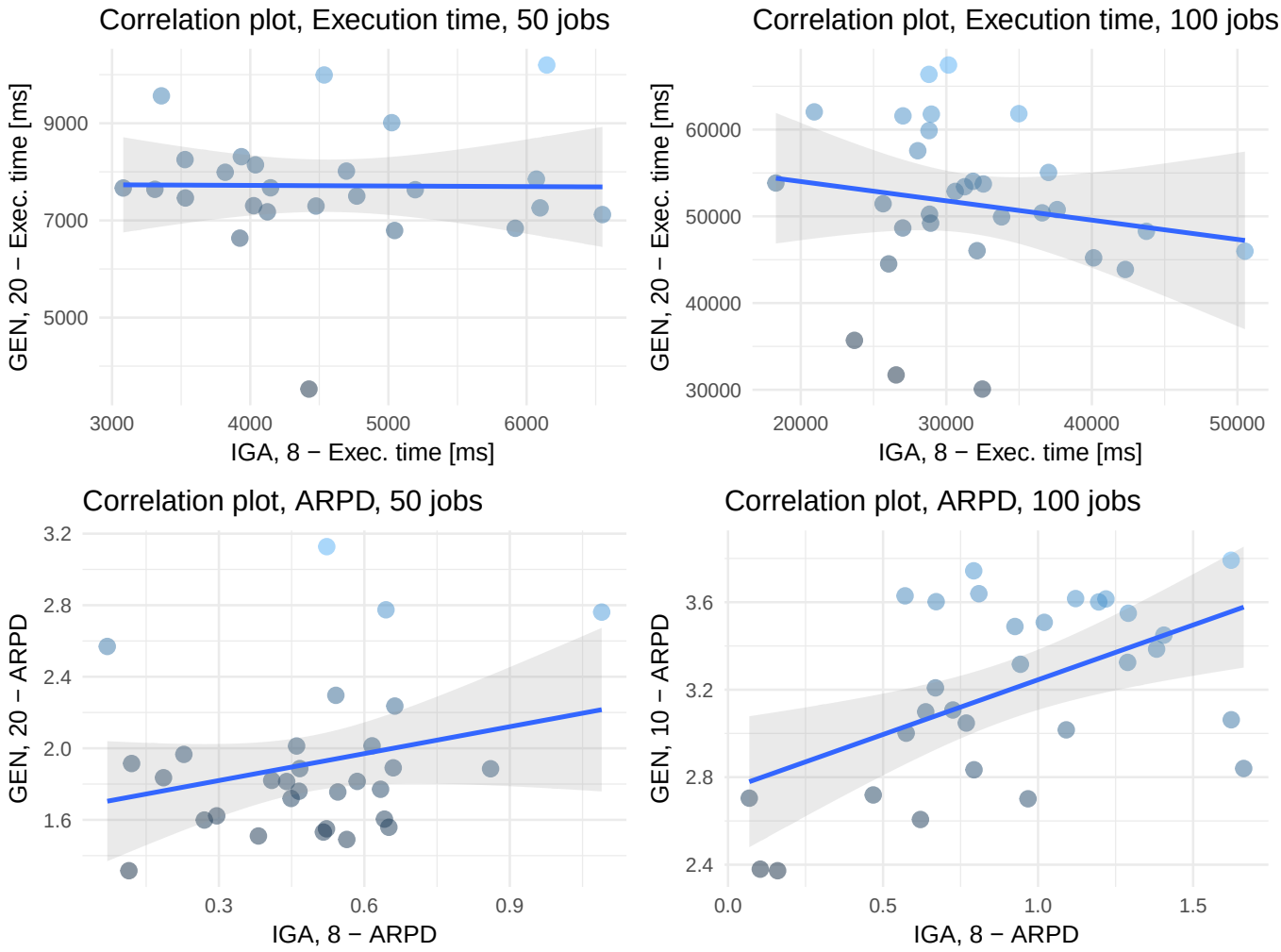


Figure 4: *Correlation plots of the execution times and of the ARPD for the various algorithms.*

Type	Correlation	p.value	Mean, IGA	Mean, GEN
Execution time, 50	-0.01	0.96	4550.76	7714.12
ARPD, 50	0.26	0.16	0.49	1.91
Execution time, 100	-0.17	0.38	31509.20	51450.10
ARPD, 100	0.52	0.003	0.91	3.20

Table 7: **Pearson correlation** and **p-values** of the execution times and of the ARPD for the various algorithms.

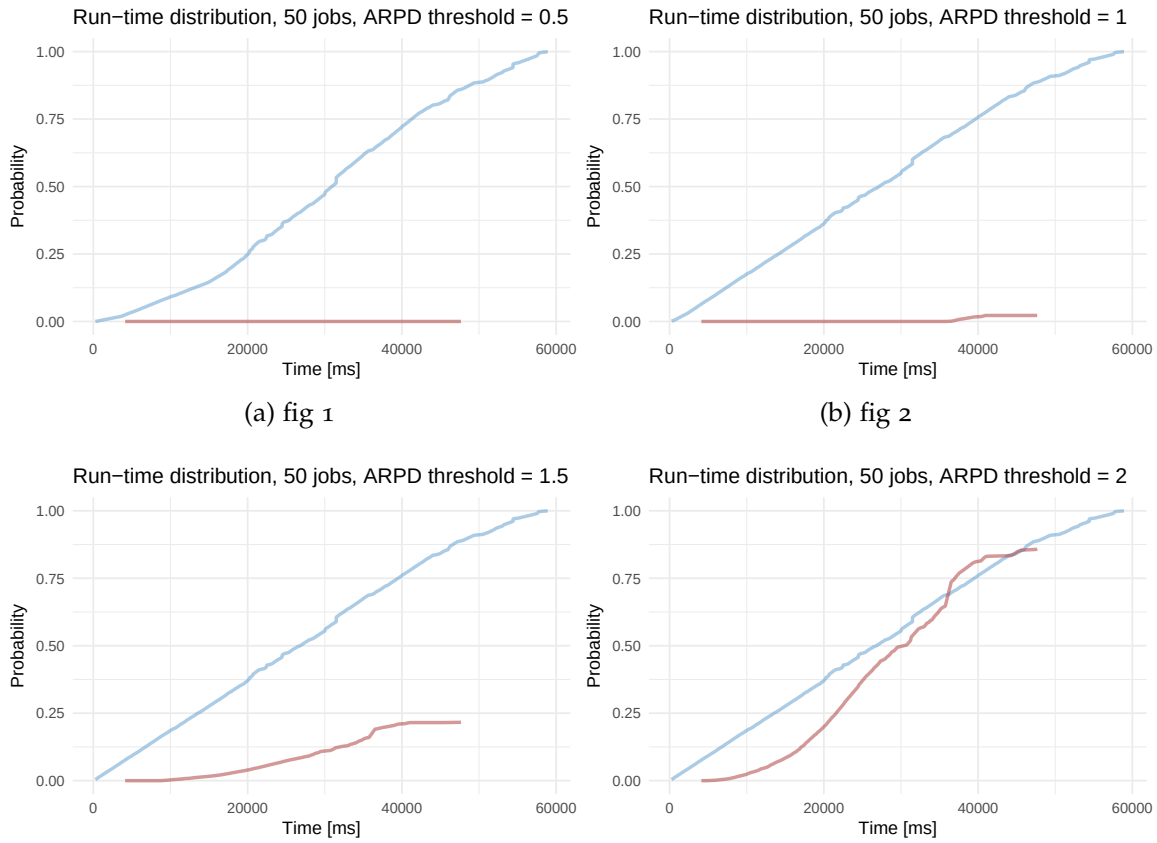
The only case in which there is a statistically significant correlation is for the **ARPD** of 100 jobs instances. In all the other cases, it seems that the **Memetic algorithm** is on average slower, but is more consistent in terms of results and execution time.

4.6 *Run-Time Distributions*

A way to compare the behaviour of different algorithms is to look at the **run-time distribution**, which can be seen as the function of solution quality over time. More formally, it is a function $P(t) : \mathbb{R}^+ \rightarrow [0, 1]$ such that $P(\bar{t})$ is the probability to find the optimal solution (or some very good solution used as reference point) at time $\bar{t}$.

The run-time distributions were computed by running the first 5 instances with 50 jobs, 25 times each. For each of them, the performance over time were recorded by writing the execution trace: to approximate the behaviour of the algorithm over time, it was recorded the solution quality every 100 steps of the execution. The quality of the approximation can be seen by noticing at how to similar amounts of steps correspond similar elapsed times.
Then the values for each instance have been averaged, to produce a smoother curve over time.

The comparison is rather difficult to make, as **IGA** performs much better than the **Memetic Algorithm**.
It was measured the percentage of times that at a certain time point the solution quality is close than the optimum, where close means that the **ARPD** is smaller than a certain threshold.
The values of **ARPD** that were considered are $\{0.5\%, 1\%, 2\%\}$.



(a) fig 1

(b) fig 2

(c) fig 3

(d) fig 4

Figure 5: Run time distribution with different thresholds. **IGA** is blue, **Memetic** is red.

We can see once again how **IGA** works better than the **Memetic algorithm**. For small **ARPD** thresholds, the **Memetic algorithm** often has a probability close to zero of giving solutions better than the threshold, even for high execution times. Only for **ARPD** $= 2$ the **Memetic algorithm** seems to be comparable with **IGA**, and it is even sightly better for high execution times.

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] Quan-Ke Pan and Rubén Ruiz. Local search methods for the flowshop scheduling problem with flowtime minimization. *European Journal of Operational Research*, 222(1):31–43, 2012.

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

[4] Göktürk Üçoluk. Genetic algorithm solution of the tsp avoiding special crossover and mutation. *Intelligent Automation & Soft Computing*, 8(3):265–272, 2002.

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

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