Heuristic Optimization : Implementation exercise 2

Nikita Marchant (nimarcha@vub.ac.be)

May 17, 2017

1 Introduction

This document is the report of the implementation exercise 2 for the Heuristic Optimization course. The exercise asks to implement two stochastic local search algorithms for the the permutation flow-shop problem with the sum weighted completion time objective (also called PFSP-WCT).

The PFSP-WCT has not been thoroughly studied in the literature so i used some inspiration from papers studying the PFSP with flowtime that is more studied.

2 Algorithms

The first algorithm is an Iterated Local Search (ILS) inspired from [Pan and Ruiz, 2012] and the second is a genetic algorithm inspired from [Zhang et al., 2009].

2.1 Iterated Local Search

The Iterated Local Search is a stochastic meta-heuristic known to be applicable to multiple optimization problems [Gendreau and Potvin, 2010].

It consists of these steps :

1. Generate an initial solution
2. Apply a local search to the solution
3. When stuck in a local optimum, apply a perturbation
4. Apply a local search to the solution
5. Use an acceptance criterion for the perturbed then optimized solution
6. Go back to 3 or stop when a termination criterion is met.

2.1.1 Initialization method: $LR(x)$

The initialization method used in this implementation and in [Pan and Ruiz, 2012] is the $LR(x)$ heuristic introduced by [Liu and Reeves, 2001]. It consist in three steps :

1. Rank the jobs by their weighed sum of flowtime
2. Generate x solutions by inserting the job from position x at the front of the solution
3. Select the sequence with the minimum weighted flow time

2.1.2 Local search

The local search that was implemented is the iterated RZ (IRZ). The RZ uses a insertion neighborhood. It sequentially inserts each job at each possible position in the candidate solution (thus is in $O(n^2)$ complexity if n is the number of jobs) and keeps the best one.

The IRZ applies RZ until a local optimal solution is found (i.e. the RZ does not yield a better result).

2.1.3 Perturbation method

After finding a local optimum, a perturbation is applied to be able to escape the local optimum and extend the search space.

The perturbation method consists of γ random insertion moves : each move selects randomly a job and moves it to a random position.

2.1.4 Acceptance criterion

After finding a local optimum, we have to decide if we keep the solution. I chose to implement the simulated annealing as the criterion with $\lambda \cdot \frac{\sum_{j=1}^{n} \sum_{i=1}^{m} p_{ij}}{10min}$ as a constant temperature.

2.1.5 Termination criterion

For both algorithm, the termination criterion is the time. For instances of size $N = 50$, i stop at 70 seconds and for $N = 100$ i stop at 200 seconds. Theses values were chosen as 100 times the runtime of the algorithms implemented in the first part.

2.2 Genetic algorithm

My implementation of a genetic algorithm is inspired form [Zhang et al., 2009] with some slight differences.

1. Generate a initial population
2. Generate a new generation with the crossover operator
3. Perturbate each chromosome with a given probability
4. Apply a local search on each chromosome
5. Merge the old and the new population and select the eligible chromosomes to be kept
6. Goto 2 or stop if the termination criterion is met

2.2.1 Generation of the initial population

The initial population consists of one chromosome generated with $LR(x)$ (see 2.1.1) and the other are generated as random permutation of the job set.

2.2.2 Crossover operator

Two crossover operators were implemented, both described in [Zhang et al., 2009]. SJOX and its improvement with an artificial chromosome generated by WSMGS (weighted simple mining gene structure). The latter was used for the result section.

2.2.3 Mutation operator

The mutation operator is the same as in 2.1.3 but applied only with $P_m \cdot \sqrt{U+1}$ probability. U is defined as the number of generation since the last improvement to the global solution. This is not present in [Zhang et al., 2009] but helps to unstuck the algorithm when it is stuck in a local optimum since a long time.

2.2.4 Local search

The local search method is RZ (as explained in 2.1.2) but also has an original addition, if U is bigger than U_{IRZ}, a IRZ algorithm is used instead because the mutation rate is much higher so the chromosome is highly perturbed.

Using a IRZ all the time was tried but yielded no better results and was significantly slower[1].

2.2.5 Population selection

When selecting the population, many methods were tried : the best, uniform and roulette methods.

These methods were extracted from [Kouki et al., 2016]. They conclude that the uniform method is superior for problems on PFSP of our size but during my experiments, it seemed that the method *best* was superior[2].

The *best* method selects the N best chromosomes form the concatenation of parents and children.

3 Parameters

The parameters cited above were fixed empirically or taken from the literature because of computation power issues (see 3.1). Here are the values fixed in the implementation :

- N (population size): 50 (taken directly from [Zhang et al., 2009])
- U_{IRZ}: 5
- P_m (mutation rate): 0.02
- E (number of elites): 5
- x (from $LR(x)$): $\frac{m}{n}$ (the number of jobs divided by the number of machines) (taken directly from [Pan and Ruiz, 2012])
- γ (number of random moves) : 4
- λ (temperature parameter): 4

3.1 Computation power limitation

A big road-blocker for finding the optimal parameters of choosing a method over another was the limitation by the available computing power. As the exercise suggests running at least each instance 5 times for 200 seconds, the run time is over 16 hour per method or parameter on one CPU.

This meant that some choices were made relying solely on the literature (that was most of the time studying PFSP and not PFSP-WCT) or a few tests on a few instances without any statistical study due to the lack of data.

[1]Due to the lack of computing power (see 3.1), this hypothesis could not be formally proved.
[2]Again, due to a lack of computing power, this could not be proved

The final runs made for the result section were ran on two 32-CPU virtual machines in the cloud to be able to (re-)produce them in time because running experiments for 32 continuous hours on my laptop was not feasible.

4 Results

For the exercise, two experiments were ran. The first was a run of both algorithms five times on each instance. The second was a run of both algorithms, 25 times on the first 5 instances of size 5 with a cutoff time 10 times bigger.

4.1 Score

4.2 Average percentage deviation

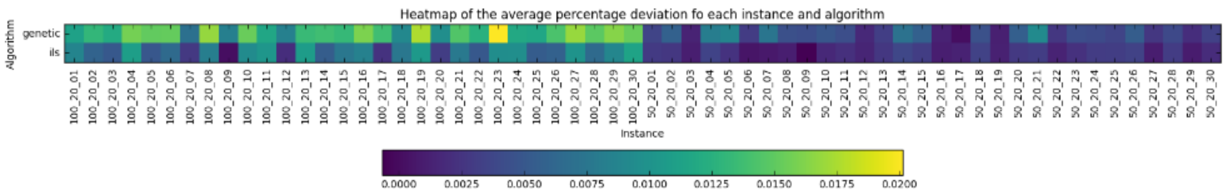
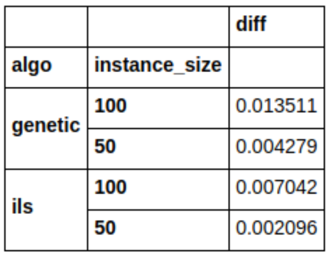


Figure 1: Average percentage deviation from best known solutions for each instance and algorithm

In this heatmap, we can clearly see that my algorithms perform better on the instances of size 50 than size 100. This might be because the time termination criterion is too low for the instances of size 100. It also seems that ILS is better than the genetic on instances for size 100.



		diff
algo	**instance_size**	
genetic	**100**	0.013511
	50	0.004279
ils	**100**	0.007042
	50	0.002096

Figure 2: Average percentage deviation from best known solutions for and algorithm

By looking at the average percentage deviation in the Figure 2, it seems that ILS is better for both sizes.

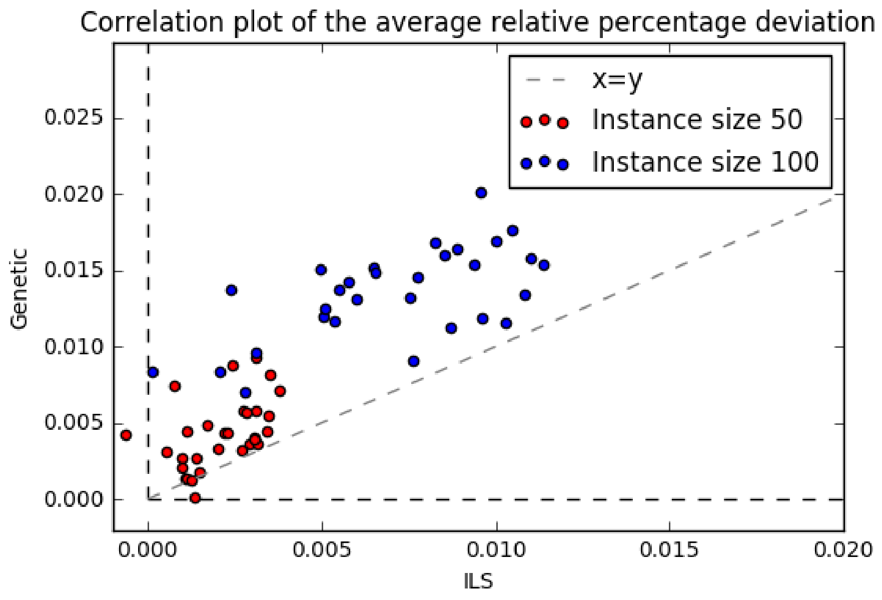


Figure 3: Average percentage deviation from best known solutions for and algorithm

We can then make a correlation plot of the same data to visualize the distribution. We can see that almost every point is above $x = y$ thus, that ILS was better than genetic almost every time.

To be sure of this assumption, we can use statistical tests to verify this hypothesis. The metric used is the Wilcoxon test with the null hypothesis that both samples are drawn from the same distribution.

For the instances of size 50, the p-value is 6.33e−6 and for the size 100, 1.73e−6. Both are bigger than 0.5 thus the hypothesis is rejected. This means that ILS is significantly better than genetic on both instance sizes.

4.3 Run times

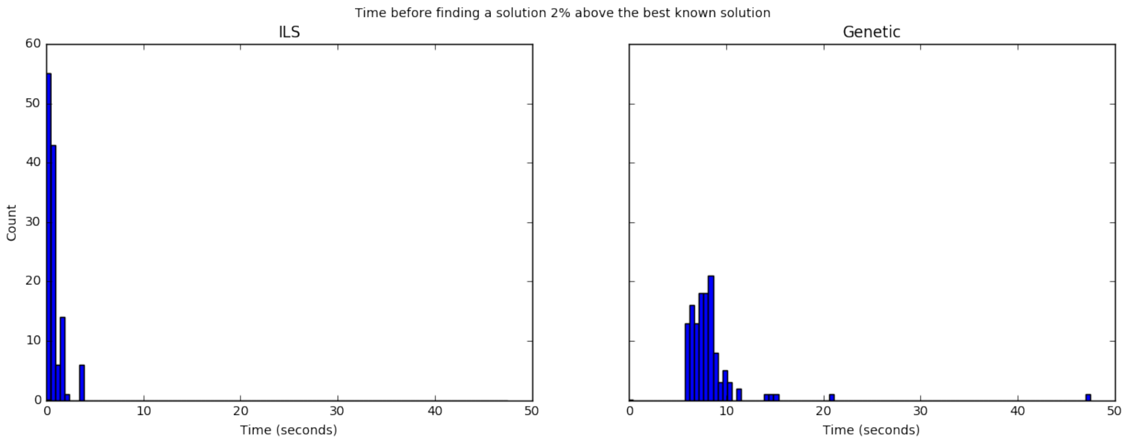
After comparing the scores of both algorithms, we can compare the run times before finding a solution close enough to the best known solution. Theses graphs show the distribution of the time needed per algorithm and per allowed deviation.

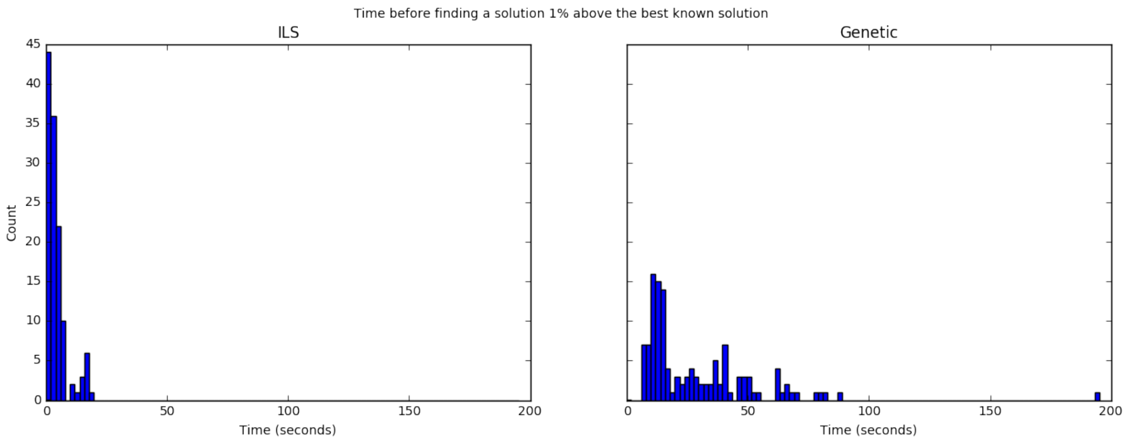
Unfortunately, those graphs are not representative because some instances did not find an acceptable solution before the timeout. For the 0.05% deviation the genetic algorithm was not able to find a acceptable solution for 20 instances out of 125. For the 0.01% deviation, the ILS missed 46 instances and the genetic 56.

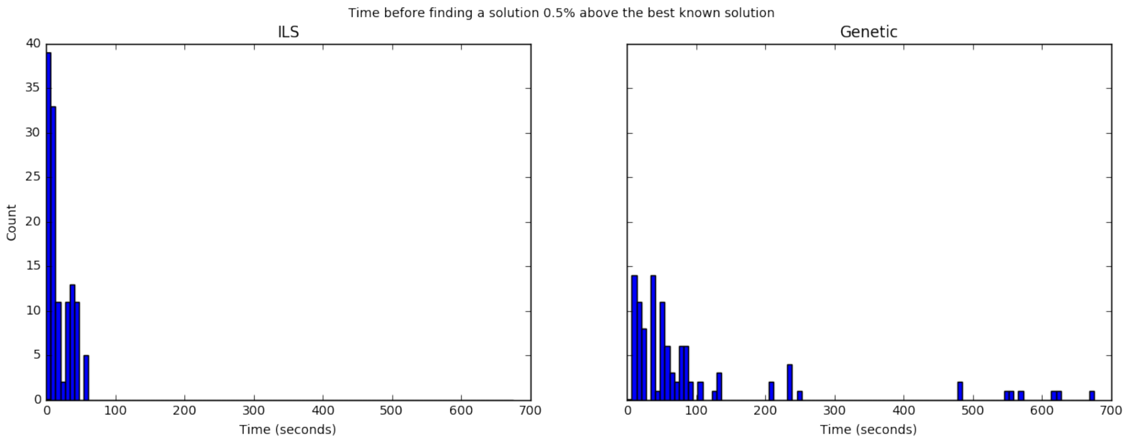
The bar charts just ignore those missing instances and thus do not have the same amount of points.

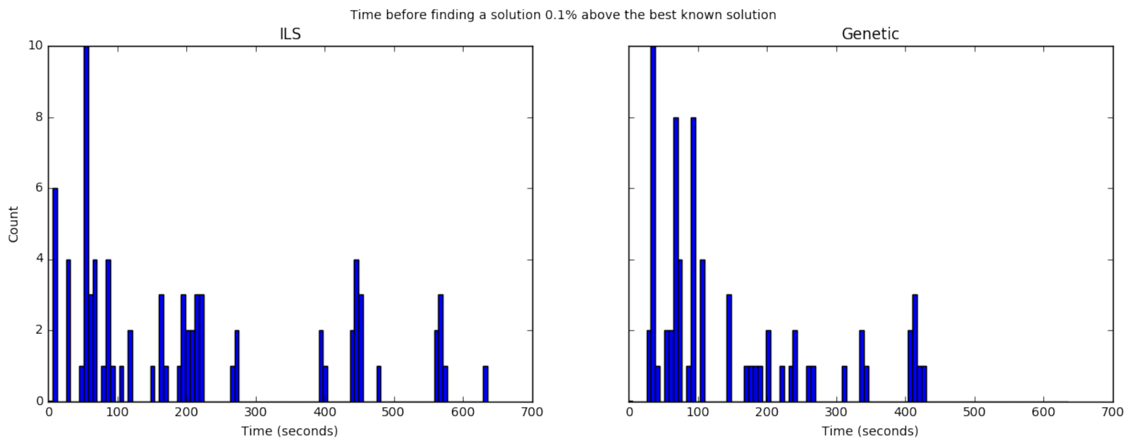
These charts seem to show that ILS is faster to find acceptable solutions for each tolerance. (And the count of missing instances too).

We can confirm this with Wilcoxon tests: they all reject the null hypothesis with p-values of 2.96e−22, 6.87e−22, 7.22e−19 and 6.07e−17.


Time before finding a solution 2% above the best known solution
ILS
Genetic
Count
Time (seconds)
Time (seconds)


Time before finding a solution 1% above the best known solution
ILS
Genetic
Count
Time (seconds)
Time (seconds)


Time before finding a solution 0.5% above the best known solution
ILS
Genetic
Count
Time (seconds)
Time (seconds)



5 Implementation

The implementation is written in C++17 and uses only the standard library. It can be compiled with `make` and ran with `./fssp` *`instance_path timeout algorithm`* where *`instance_path`* is the UNIX path to the instance, *`timeout`* is the timeout for the termination criterion in seconds and *`algorithm`* is `ils` or `genetic`. Example : `./fssp instances/100_20_28 200 ils`.

It outputs the score of the initial solution. Then for each generation or improvement, it outputs the time elapsed in seconds and the intermediate score. The last line is the best achieved score for the run.

The implementation was based on the one from the previous exercise and thus still contains old code, not used anymore.

References

[Gendreau and Potvin, 2010] Gendreau, M. and Potvin, J.-Y. (2010). *Handbook of metaheuristics*, volume 2. Springer.

[Kouki et al., 2016] Kouki, S., Guenaoui, M., and Jemni, M. (2016). A genetic algorithm for the permutation flow shop-problem: A parametric study. In *2016 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6.

[Liu and Reeves, 2001] Liu, J. and Reeves, C. R. (2001). Constructive and composite heuristic solutions to the p//$\sum$ ci scheduling problem. *European Journal of Operational Research*, 132(2):439–452.

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

[Zhang et al., 2009] Zhang, Y., Li, X., and Wang, Q. (2009). Hybrid genetic algorithm for permutation flowshop scheduling problems with total flowtime minimization. *European Journal of Operational Research*, 196(3):869 – 876.