INFO-H413: Heuristic Optimization
First Implementation Exercise

Arkady MOSPAN

April 13, 2021

1 Introduction

We will implement here a few iterative improvement (II) algorithms for the permutation flow-shop scheduling problem (PFSP), analyze the results and compare them to the best known solutions provided.

We have 60 instance files, 30 of them are composed of 50 jobs, and the 30 remaining are composed of 100 jobs. All the code and the analysis will be made using python. The analysis will be using Jupyter notebooks. All the code is available in annex, or on the GitHub repository found at `https://github.com/Mosfet171/INFO-H413`.

1.1 Implementation

The implementation is done via 3 files:

- `psfpfunctions.py` which contains all the core of the implementation of the algorithms by defining a set of functions,
- `pfsp-ii.py` that uses the file above to solve the first exercise (1.1), and
- `pfsp-vnd.py` that also uses the first file to solve the second exercise (1.2).

1.1.1 Variables

The common variables that we handle here are:

- `nJobs`: integer, equal to 50 or 100 depending on the number of jobs in the instance file
- `matrix`: a (`nJobs` $\times$ 20) matrix
- `weights`: a (`nJobs` $\times$ 2) matrix, with the first column containing numbers from 1 to `nJobs` and the second column corresponding to the weights of the jobs defined on the first column of the same row
- `cum_mat`: the cumulative completion times (non weighted) matrix, same size as `matrix`
- `summ`: the weighted sum of completion times, calculated with the element-wise multiplication of the last columns of `cum_mat` and `weights`, and then summing the elements of the obtained vector

1.1.2 Functions

There are 7 functions defined in `psfpfunctions.py`, 1 to calculate the metrics, 1 to generate an initial solution, 3 corresponding to the 3 neighbourhoods, and 2 corresponding to the 2 pivoting rules:

- `calculateMetrics(matrix,weights,cum_mat=[],begin=0)`: returns the makespan, the weighted sum of completion times and the cumulative matrix based on the matrix and the weights vector. If a previous cumulative matrix and a 'begin' is provided, then the calculation is optimized because the calculation will only be done for the rows bigger than 'begin'.

- `initialSolution(instance_path, method)`: returns the initial `matrix`, `weights` and `summ` based on the initial method provided ('srz' or 'ri')
- `exchange(matrix,i,j)`: returns the matrix `matrix` with rows `i` and `j` exchanged.
- `transpose(matrix,i,lr)`: returns `matrix` with rows `i` and `i-1` (`i+1`) exchanged if `lr` $= -1$ (`lr` $= 1$) respectively.
- `insert(matrix,i,abs_loc)`: returns `matrix` with row `i` inserted just before row `abs_loc`
- `calculateFirstNeighbour(matrix, weights, nmethod)`: returns the `matrix`, `weights` and `summ` for the first neighbour of the input matrix having an inferior `summ`, and with the neighbourhood described by `nmethod` ('transpose', 'exchange' or 'insert')
- `calculateBestNeighbour(matrix, weights, nmethod)`: same as above, but with the best neighbour.

Note that in order to keep track of the solution as well as of which weights correspond to which row of `matrix`, the 3 neighbourhood functions must also be applied to the variable `weights`.

2 First exercise

2.1 Introduction

We will implement here 12 II algorithms, one for every combination of the following:

- Initial solution: simplified RZ heuristic (s) or random (r)
- Neighbourhood: transpose (t), exchange (e) or insert (i)
- Pivoting rule: first (f) or best (b)

The python code can be run with the following command:

```
pfsp-ii.py [-h] [-y] (-r | -s) (-t | -e | -i) (-b | -f) filename
```

where '-h' displays a help message. The program returns the estimated weighted sum of completion times for the instance given, or the solution as a vector of the job numbers if the flag `-y` is risen. To give its final result, the code simply calls `calculateFirstNeighbour` (or `-best-`) until the returned sum hasn't improved since the last call.

2.2 Results

2.2.1 Average computation times

Figure 1 and 2 show the average computation times for each variant of the algotithm. The first two tables (figure 1) are for the 50 jobs instances, and the last two (figure 2) for the 100 jobs.

SRZ	**First**	**Best**
Transpose	1.5628	1.7623
Exchange	25.816	14.041
Insert	63.933	34.807

(a) Using simplified RZ heuristic as initial solution

RI	**First**	**Best**
Transpose	3.0884	2.8536
Exchange	28.523	98.714
Insert	75.852	243.35

(b) Using random initial solution

Figure 1: Average computation times (in seconds) of the different variants of the II algorithm for the 50 jobs instances (30 different instances).

SRZ	First	Best
Transpose	11.700	10.806
Exchange	218.70	570.76
Insert	491.24	1181.7

(a) Using simplified RZ heuristic as initial solution

RI	First	Best
Transpose	27.459	24.086
Exchange	479.26	2443.5
Insert	1102.6	5736.0

(b) Using random initial solution

Figure 2: Average computation times (in seconds) of the different variants of the II algorithm for the 100 jobs instances (30 different instances).

2.2.2 Average percentage deviations from best known solutions

Figure 3 and 4 show the average percentage deviations from the best known solutions for each variant of the algotithm. The first two tables (figure 3) are for the 50 jobs instances, and the last two for the 100 jobs.

SRZ	First	Best
Transpose	5.5608	5.5634
Exchange	2.9310	3.7714
Insert	1.7517	2.28702

(a) Using simplified RZ heuristic as initial solution

RI	First	Best
Transpose	33.180	32.339
Exchange	4.4567	2.0115
Insert	2.5998	1.4518

(b) Using random initial solution

Figure 3: Average percentage deviations of the different variants of the II algorithm for the 50 jobs instances (30 different instances).

SRZ	First	Best
Transpose	7.8225	7.8111
Exchange	4.8884	3.2619
Insert	3.6489	2.3600

(a) Using simplified RZ heuristic as initial solution

RI	First	Best
Transpose	37.656	36.958
Exchange	4.6538	1.8966
Insert	4.4785	1.6446

(b) Using random initial solution

Figure 4: Average percentage deviations of the different variants of the II algorithm for the 100 jobs instances (30 different instances).

2.2.3 Statistical tests

Both the paired Student t-test and the Wilcoxon signed-rank test tell us that the results are significantly different for every different variant (significance level 0.05), except for 'SRZ-Transpose-First' and 'SRZ-Transpose-Best' for the 50 jobs instances (p-values 0.7961 for t-test and 0.4074 for Wilcoxon). Note that we only executed the tests for variants that had 1 parameter that differed, as it makes less sense to do it when more than one was different (for example, if we compare SRZ-Transpose-First and SRZ-Exchange-Best and obtain by the statistical tests that those are not significantly different, we cannot conclude anything about the role of Transpose/Exchange or First/Best as they could have a combined impact on the results).

2.3 Analysis

Which initial solution is preferable ?

Comparing the variants that differ only by the initial solution, we see that the simplified RZ heuristics 'wins' 7 times over twelve comparisons. Moreover, if we check the average computation time, we see that it is always lower for the SRZ (sometimes only by 16%, sometimes by 86% !). So we can quite surely state that the simplified RZ heuristics is the best way to initialize the solution.

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

Again, we compare the average percentage deviations and computation time when only the pivoting rule differ. For the better quality solutions, the 'best' rule is by far the winner, with 9.5 wins over 12 (the 0.5 being a 'tie' for SRZ-Transpose for the 50 jobs instances, as explained in the statistical tests section above). For the faster however, we see that the 'first' rule wins 7 times over 12. Two interesting notes though: (1) for the 'transpose' neighbourhood, the 'best' pivoting rule seems to be both the better and the faster, and (2) the 2.5 points 'missing' over 12 are for the 50 jobs instances with the SRZ as initial solution, where the 'first' pivoting rule seems to be the best there.

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

We see clearly that the 'insert' neighbourhood always generates better quality solutions, while the 'transpose' neighbourhood is always the fastest. We should note that in order to achieve its performances, the 'insert' neighbourhood needs a lot more time than its adversaries (in average 75 times the time taken by the 'transpose', and 2.4 times the one taken by the 'exchange').

2.4 Summary

- Best initial solution: simplified RZ heuristic
- Best pivoting rule: 'best'
- Fastest pivoting rule: 'first'
- Best neighbourhood: 'insert', but very slow

3 Second exercise

3.1 Introduction

We will re-use the code written for the first exercise to implement here a variable neighbourhood descent algorithm. The two following orderings of neighbourhoods will be compared:

- Transpose, Exchange and then Insert (`-e` or `--tei`) or
- Transpose, Insert and then Exchange (`-i` or `--tie`)

The python code can be run with the following command:

```
pfsp-vnd.py [-h] [-y] (-i | -e) filename
```

where '-h' displays a help message. The program returns the estimated weighted sum of completion times for the instance given, or the solution as a vector of the job numbers if the flag `-y` is risen. This code uses the functions defined in the `pfspfunctions.py`, in a similar fashion as the `pfsp-ii.py` but of course changing its neighbourhood method.

3.2 Results

3.2.1 Average computation times

Figure 5 shows the average computation times of the two algorithms for the 50 and 100 jobs instances.

Time in s	TIE	TEI
50 jobs	59.175	47.049
100 jobs	1176.8	793.77

Figure 5: Average computation times (in seconds) for two variable neighbourhood descent algorithms, with 'first' pivoting rule and 'SRZ' as initial solution.

3.2.2 Average percentage deviation

Figure 6 shows the average percentage deviation from the best known solutions of the two algorithms for the 50 and 100 jobs instances.

Av. % dev.	TIE	TEI
50 jobs	1.7402	1.8662
100 jobs	2.3925	2.4991

Figure 6: Average percentage deviation from the best known solutions for two variable neighbourhood descent algorithms, with 'first' pivoting rule and 'SRZ' as initial solution.

3.2.3 Percentage improvement

Figure 7 shows the percentage improvement of the two algorithms over the usage of a single neighbourhood, calculated as:

$$PI = \frac{SN - VND}{SN} \cdot 100$$

with SN the value obtained by the single neighbourhood method, and VND the one obtained here with the variable neighbourhood descent. Note that a positive PI shows an improvement, as a negative one shows a worse result than the single neighbourhood method.

W.r.t. exchange (%)	TIE	TEI
50 jobs	40.6	36.3
100 jobs	51.0	48.8
Mean improvement	45.8	42.5

(a) Percentage improvement over the usage of the 'exchange' neighbourhood

W.r.t. insert (%)	TIE	TEI
50 jobs	0.7	-6.5
100 jobs	34.4	31.5
Mean improvement	17.5	12.5

(b) Percentage improvement over the usage of the 'insert' neighbourhood.

Figure 7: Percentage improvement over the usage of a single neighbourhood for two variable neighbourhood descent algorithms, with 'first' pivoting rule and 'SRZ' as initial solution.

3.2.4 Statistical tests

Applying the paired Student t-test on the results, we obtain that for both the 50 and the 100 jobs instances, the results are **not** significantly different (p-value 0.36 and 0.57 respectively).

3.3 Analysis

From the results obtained, we see that the Transpose-Insert-Exchange variable neighbourhood achieves a slightly better improvement than the Transpose-Exchange-Insert, whether it be compared to the single Insert or single Exchange. However, the statistical tests return us that no significant difference is presence, but still the average time taken to compute the algorithms is always higher for the TIE (between 125% and 148%

the one taken by TEI for the 50 and 100 jobs instances respectively).

Compared to the single Exchange neighbourhood, both TIE and TEI achieve better performance, but somehow compensated by a bigger running time. However, compared to the single Insert neighbourhood (which gave the best results in the first implementation), we see:

- a shorter time and a (slightly) better solution for the TIE on 50 jobs,
- a shorter time but a (slightly) worse solution for the TEI on 50 jobs,
- a longer time but a significantly better solution for both TEI and TIE on 100 jobs.

If we were to choose an overall winner, I would personally choose the TEI as it produces solutions that are significantly the same as TIE but with a shorter running time.

3.4 Summary

- TEI and TIE almost same results, but TEI faster
- In general, improvement compared to single neighbourhoods, but at the price of a longer running time
- Improvement more pronounced with larger number of jobs