PFSP heuristics project

Dorian Dumez

April 2, 2017

1 Code

1.1 How to use

In the code directory the command "make" (in the code directory) can be use to compile the whole project. Beside "make clean" (in the code directory too) delete all .o files.

After compiling the project experiment can be done on a single instance with "./main path_to_the_instance optimal_value". Then all possibility are use on this instance, for stochastic setting an average is done over 10 runs. Depending of the settings of the code execution time,value of solutions or proportional difference is output. This can be chose before compiling by the pre-compilator variable, commented or not, at the beginning of the "main.cpp" file. Values for each settings are separated by ":" in the text output.

To have these result on all instances the multiple_run.sh script can be use. This bash script will execute the main on each instance and store all results in the res.txt file.

To compute few statistic R script have been created :

- "compute_average.R" which compute mean value for each algorithm. Values are taken in the file "tmp.txt", so it can be use to compute both average deviation and computation time. Values are output in the "avg.txt" file. We notice that an idicate line can be uncomment if the average is done with relative deviation in $[0; 1]$ and the result is wanted in percentage.

- "compute_standard_deviation.R" which calculate standard deviation, it's have been design in the same way as "compute_average.R" and both take their input in "tmp.txt". Values are output in the "sd.txt" file. The same line can be uncomment to follow the same principle as "compute_average.R".

- "wilcoxon_test.R" compute p-value of the wilconxon pairwise test on each pair of column. Beside it compute the Bonferroni correction on each of them. Output table are stoked into the "wilconxon.txt" file. Input file is "tmp.txt".

1.2 Datastructure of a solution

The solution is represented by a permutation of jobs. It's stored in an static array, the compartment n (we start at the index 0) store the id of the $n + 1$th job to do.

To compute the score we loop on both jobs and machine to compute end date of each jobs on each engine. Then we sum up, with weight, ends date on the last machine. So compute the score of a solution is done in $\Theta(nm)$, where n is the number of jobs and m the number of machine.

Beside, to avoid this too heavy computation time, we design a partial computation function. She use previous computed data to speed up this process. In fact when we compute end date, we store it into the solution's datastructure. And we can notice that end date of a job only depend of the previous one, so if only next one are modified we doesn't need to recompute it, because it doesn't change. In practice this function work like the classical computation of the score but start to compute end date from the index give in parameter, based on previous one. With that the complexity become $O(nm)$.

1.3 Transpose neighbourhood

The transpose neighbourhood is composed of all solution that can be reach by exchange of successive jobs. To scan it we just have to test, for all jobs, if the solution in which the job and the next are switch is better.

In practice we loop on jobs, for each one except the last, we switch him with the next one, then compute the score. If it's better than the actual solution we stop, else we switch them again and go to the next job. If the "dofor" mode is activated we doesn't stop, and just go to the next when we improve. If we are in deepest descent we always undo the switch, and if it's better than the best found so far in this scan we note the current index, after we can do this move in $O(1)$.

To avoid to compute the complete score each time we use the "recomputeWCT" function. But it's work only if jobs before the index parameter haven't been change since the previous computation of the score. Before we notice that we always compute the whole score before the loop to have the score of solution passed in parameter. So we just have to do the loop in the reverse order, start by switch last jobs, to be able to use the partial computation all the time.

1.4 Exchange neighbourhood

The exchange neighbourhood is composed of all solution that can be reach by exchange to job in the queue. To scan it we have, for each job, to test for all other if their switch improve.

In practice we loop on job, and inside we do an other loop to go all over jobs which are after in the queue. Indeed the switch is identical in both way, and we doesn't want to compute two times each neighbour. If it's better than the actual solution we stop, else we switch them again and go to the next job, when all are test we try to move the next job with next ones. If the "dofor" mode is activated we doesn't stop, and just go to the next when we improve. If we are in deepest descent we always undo the switch, and if it's better than the best found so far in this scan we note currents indexes, after we can do this move in $O(1)$.

With the same idea as before we use the "recomputeWCT" function. And to do it we again loops in the reverse order, so we try to exchange jobs with previous ones in the queue. But at the end of all second loop we need to fully compute the score to get the

good end date table. Indeed if we change the last job after working on first one, the end date table might not be correct (in fact it's correct only if the first job has been change and the switch retain).

1.5 Insert neighbourhood

The insert neighbourhood is composed by all solution that can be reach by insert a job between two other. To scan it we need, for all jobs, to try all other places in the queue.

In practice we use two loop, the first one go all over jobs, and the second one move it. Indeed try all possible indexes for a job is equal to switch it with the next one (in a cyclic vision, with a modulo) n times. So for all jobs we do this loop, if an improvement is find we stop all, else if move this job isn't worth we use a loop like the second one, but in reverse order, to bring him back at his original position. In the "dofor" mode we do the first loop for all jobs, and the second one run until an improvement (or the end of possibility). So in this mode we need to notice than a job can be move two times in the same call of the function, but we keep it because the solution might don't be the same until we move it again (beside it's not a deepest descent so an improvement can be found even if the solution haven't been change). Finally for the deepest descent we do all loop until the end, keep tracks of indexes of the best improvement, but in this case the application take a $O(n)$ to be done.

Unfortunately the "recompureWCT" function can't be use with this implementation due to cyclic nature of the walk.

1.6 VND

The implementation of VND is very classic. It have been implement in a generic way, with an array of neighbourhood to be re-used easily. Finally we need to notice that the first neighbourhood is used in descent so it's call in a slightly different way to use the "dofor" improvement if the neighbourhood is set for.

2 Testing

2.1 Test's average of neighbourhood

"tr" point out the transpose neighbourhood, "ex" exchange and "in" for insert. Finally "DD" means we use it int deepest descent. And we need to notice that all these experiment have been done in "dofor" mode.

size	tr	trDD	ex	exDD	in	inDD
50	31.8	32.5	4.1	4.0	6.3	6.5
100	40.9	40.9	4.7	4.7	8.2	9.1
all	36.3	36.7	4.4	4.3	7.3	7.8

Table 1: Relative deviation average in % with random construction

size	tr	trDD	ex	exDD	in	inDD
50	27.6	27.7	4.2	4.1	5.1	5.3
100	35.1	35.2	4.6	5.1	7.3	7.4
all	31.3	31.4	4.4	4.6	6.2	6.3

Table 2: Relative deviation average in % with rz construction

size	tr	trDD	ex	exDD	in	inDD
50	2.99	2.72	0.48	0.45	0.78	0.79
100	3.34	2.66	0.57	0.40	0.94	0.89
all	5.56	4.99	0.61	0.55	1.28	1.57

Table 3: standard deviation of the relative deviation with random construction

size	tr	trDD	ex	exDD	in	inDD
50	8.94	8.82	1.55	1.21	1.51	1.78
100	5.78	6.27	0.95	0.98	1.69	1.26
all	8.40	8.51	1.30	1.19	1.93	1.88

Table 4: standard deviation of the relative deviation with rz construction

size	tr	trDD	ex	exDD	in	inDD
50	0.22	0.74	5.19	25.01	14.82	78.41
100	0.93	4.80	45.58	358.2	156.1	1236
all	0.57	2.73	25.04	188.8	84.27	647.70

Table 5: execution time average with random construction

size	tr	trDD	ex	exDD	in	inDD
50	0.61	0.76	5.43	24.04	14.57	58.42
100	3.91	5.12	47.37	335.6	141.3	952.6
all	2.23	2.90	26.04	177.20	76.88	497.9

Table 6: execution time average with rz construction

size	tr	trDD	ex	exDD	in	inDD
50	0.0008	0.002	0.004	0.012	0.014	0.031
100	0.002	0.005	0.034	0.166	0.129	0.842
all	0.004	0.020	0.205	1.684	0.718	5.869

Table 7: execution time standard deviation with random construction

size	tr	trDD	ex	exDD	in	inDD
50	0.0003	0.001	0.010	0.057	0.037	0.161
100	0.001	0.005	0.097	0.442	0.302	1.855
all	0.016	0.022	0.222	1.601	0.673	4.690

Table 8: execution time standard deviation with rz construction

2.2 Statistical test of neighbourhood

We perform the wilcoxon test on score, in a pairwise way, and after applied the Bonferroni correction to the obtained p-value. These result are compiled in the table 9. Beside we do the same calculation for the execution time, these results are in the table 10. A population isn't comparable to itself so NaN is used in these cases.

"tr", "ex" and "in" are use in the same way as previously for transpose, exchange and insert. "rnd" is used when the start solution isn't construct, and "rz" when the rz heuristic have been used. "DD" is always here when the descent have been done in deepest descent.

	rndtr	rndtrDD	rndex	rndexDD	rndin	rndinDD	rztr	rztrDD	rzex	rzexDD	rzin	rzinDD
rndtr	NaN	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.24e-4	3.05e-3	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rndtrDD	1.00	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	2.64e-4	7.39e-4	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rndex	9.32e-9	9.32e-9	NaN	1.00	9.32e-9	9.32e-9	9.81e-9	9.81e-9	1.00	1.00	5.78e-7	2.75e-6
rndexDD	9.32e-9	9.32e-9	1.00	NaN	9.32e-9	9.32e-9	9.81e-9	9.81e-9	1.00	1.00	5.02e-7	2.10e-6
rndin	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	5.47e-4	9.81e-9	9.81e-9	2.00e-8	9.32e-9	1.01e-2	3.45e-2
rndinDD	9.32e-9	9.32e-9	9.32e-9	9.32e-9	5.47e-4	NaN	9.81e-9	9.81e-9	1.81e-8	9.32e-9	3.76e-6	1.65e-5
rztr	9.24e-4	2.64e-4	9.81e-9	9.81e-9	9.81e-9	9.81e-9	NaN	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rztrDD	3.05e-3	7.39e-4	9.81e-9	9.81e-9	9.81e-9	9.81e-9	1.00	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rzex	9.32e-9	9.32e-9	1.00	1.00	2.00e-8	1.81e-8	9.32e-9	9.32e-9	NaN	1.00	7.60e-6	2.40e-6
rzexDD	9.32e-9	9.32e-9	1.00	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	NaN	4.69e-6	1.65e-5
rzin	9.32e-9	9.32e-9	5.78e-7	5.02e-7	1.01e-2	3.76e-6	9.32e-9	9.32e-9	7.60e-6	4.69e-6	NaN	1.00
rzinDD	9.32e-9	9.32e-9	2.75e-6	2.10e-6	3.45e-2	1.65e-5	9.32e-9	9.32e-9	2.40e-6	1.65e-5	1.00	NaN

Table 9: Befferoni correction of the wilcoxon pairwise test, solution quality

	rndtr	rndtrDD	rndex	rndexDD	rndin	rndinDD	rztr	rztrDD	rzex	rzexDD	rzin	rzinDD
rndtr	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rndtrDD	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.22e-5	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rndex	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9
rndexDD	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	9.32e-9	1.03e-8
rndin	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.72e-8	1.00	9.32e-9
rndinDD	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	2.11e-8
rztr	9.32e-9	1.22e-5	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rztrDD	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rzex	9.32e-9	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	9.32e-9	9.32e-9	9.32e-9
rzexDD	9.32e-9	9.32e-9	9.32e-9	1.00	1.72e-8	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	1.14e-8	9.32e-9
rzin	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.14e-8	NaN	9.32e-9
rzinDD	9.32e-9	9.32e-9	9.32e-9	1.03e-8	9.32e-9	2.11e-8	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN

Table 10: Befferoni correction of the wilcoxon pairwise test, execution time

We compute the average of the relative deviation, for all instances and all neighbourhood relation (in deepest descent and first improvement), when we start with a random solution and the rz heuristic. After we compute a pairwise wilcoxon test, compared value are always output by the same algorithm (same neighbourhood, same browsing, different initialisation) on the same instance so when a comparison is done only the initialization differ. With a random start the relative deviation average is 16.1% and with the heuristic it's 14.0%. Finally the Befferoni corrected p-value of the wilcoxon pairwise test is

$5.66e^{-15}$. So we can conclude that an initialisation with the rz heuristic is better as a starting point for a local search.

Then the same study have been done with the pivoting rule. With the first improvement the relative deviation average is 15.0% and with the deepest descent it's 15.2%. Beside the Befferoni corrected p-value of the wilconxon pairwise test (same idea as before, the only things that differ between two compared value is the pivoting rule) is 0.0004. Surprisingly the first improvement with a for loop give better result than a deepest descent.

Finally we do this a third time with the neighbourhood relationship. With the transpose relation the relative deviation average (on every pivoting rules and starting point) is 33.9%, with exchange it's 4.4% and with insert it's 6.9%. As before a wilconxon pairwise test have been computed, Befferoni corrected p-value are in the table 11. So with that we can said that exchange is better than insert which is better than transpose, and the relation is transitive.

	tr	ex	in
tr	NaN	1.10e-39	1.10e-39
ex	1.10e-39	NaN	2.33e-35
in	1.10e-39	2.33e-35	NaN

Table 11: Befferoni corrected wilconxon p-value of neighbourhood, solution quality

With all of that the best setting might be the exchange neighbourhood browse in first improvement with a for loop starting by the rz heuristic. According to the table 1, 2 and 9 it's true. Except that we can't said that population of result output by other usage of the exchange neighbourhood (with the random start or in deepest descent) are different than the one output by this version. So they might output solution as good as this version.

Now go on the executions time point of view.

The average computation time average overall in first improvement is 35.83 seconds against 252.8 seconds for the deepest descent. Beside we can notice that for every variant the first improvement version is always faster than the deepest version one, at least until the instance size level (so on average on same instance size first improvement is faster). So no need to a wilcoxon test to understand that the first improvement is clearly faster.

The average computation (over all other settings) time with the transpose neighbourhood is 2.11 seconds, 104.28 seconds with exchange and 326.70 with insert. And the p-value of the pairwise wilconxon test, table 12, show that it's absolutely not an average artefact. So the descent in the transpose neighbourhood take clearly less time to reach a local optima, then exchange is slower but faster than insert.

	tr	ex	in
tr	NaN	1.1e-39	1.1e-39
ex	1.1e-39	NaN	1.1e-39
in.	1.1e-39	1.1e-39	NaN

Table 12: Befferoni corrected wilconxon p-value of neighbourhood, execution time

To conclude about the usage of a single neighbourhood we can prefer the exchange neighbourhood in first improvement, but the initialisation doesn't matter. Indeed in the first part we said that all usage of this neighbourhood was equal regard of the solution quality (Cf table 1, 2, 9), and in the second on we said that first improvement is faster that deepest descent but initialisation doesn't matter (Cf table 5, 6, 10).

2.3 Test's average of VND

VND1 indicate the VND algorithm which use transpose, then exchange and finish with insert. VND2 is for the one which use transpose, then insert, then exchange. DD mean we use neighbourhoods in deepest descent. Same as previously the transpose neighbourhood is use in "dofor" mode.

size	VND1	VND1DD	VND2	VND2DD
50	3.5	3.4	4.2	4.3
100	4.4	4.6	5.5	5.5
all	4.0	4.0	4.9	4.9

Table 13: relative deviation average in % with random construction

size	VND1	VND1DD	VND2	VND2DD
50	3.3	3.5	3.4	3.9
100	4.4	4.4	5.3	4.9
all	3.9	3.9	4.3	4.4

Table 14: relative deviation average in % with rz construction

size	VND1	VND1DD	VND2	VND2DD
50	0.51	0.52	0.54	0.54
100	0.56	0.63	0.60	0.59
all	0.72	0.81	0.85	0.83

Table 15: standard deviation of the relative deviation with random construction

size	VND1	VND1DD	VND2	VND2DD
50	1.26	0.99	1.06	1.21
100	1.10	0.84	1.27	1.01
all	1.28	1.04	1.52	1.23

Table 16: standard deviation of the relative deviation with rz construction

size	VND1	VND1DD	VND2	VND2DD
50	31.34	31.79	68.83	71.70
100	454.0	412.5	1511	1229
all	239.1	218.9	778.0	640.8

Table 17: execution time average with random construction

size	VND1	VND1DD	VND2	VND2DD
50	29.15	30.99	63.65	57.08
100	409.8	428.8	1407	1034
all	216.2	226.5	724.2	537.6

Table 18: execution time average with rz construction

size	VND1	VND1DD	VND2	VND2DD
50	0.033	0.020	0.047	0.036
100	0.515	0.279	1.519	0.757
all	2.161	1.929	7.351	5.861

Table 19: execution time standard deviation with random construction

size	VND1	VND1DD	VND2	VND2DD
50	0.097	0.086	0.178	0.139
100	0.780	0.906	2.820	1.974
all	1.995	2.103	7.055	5.118

Table 20: execution time standard deviation with rz construction

	rndex	rndexDD	rndin	rndinDD	rzex	rzexDD	rzin	rzinDD
rndVND1	0.42	0.38	3.17	3.66	0.42	0.61	2.15	2.27
rndVND1DD	0.38	0.34	3.13	3.63	0.39	0.58	2.11	2.24
rzVND1	0.53	0.48	3.28	3.77	0.53	0.72	2.25	2.38
rzVND1DD	0.44	0.39	3.19	3.68	0.44	0.63	2.17	2.29
rndVND2	-0.43	-0.47	2.28	2.77	-0.43	-0.24	1.27	1.40
rndVND2DD	-0.49	-0.53	2.23	2.72	-0.48	-0.29	1.22	1.34
rzVND2	0.09	0.04	2.82	3.32	0.09	0.28	1.80	1.93
rzVND2DD	0.007	-0.03	2.74	3.23	0.01	0.20	1.72	1.85

Table 21: relative improvement average of VND to single neighbourhood, in %

2.4 Statistical test of VND

We perform the wilcoxon test on score, in a pairwise way, and after applied the Bonferroni correction to the obtained p-value. A population isn't comparable to itself so NaN is used in these cases. First, in the table 22, VND version are compared to each other. Secondly, in the table 23, VND version are compared to the usage of a single neighbourhood. Finally, after the comparison in term of solution quality, we compared VND version regard of the execution time, result are in the table 24, and 25 for the comparison with single neighbourhood usage.

	rndVND1	rndVND1DD	rzVND1	rzVND1DD	rndVND2	rndVND2DD	rzVND2	rzVND2DD
rndVND1	NaN	1.00	1.00	1.00	2.67e-9	2.95e-9	0.98	0.01
rndVND1DD	1.00	NaN	1.00	1.00	2.41e-9	1.37e-9	0.78	0.26
rzVND1	1.00	1.00	NaN	1.00	1.17e-4	3.57e-6	0.62	0.42
rzVND1DD	1.00	1.00	1.00	NaN	1.05e-5	1.73e-6	1.00	0.54
rndVND2	2.67e-9	2.41e-9	1.17e-4	1.05e-5	NaN	1.00	0.20	0.15
rndVND2DD	2.95e-9	1.37e-9	3.57e-6	1.73e-6	1.00	NaN	0.08	4.78e-3
rzVND2	0.98	0.78	0.62	1.00	0.20	0.08	NaN	1.00
rzVND2DD	0.01	0.26	0.42	0.54	0.15	4.78e-3	1.00	NaN

Table 22: Befferoni correction of the wilcoxon pairwise test, solution quality

	rndtr	rndtrDD	rndex	rndexDD	rndin	rndinDD	rztr	rztrDD	rzex	rzexDD	rzin	rzinDD
rndVND1	9.32e-9	9.32e-9	0.03	1.85e-3	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	0.04	2.45e-8	2.14e-7
rndVND1DD	9.32e-9	9.32e-9	0.08	0.17	9.32e-9	9.32e-9	9.81e-9	9.81e-9	1.00	0.24	6.03e-8	2.04e-7
rzVND1	9.32e-9	9.32e-9	1.00	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	0.39	1.76e-7	1.94e-7
rzVND1DD	9.32e-9	9.32e-9	1.00	1.00	9.81e-9	9.32e-9	9.32e-9	9.32e-9	1.00	0.16	5.51e-7	1.38e-7
rndVND2	9.32e-9	9.32e-9	5.27e-4	7.71e-5	9.32e-9	9.32e-9	9.81e-9	9.81e-9	0.55	1.00	1.06e-4	9.43e-5
rndVND2DD	9.32e-9	9.32e-9	1.15e-4	6.06e-7	9.32e-9	9.32e-9	9.81e-9	9.81e-9	0.20	1.00	7.71e-5	9.06e-5
rzVND2	9.32e-9	9.32e-9	1.00	1.00	9.81e-9	9.32e-9	9.32e-9	9.32e-9	1.00	1.00	3.61e-7	1.11e-6
rzVND2DD	9.32e-9	9.32e-9	1.00	1.00	1.03e-8	9.81e-9	9.32e-9	9.32e-9	1.00	1.00	7.71e-8	9.64e-7

Table 23: Befferoni correction of the wilcoxon pairwise test, solution quality

	rndVND1	rndVND1DD	rzVND1	rzVND1DD	rndVND2	rndVND2DD	rzVND2	rzVND2DD
rndVND1	NaN	1.50e-1	1.00	1.00	9.32e-9	9.32e-9	1.14e-8	1.27e-8
rndVND1DD	1.50e-1	NaN	1.00	1.00	9.32e-9	9.32e-9	1.14e-8	1.40e-8
rzVND1	1.00	1.00	NaN	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rzVND1DD	1.00	1.00	1.00	NaN	9.32e-9	9.32e-9	9.32e-9	9.32e-9
rndVND2	9.32e-9	9.32e-9	9.32e-9	9.32e-9	NaN	4.16e-2	1.00	8.10e-8
rndVND2DD	9.32e-9	9.32e-9	9.32e-9	9.32e-9	4.16e-2	NaN	1.00	1.02e-4
rzVND2	1.14e-8	1.14e-8	9.32e-9	9.32e-9	1.00	1.00	NaN	5.27e-4
rzVND2DD	1.27e-8	1.40e-8	9.32e-9	9.32e-9	8.10e-8	1.02e-4	5.27e-4	NaN

Table 24: Befferoni correction of the wilcoxon pairwise test, execution time

	rndtr	rndtrDD	rndex	rndexDD	rndin	rndinDD	rztr	rztrDD	rzex	rzexDD	rzin	rzinDD
rndVND1	9.32e-9	9.32e-9	9.32e-9	9.81e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.48e-8	9.32e-9	1.20e-8
rndVND1DD	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	7.30e-7	9.32e-9	1.48e-8
rzVND1	9.32e-9	9.32e-9	9.32e-9	1.03e-1	1.48e-8	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.66e-3	9.32e-9	9.32e-9
rzVND1DD	9.32e-9	9.32e-9	9.32e-9	5.79e-5	1.03e-8	9.32e-9	9.32e-9	9.32e-9	9.32e-9	5.56e-5	9.32e-9	9.81e-9
rndVND2	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	5.28e-1	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	5.51e-7
rndVND2DD	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	5.51e-7
rzVND2	9.32e-9	9.32e-9	9.32e-9	9.81e-9	9.32e-9	1.00	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	4.72e-5
rzVND2DD	9.32e-9	9.32e-9	9.32e-9	9.81e-9	9.32e-9	1.08e-5	9.32e-9	9.32e-9	9.32e-9	9.32e-9	9.32e-9	1.00

Table 25: Befferoni correction of the wilcoxon pairwise test, execution time

First speak about the solution quality reach by the two order of neighbourhood used by VND. The average relative deviation overall with VND1 is 3.98% and it's 4.66% with VND2, beside the corrected p-value of the pairwise wilcoxon test is $1.29e^{-18}$. So we can conclude that VND1 perform better quality solution. But after VND1 version aren't

comparable to each other relatively to the solution quality, according to the table 22 we can't say that results populations are different.

Secondly we look at the computation time of these two neighbourhood order. The average overall of VND1 is 225.2 seconds against 670.2 seconds for VND2 with a wilcoxon p-value of $3.65e^{-40}$. So VND1 outperformed again VND2. As before, regarding to table 17, 18, and 24, we can just say that with a random initialisation VND1 in first improvement is faster than VND1 in deepest descent.

To conclude about VND we can just say that VND2 is completely dominated by VND1, but VND1 versions are equals. In the first part conclusion we preferred exchange in first improvement, so which one is the best. According to table 21 and 23 we can't said that VND1 in first improvement and exchange in first improvement give better solution. And according to table 5, 2, 17, 18 and 25 we can say that the single neighbourhood is faster than VND1 when both are used in first improvement (and now matter which construction heuristic is chosen).

3 Conclusion

Do a descent in first improvement (with a for loop) into an exchange neighbourhood, and no matter how the initial solution solution is construct, is the best algorithm we study here for the PFSP.

As a further development we can study a VND algorithm which use only exchange and insert. Indeed in the first part we saw that transpose doesn't find any good solution, so it seems it doesn't help other neighbourhood to converge faster in a VND. But quite the opposite it will slow down them because they won't be able to converge with a for loop and useless search phase into transpose neighbourhood will be performed between every search into promising neighbourhood.

Finally, even if we haven't speak about that before, we can notice that all algorithm are very stable over all point of view across table 3, 4, 7, 8, 15, 16, 19, 20.