PFSP heuristics project

Dorian Dumez

March 26, 2017

1 Code

1.1 How to use

In the code directory the command "make" can be use to compile the whole project. Beside "make clean" 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 run. 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, comment 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.

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 do 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 we again change last job after work on first one, so 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).

TODO le calcul qui montre que le nombre d'opération est moindre.

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 implantation of VND is very classic. It have been implement in a generic way, with an array of neighbourhood to be re-used easily. Finaly 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.

taille	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 in % with random construction

taille	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 in % with rz construction

taille	tr	trDD	ex	exDD	in	inDD
50	2.9	2.7	0.4	0.4	0.7	0.7
100	3.3	2.6	0.5	0.4	0.9	0.8
all	5.5	4.9	0.6	0.5	1.2	1.5

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

taille	tr	trDD	ex	exDD	in	inDD
50	8.9	8.8	1.5	1.2	1.5	1.7
100	5.7	6.2	0.9	0.9	1.6	1.2
all	8.4	8.5	1.3	1.1	1.9	1.8

Table 4: standard deviation of the relative 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 following table. A population isn't comparable to itself so NaN is used in these cases.

	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 5: Befferoni correction of the wilcoxon pairwise test

2.3 Test's average of VND

VND1 design 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.

taille	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 6: relative deviation in % with random construction

taille	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 7: relative deviation in % with rz construction

taille	VND1	VND1DD	VND2	VND2DD
50	0.5	0.5	0.5	0.5
100	0.5	0.6	0.6	0.5
all	0.7	0.8	0.8	0.8

Table 8: standard deviation of the relative deviation with random construction

taille	VND1	VND1DD	VND2	VND2DD
50	1.2	0.9	1.0	1.2
100	1.1	0.8	1.2	1.0
all	1.2	1.0	1.5	1.2

Table 9: standard deviation of the relative deviation with rz construction

2.4 Statistical test of VND

3 Conclusion

3.1 Best construction heuristic

3.2 Best neighbourhood

3.3 Best setting for VND

3.4 Best solution overall