Département de Génie Électrique & Informatique

Rapport de projet Metha Heuristique

4ème année Informatique et Réseaux

Problème de JobShop

Réalisé par

Emile Sebastianutti sebastia@etud.insa-toulouse.fr

22nd May 2020

Lien Git

https://github.com/emileseb/template-jobshop

Table of Contents

1 Appropriation du projet

1.1 Prise en main du code

Le projet source (nous ne parlerons pas des fichiers de traitement du code gradle) est divisé en trois parties :

- Une partie représentation du problème : il s'agit du package encodings. Celui-ci contient les objets nécéssaires à la représentation générale d'un JobShop sur lequel nous allons ensuite pouvoir appliquer des algorithmes.
- Une partie Solvers : située dans le package du même nom, elle contient les différentes approches de résolution d'un problème JobShop en général. Certains de ces solvers peuvent s'appuyer sur d'autres comme nous le verrons par la suite avec Descent et Greedy par exemple.
- Une partie instanciation du problème. Cette fois-ci nous appliquons aux ressources générales précédentes des cas concrets de JobShop. Cette partie est contenue dans le dossier test mais aussi et principalement dans le Main et ce qui l'entoure. C'est dans cette partie que vont être appliqués les solvers précédents sur des instances de problème.

Lors de la prise en main du code, j'ai implémenté la méthode toString de la classe Schedule afin de me familiariser avec cette classe déterminante pour la suite et comprendre mieux la représentation d'un problème. Voici le résultat :

```
Job number 0 :
Task number 0 starts at time 0
Task number 1 starts at time 3
Task number 2 starts at time 6
Job number 1 :
Task number 0 starts at time 0
Task number 1 starts at time 3
Task number 2 starts at time 8
```

FIG. 1 – *Représentation d'une solution pour l'instance aaa1*

Nous avons donc ici les tâches de chaque job et leur date de début. C'est de là que nous allons pouvoir partir pour implémenter les algorithmes.

1.2 Représentation de solutions et Espace de recherche

Les représentations des instances servent pour l'exécution des algorithmes. Il vont itérer au sein de l'espace de recherche pour trouver une solution au problème. Nous serions tenté en premier lieu de calculer une solution optimale en parcourant l'ensemble de l'espace de recherche. Regardons donc la taille de cet espace de recherche.

Si l'on se base sur une représentation par numéros de Jobs, on se trouve face à un espace de recherche de taille

$$|\Omega| = \frac{(nJobs * nMachines)!}{nMachines!^{nJobs}}$$

Dans le cas par exemple de l'instance ft06 (6 Jobs, 6 Taches et 6 machines) la taille de l'espace de recherche est donc $|\Omega| = \frac{(6*6)!}{6!^6} = 2.67.10^{24}$ c'est énorme pour un problème si banal (un collège de 6 classes, 6 salles de cours et 6 cours). Pour nous en convaincre, admettons qu'il faille 1 nanoseconde pour explorer chaque cas, cela reviendrait à environs **84 millions d'années** de temps de calcul.

Prenons alors d'autres représentations : par ordre de passage sur les ressources :

$$|\Omega| = (nJobs!)^{nMachines}$$

Pour l'instance ft06 nous aurions alors $|\Omega| = (6!)^6 = 1.39.10^{17}$ ce qui en terme de temps de parcours vaudrait environ **4 ans, 4 mois, 27 jour, 3 heures, 6 minutes, 28 secondes** (sans coupure de courant). C'est déjà beaucoup plus faible mais toujours incalculable. Enfin pour la représentation par date de début de chaque tâches, la taille de l'espace de recherche serait de

$$|\Omega| = (D_{max})^{nJob*nMachines}$$

Appliqué à l'instance ft06 nous obtenons $|\Omega| = (197)^{6*6} = 3.99.10^{82}$ qui demanderai de l'ordre de 10^{66} **années**.

Nous sommes face à des espaces de recherches bien trop grand pour les parcourir entièrement, nous allons devoir compromettre l'optimalité du résultat au profit du temps de calcul. Pour cela allons nous baser sur des méthodes Heuristiques approchées.

2 Heuristique Gloutonne

L'heuristique gloutonne a pour but de constituer un ordre de passage des tâches sur les machines. Elle ne sera donc pas une méthode de recherche parmis l'ensemble des solutions possible, ce qui peut être très long comme nous l'avons vu précédemment, mais elle consiste à choisir une tâche parmis la liste des taches exécutable selon une règle gloutonne (voir ci-dessous) et l'ajouter à la bonne machine comme prochaine tâche à exécuter dans le RessourceOrder de retour. Cela aura pour effet d'être une méthode très rapide, instantanée à l'échelle humaine, et donnera un résultat plus ou moins satisfaisant. Elle aura vocation à être la base d'heuristiques de recherche parmis l'espace de solutions qui, elles, vont tenter de trouver un optimum. L'optimum n'est pas le but premier de l'heuristiques gloutonne : c'est la rapidité.
Elle fonctionne suivant ce schéma :

```
Data : instance, règle
Result : Result
order ← newResourceOrder();
listeTachesExecutable ← null;
for job ∈ Instance.Jobs do
    listeTachesExecutable.add(job.firstTask);
end
tant que ListeTachesExecutable.nonVide() faire
    taskToAdd ← listeTachesExecutable.ChoixSelon(regle);
    order.add(taskToAdd);
    listeTachesExecutable.remove(taskToAdd);
    if taskToAdd.number <instance.nombreDeTacheParJob then
        listeTachesExecutable.add(taskToAdd.nextInJob());
    end
fin
return result(instance,order.toSchedule);
```

L'efficacité d'une telle méthode réside donc dans la règle que nous allons choisir pour remplir l'ordre.

2.1 Les règles de base

Les règles définissent quelle tache va être la prochaine à être ajoutée. Les règles de basent choisissent parmis toute la liste de tâche exécutable.

2.1.1 Shortest or Longest Processing Time

Cette règle est basée sur le temps d'exécution de la tâche.

```
Data : instance, listeTachesExecutables
Result : Prochaine tache à exécuter
taskToAdd ← listeTachesExecutables.get(0);
for tacheI ∈ listeTachesExecutables do
    if tacheI.duration() [<ou >selon SPT ou LPT] taskToAdd.duration() then
        taskToAdd ← tacheI;
    end
end
return taskToAdd;
```

2.1.2 Shortest or Longest Remaining Processing Time

Cette fois-ci la règle est basé non pas sur le temps d'exécution de la tâche mais sur le temps restant d'exécution du Job.

```
Data : instance, listeTachesExecutables
Result : Prochaine tache à exécuter
taskToAdd ← listeTachesExecutables.get(0);
for tacheI ∈ listeTachesExecutables do
    remainingTime ← 0;
    for tacheRestante > tacheI ∈ tacheI.job do
        remainingTime ← +tacheRestante.getDuration();
    end
    if remainingTime [<ou >selon SRPT ou LRPT] bestRemainingDuration then
        taskToAdd ← tacheI;
        bestRemainingDuration ← remainingTime;
    end
end
return taskToAdd;
```

2.2 Les règles améliorées - EST

L'amélioration EST va, quand à elle, réduire la liste sur laquelle nous allons exécuter les règles précédentes. En effet, son travail va être de fournir aux méthodes précédentes non plus une liste de toutes les tâches exécutables mais une liste des taches pouvant démarrer les plus rapidement. Le but de cette amélioration va être de réduire les temps morts pendant lesquels une machine est inutilisée ou un job n'est pas en traitement.

Pour cela nous allons modifier l'algorithme glouton général pour qu'il mette à jour deux tableaux qui pour chaque tâche donne l'un le delais le plus court avant l'exécution sur la machine nécéssaire et l'autre le delais le plus court avant l'exécution d'une nouvelle tâche du Job.

```
nextStartingTimeForJob[addedTask.job] ←
    addedTask.startingTime + addedTask.duration;
/*idem pour les machines.*/
```

L'algorithme de la méthode EST se base donc sur ces tableaux et sur la règle à utiliser ensuite :

Data : priorityRule, instance, listeTachesExecutables, nextStartingTimeForMachine[int], nextStartingTimeForJob[int]
Result : Prochaine tache à exécuter

```
ESTTaskList ← null;
for tacheI ∈ listeTachesExecutables do
    t_startingTime ←
      Max(nextStartigTimeForJob[tacheI.job],nextStartigTimeForMachine[tacheI.machine]);

    if t_startingTime <earliestDate then
        earliestDate ← t_startingTime;
        ESTTaskList.clear();
        ESTTaskList.add(tacheI);
    else
        if t_startingTime == earliestDate then
            ESTTaskList.add(tacheI);
        end
    end
end
return priorityRule(ESTTaskList);
```

2.3 Comparatif des résultats

J'ai testé ces différentes méthodes en les appliquant sur les instances aaa1, ft06, ft10, ft20, la01 et la40, en voici les résultats :

```
emile@A1L1:~/Documents/INSA/sem8/metha/template-jobshop$ java -jar build/libs/JSP.jar --solver greedySPT greedyLPT greedySRPT greedyLRPT -
```

			greedySPT			greedyLPT			greedySRPT			greedyLRPT		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	7	16	45,5	1	14	27,3	0	14	27,3	0	11	0,0
ft06	6x6	55	1	108	96,4	1	133	141,8	2	157	185,5	1	70	27,3
ft10	10x10	930	3	2569	176,2	2	2860	207,5	2	2938	215,9	2	1532	64,7
ft20	20x5	1165	1	2765	137,3	1	2603	123,4	1	3003	157,8	1	2248	93,0
la01	10x5	666	1	1464	119,8	1	1996	199,7	1	2178	227,0	1	1339	101,1
la40	15x15	1222	1	6098	399,0	1	7428	507,9	1	9283	659,7	1	2137	74,9
AVG	-	-	2,3	-	162,4	1,2	-	201,3	1,2	-	245,5	1,0	-	60,1

```
emile@A1L1:~/Documents/INSA/sem8/metha/template-jobshop$ java -jar build/libs/JSP.jar --solver greedyESTSPT greedyESTLPT greedyESTSRPT gre
```

			greedyESTSPT			greedyESTLPT			greedyESTSRPT			greedyESTLRPT		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	7	11	0,0	0	11	0,0	0	11	0,0	0	11	0,0
ft06	6x6	55	1	88	60,0	1	67	21,8	1	70	27,3	1	65	18,2
ft10	10x10	930	2	1074	15,5	2	1295	39,2	2	1257	35,2	2	1145	23,1
ft20	20x5	1165	2	1267	8,8	1	1621	39,1	1	1380	18,5	1	1538	32,0
la01	10x5	666	0	751	12,8	0	792	18,9	0	1003	50,6	1	701	5,3
la40	15x15	1222	1	1476	20,8	1	1856	51,9	1	1564	28,0	1	1454	19,0
AVG	-	-	2,2	-	19,6	0,8	-	28,5	0,8	-	26,6	1,0	-	16,3

Fig. 2 – *Resultats des différentes règles Gloutonnes*

Ce que l'on remarque tout d'abord c'est le temps d'exécution : faces aux années des méthodes exhaustives, nous sommes ici dans de l'instantané pour l'Homme. Nous avons donc résolu la problématique soulevé lors du point précédent et même s'il existe de faibles écarts lors des exécutions, les espaces parcourus ne sont plus des puissances de 10 mais des tableaux de taches exécutable donc la différence est ici négligeable.
Maintenant si le temps d'exécution est acceptable, qu'en est-il du résultat ? Ce qui nous intéresse ici n'est pas tant le makespan qui est relatif à chaque problème que l'écart entre notre solution et la solution optimale.
On observe alors sur ces résultats plusieurs choses. Tout d'abord, nous ne trouvons quasiment jamais une solution optimale, et jusqu'ici, rien d'étonnant, c'est le principe d'un algorithme glouton. Regardons alors les règles de bases. On note qu'elles ont leur écart relativement proche les unes des autres, et qu'il est plutôt éloigné du résultat optimal. Une seule se démarque : LRPT qui met un tiers du temps de la

deuxième plus rapide SPT. Elle est donc bien plus efficace.

Ajoutons maintenant l'amélioration EST et nous voyons :
- Une nette amélioration de toutes les règles de bases précédentes,
- Un écart restreint entre LRPT et ses concurrentes.

Cet écart considérable comme le montre le tableau ci-dessous montre bien que des grandes périodes de vide étaient laissées dans l'utilisation des machines, chose corrigé en grande partie par EST.

Ecarts entre les méthodes et l'optimal				
	SPT	LPT	SRPT	LRPT
Basique	162.4	201.3	245.5	60.1
EST	19.6	28.5	26.6	**16.3**
% EST du temp basique	12%	14%	11%	27%

En moyenne, l'utilisation de EST donne un résultat mettant seulement 16% du temps initial. La meilleure méthode étant EST_LRPT, c'est elle que nous utiliserons comme base pour la suite.

3 Methode de Descente

3.1 Trouver des voisins

La méthode de descente n'a plus pour but de trouver un agencement de tâches mais de regarder parmis les agencements possible, ceux qui fournissent un meilleur résultat. Nous allons donc devoir définir un ce qu'est un voisin et comment le trouver. Pour cela, nous regardons dans une solution son chemin critique et nous recherchons les différents blocs. Nous pourrons alors échanger des tâches entre elles en gardant la validité du resultat pour essayer d'améliorer celui-ci.

3.1.1 Recherche des blocs

Cette première tâche consiste a regarder les séquences de taches consécutives utilisant la même ressource.

```
Data : RessourceOrder order
Result : Liste des blocs
criticalPath ← order.criticalPath;
blocklist ← null;
for machine ∈ order.machine do
    job ← 0;
    tant que job < order.jobs − 1 faire
        if tache[machine][job] ∈ criticalPath then
            criticalPathPointeur ← criticalPath.IndexOf(tache[machine][job]);
            if machine = criticalPath.get(criticalPathPointeur + 1).machine then
                first_task ← job;
                tant que machine = criticalPath.get(criticalPathPointeur + 1).machine
                 faire
                    criticalPathPointeur++;
                    job++;
                fin
                blocklist.add(Block(machine, first_task, job);
            end
        end
        job++;
    fin
end
return blocklist;
```

Cet algorithme permet de récupérer tous les blocks d'une solution donnée. C'est à partir de cela que nous allons pouvoir trouver des voisins. Il suffit d'échanger l'ordre des tâches au sein d'un block.

3.1.2 Echanger des tâches

C'est là dedans que réside ce que l'on appelle un voisin. En échangeant deux tâches de places, nous obtenons une solution proche de la précédente mais dont le résultat peut être meilleur ou pas. Nous avons trouvé grâce au blocs des tâches échangeable. Dorénavant il faut pouvoir faire tous les changements possibles sur un block afin d'avoir accès à ces voisins. Voici comment j'ai implémenté ceci :

Data : block
Result : Liste des échanges possibles

```
firstTask ← block.firstTask;
lastTask ← block.lastTask;
if firstTask +1 = lastTask then
    swapList.add(Swap(machine,firstTask,lastTask));
else
    swapList.add(Swap(machine,firstTask,firstTask +1));
    swapList.add(Swap(machine,lastTask -1,lastTask));
end
swapListlist ← null;
return swapList;
```

Ainsi dans la prochaine partie, nous serons en mesure de faire les échanges ainsi préparé, c'est à dire la première tache et la seconde puis la dernière et l'avant dernière de chaque block.

3.2 Explorer le voisinage

L'essence de cet algorithme est de regarder les voisins d'une solution tant qu'elle en possède qui l'améliore. La méthode de Descente part avec pour base un résultat glouton (nous prennons donc EST LRPT, c'est pour cela qu'on le comparera à cette heuristique) et explore le voisinage de cette solution pour en extraire une meilleure solution.

Data : instance, deadline
Result : result

```
order ← greedyEST_LRPT.solve(instance);
tant que amliorationOrder and tempsExecution < deadline faire
    for Swap voisin ∈ listeVoisins(blocsListe) do
        orderToTest ← voisin(order);
        if orderToTest.makespan <order.makespan then
            order ← orderToTest;
        end
    end
fin
if tempsExecution <deadline then
    return result(instance, order.toSchedule, ProvedOptimal);
else
    return result(instance, order.toSchedule, Timeout);
end
```

Voici donc le comparatif de cette méthode avec l'heuristique gloutonne sur laquelle elle se base :

```
emile@A1L1:~/Documents/INSA/sem8/metha/template-jobshop$ java -jar build/libs/J
```

			greedyESTLRPT			descent		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	8	11	0,0	12	11	0,0
ft10	10x10	930	3	1145	23,1	154	1096	17,8
ft20	20x5	1165	0	1538	32,0	34	1523	30,7
la01	10x5	666	0	701	5,3	4	695	4,4
la02	10x5	655	0	817	24,7	4	806	23,1
la03	10x5	597	0	764	28,0	5	761	27,5
la04	10x5	590	0	758	28,5	4	747	26,6
la05	10x5	593	0	593	0,0	2	593	0,0
la06	15x5	926	0	949	2,5	16	926	0,0
la07	15x5	890	0	935	5,1	13	917	3,0
la08	15x5	863	1	974	12,9	17	947	9,7
la09	15x5	951	0	1015	6,7	16	951	0,0
AVG	-	-	1,0	-	14,1	23,4	-	11,9

Fig. 3 – *Comparatif entre la méthode de descente et l'algorithme glouton sur lequelle elle se base*

On observe alors deux choses

- Même si le temps d'exécution reste nul à l'échelle de l'Homme, on remarque qu'il a quand même été multiplié par 23 comparativement à la méthode gloutonne : en effet, si la méthode gloutonne se contentait de remplir un ordre selon une règle, la méthode de Descente commence à explorer l'espace de recherche précédement calculé. Seulement une infime prtie au voisinage du résultat glouton certes mais nous somme passé à un étage supérieur, cela se constate sur le runtime.
- Deuxième chose, l'amélioration du résultat. Nous ne sommes pas certes sur le gain de EST par rapport aux règles basiques mais la méthodes de Descente réduit de 15% le résultat glouton, ce qui est, encore une fois une amélioration souhaitable. Nous voyons d'ailleurs que nous obtenons plus de résultats optimaux.

Nous avons donc changé d'approche : nous ne recherchons plus une solution à peu près satisfaisante en un temps minimal mais une solution optimale, au moins localement, quitte à sacrifier quelques milisecondes de temps de calcul. Et c'est ce que nous allons essayer de faire encore mieux en utilisant la métha Heuristique Taboo pour trouver cette fois-ci, des optimums globaux.

4 Methode Taboo

4.1 Principe de la méthode

La méthode taboo fonctionne de manière très similaire à la méthode de descente : nous allons observer le voisinage d'une solution gloutonne pour en trouver de meilleurs résultats. La différence réside cette fois ci, non pas dans la manière de trouver des voisins mais dans la manière de les explorer. Nous partirons donc des fonctions précédemment implémentés, c'est la méthode Solve() qui diffère.

Dans cette méthode, nous ne nous arrêtons pas lorsqu'une solution ne trouve pas de voisins l'améliorant. A la place, nous enregistrerons la meilleure solution trouvée mais nous continuerons de regarder le meilleur voisins, quand bien même son résultat est moins bon que le précédent. Ainsi, nous ne nous arrêterons pas à un minimum local mais nous explorerons un plus grand panel de solutions.

Deux choses sont cependant à mettre en place pour que cette métha-heuristique puisse fonctionner :

- **Un nombre d'Itération maximum** pour ne pas parcourir l'ensemble de l'espace de recherche et retomber sur les miliers d'années nécéssaire à cela.
- **Un temps «taboo»** pendant lequel les solutions ne peuvent être de nouveau explorée et ce pour ne pas tourner en rond.

Voici le pseudo code de mon implémentation :

```
Data : instance, deadline
Result : result
s⋆ ← greedyEST_LRPT.solve(instance);
s ← s′ ← s″ ← s⋆;
sTaboo[nbTaskTotal][nbTaskTotal] ← 0;
k ← 0;
tant que k < maxIter and tempsExecution < deadline faire
    k++;
    updated ← false;
    makespan′ ← MAX_INT;
    s ← s′;
    for Swap voisin ∈ listeVoisins(blocsListe) do
        s″ ← voisin(s);
        if sTaboo[s″[0]][s″[1]] < k then
            updated ← true;
            if s″.makespan < makespan′ then
                s′ ← s″;
                makespan′ ← s″.makespan;
            end
        end
    end
    if updated then
        sTaboo[s″[1]][s″[0]] ← k + dureeTaboo
    end
fin
if tempsExecution <deadline then
    return result(instance, order.toSchedule, ProvedOptimal);
else
    return result(instance, order.toSchedule, Timeout);
end
```

Nous pouvons désormais tester cette méthode en faisant varier les paramètres MaxIter et durée-Taboo.

- **MaxIter** donne la partie de l'espace de recherche explorée. Un maxIter de 0 correspond à l'algorithme glouton tout simple, plus le maxIter est grand, plus on autorise à rechercher loin dans l'espace de recherche.

– **duréeTaboo** donne le temps pendant lequelle une solution ne peut être explorée. Une duréeTaboo égale à maxIter interdit de revisiter une solution déjà explorée, cela n'implique pas que nous allons forcément observer autant de solutions que maxIter : en effet, il est possible d'atteindre un «cul de sac», c'est à dire une solution dont tous les voisins ont été exploré et sont donc devenu taboo. Dans ce cas là, la solution ne s'améliorera pas comme nous le verrons dans les test ci-dessous. En revanche une duréeTaboo de 1 annule l'intérêt d'un tel paramètre qui était pourtant nécéssaire afin, comme expliqué plus haut, de ne pas tourner en rond.

4.2 Analyse des résultats

4.2.1 Les valeurs extrêmes

```
emile@A1L1:~/Documents/INSA/sem8/metha/template-jobshop$ java -jar build/libs/JSP.jar --solver greedyESTLRPT descent taboo_max_1 taboo_max_max
06 ft10 ft20 la01 la02 la03 la04 la05 la06 la07 la08 la09 -t 20
                              greedyESTLRPT                  descent                        taboo_max_1                    taboo_max_max
instance size   best          runtime makespan ecart         runtime makespan ecart         runtime makespan ecart         runtime makespan ecart
aaa1     2x3      11                8       11   0,0              12       11   0,0           19999       11   0,0           19999       11   0,0
ft06     6x6      55                1       65  18,2               3       58   5,5           19999       58   5,5           19999       58   5,5
ft10     10x10   930                0     1145  23,1              52     1096  17,8           19999     1096  17,8           20002     1096  17,8
ft20     20x5   1165                0     1538  32,0              13     1523  30,7           20000     1458  25,2           19999     1520  30,5
la01     10x5    666                0      701   5,3               3      695   4,4           19999      695   4,4           19999      695   4,4
la02     10x5    655                0      817  24,7               2      806  23,1           20000      768  17,3           19999      806  23,1
la03     10x5    597                0      764  28,0               3      761  27,5           19999      759  27,1           19999      759  27,1
la04     10x5    590                0      758  28,5               2      747  26,6           19999      696  18,0           19999      747  26,6
la05     10x5    593                0      593   0,0               2      593   0,0           19999      593   0,0           19999      593   0,0
la06     15x5    926                0      949   2,5              10      926   0,0           19999      926   0,0           20000      926   0,0
la07     15x5    890                0      935   5,1               8      917   3,0           19999      917   3,0           19999      917   3,0
la08     15x5    863                0      974  12,9               8      947   9,7           20000      908   5,2           19999      947   9,7
la09     15x5    951                0     1015   6,7               9      951   0,0           20000      951   0,0           19999      951   0,0
AVG      -         -              0,7        -  14,4             9,8        -  11,4         19999,3        -   9,5         19999,3        -  11,4
```

FIG. 4 – *Résultat taboo pour valeurs extrêmes*

Voici les résultats de ces extrêmes. J'ai appliqué Integer.MAX_VALUE en nombre d'itération maximum et soit 1 soit le même maximum pour la durée taboo. Ce que l'on constate c'est que le nombre d'instances optimisés par taboo n'augmente pas par rapport à sa soeur descente.
Dans le cas d'une durée taboo infinie, on obtiens environ le même résultat que par descente. C'est normal : c'est ce que l'on intuitait plus haut, si une solution ne peut être réexplorée, alors quand aucune amélioration n'est possible, l'algorithme entre en «cul de sac». La méthode de descente détecte toute seule cet état en constatant qu'aucune amélioration n'à été faite là où la mètha heuristique taboo continue de boucler dans le vide.
Dans le cas où l'on enlève la durée taboo (duréeTaboo = 1) on voit que les résultats obtenus sont légèrement meilleurs, mais nous n'avons pas atteint plus d'optimum, et des résultats identiques sont obtenu en réduisant le temps d'exécution, nous pouvons en déduire que nous avons tourné en rond.

Il aurait été intéressant de tester toutes les solutions possible entre de telles extrêmes mais rien que pour tester toutes les combinaisons possibles pour un nombre d'itération maximum de 1000 avec 5 secondes de runtime max, le temps de tout tester peut monter jusqu'à 2 mois et demi. Voici donc quelques valeurs expérimentées.

4.2.2 L'impact de la durée taboo

Afin d'essayer d'entrevoir l'impact des paramètres sur l'exécution de la méthode j'ai fixé deux paramètres pour en faire varier un troisième. Ici nous testerons la durée taboo.
Paramètres : MaxIter = 100, timeout = 1;
Les résultats sont visible dans le dépot Git dans le dossier test, il s'agit du fichier testpercentresults.
On observe plusieurs choses : pour duréeTaboo >78, la durée taboo n'a plus d'impact : nous sommes dans une situation de «cul de sac» et les recherches ne portent plus sur des résultats intéressant qui sont voisins de solutions inexplorables à cause du taboo. La limite est à 78 mais dès 52 on observe ce

phénomène de stagnation.
Pour duréeTaboo <50 en revanche, l'écart moyen avec l'optimal est borné entre 4.2 et 6. La durée taboo à donc un impact à ce niveau là mais reste très léger, le minimum étant atteint pour 12 et 13.

4.2.3 L'impact du timeout ou MaxIter

Que l'on change la valeur du paramètre timeout ou MaxIter, le résultat est identique : ce changement influe sur la quantité de tour que nous ferons dans la boucle : si en t secondes nous pouvons faire i itérations, alors mettre un timeout de $2 \times t$ sans changer maxIter ne changera pas le résultat. Les deux doivent aller de pair. Nous mettons ceci en évidence dans le tableau ci-dessous lorsque l'augmentation du timeout cesse d'avoir un impact.
Paramètres : MaxIter = 2000, duréeTaboo = 10;

Impact du timeout	
Timeout	Ecart moyen avec l'optimal
1	2.8
2	1.9
5	1.5
7	1.3
8	1.3
>8	1.1

Nous avons donc ici un résulat qui s'améliore en fonction du temps d'éxécution. En revanche, quand il dépasse un certain seuil, ce n'est plus lui qui limite l'exécution mais maxIter. On constate en effet que le runtime n'augmente plus à partir de timeout=9. C'est bien que le facteur limitant du nombre de boucles n'est plus le timeout mais maxIter.

4.3 Résultat de la méthode

Le meilleur résultat testé ici est celui pour MaxIter=2000, dureeTaboo= 13 et timeout= 9 qui obtient un écart moyen avec l'optimal de 1. Ce résultat est remarquable car pour les quelques instances pour lesquelles il n'est pas optimal, il en est très proche.

la01 la02 la03 la04 la05 la06 la07 la08 la09 -t 9

			greedyESTLRPT			descent			taboo2000_13		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	7	11	0,0	11	11	0,0	224	11	0,0
ft06	6x6	55	0	65	18,2	4	58	5,5	1716	55	0,0
ft10	10x10	930	0	1145	23,1	45	1096	17,8	9001	962	3,4
ft20	20x5	1165	1	1538	32,0	13	1523	30,7	7408	1212	4,0
la01	10x5	666	0	701	5,3	3	695	4,4	1927	666	0,0
la02	10x5	655	0	817	24,7	3	806	23,1	2279	664	1,4
la03	10x5	597	0	764	28,0	2	761	27,5	2082	620	3,9
la04	10x5	590	0	758	28,5	3	747	26,6	2267	593	0,5
la05	10x5	593	0	593	0,0	2	593	0,0	1929	593	0,0
la06	15x5	926	0	949	2,5	9	926	0,0	3946	926	0,0
la07	15x5	890	0	935	5,1	7	917	3,0	4192	890	0,0
la08	15x5	863	0	974	12,9	7	947	9,7	3987	863	0,0
la09	15x5	951	0	1015	6,7	10	951	0,0	3925	951	0,0
AVG	-	-	0,6	-	14,4	9,2	-	11,4	3452,5	-	1,0

FIG. 5 – *Résultat du meilleur Taboo expérimenté*

5 Comparaison des méthodes

Voici un résultat comparatif de toutes les méthodes implémentées (pour l'algorithme glouton il s'agit de la règle donnant le meilleurs résultat) :

la09 -t 9

			basic			random			greedyESTLRPT			descent			taboo2000_13		
instance	size	best	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart	runtime	makespan	ecart
aaa1	2x3	11	2	12	9,1	8999	11	0,0	6	11	0,0	11	11	0,0	201	11	0,0
ft06	6x6	55	0	60	9,1	8999	55	0,0	1	65	18,2	2	58	5,5	1326	55	0,0
ft10	10x10	930	0	1319	41,8	8999	1156	24,3	1	1145	23,1	44	1096	17,8	8908	962	3,4
ft20	20x5	1165	0	1672	43,5	8999	1461	25,4	1	1538	32,0	11	1523	30,7	7038	1212	4,0
la01	10x5	666	0	858	28,8	8999	681	2,3	0	701	5,3	2	695	4,4	1838	666	0,0
la02	10x5	655	0	904	38,0	8999	729	11,3	0	817	24,7	2	806	23,1	2116	664	1,4
la03	10x5	597	0	775	29,8	8999	655	9,7	0	764	28,0	2	761	27,5	2073	620	3,9
la04	10x5	590	0	854	44,7	8999	631	6,9	0	758	28,5	2	747	26,6	2208	593	0,5
la05	10x5	593	0	629	6,1	8999	593	0,0	0	593	0,0	1	593	0,0	1862	593	0,0
la06	15x5	926	0	1015	9,6	8999	935	1,0	0	949	2,5	9	926	0,0	3662	926	0,0
la07	15x5	890	0	1096	23,1	8999	950	6,7	0	935	5,1	7	917	3,0	3872	890	0,0
la08	15x5	863	0	1102	27,7	8999	913	5,8	0	974	12,9	6	947	9,7	3674	863	0,0
la09	15x5	951	0	1024	7,7	8999	955	0,4	0	1015	6,7	9	951	0,0	3642	951	0,0
AVG	-	-	0,2	-	24,5	8999,0	-	7,2	0,7	-	14,4	8,3	-	11,4	3263,1	-	1,0

Fig. 6 – *Résultats comparatifs*

Ce que l'on retiendra principalement de cette exécution est le tableau ci dessous :

Résultats comparatifs					
	basic	random	greedy EST LRPT	descente	taboo
AVG runtime % de AVG basique	24.5 0.2 100%	7.2 8999.0 29.38%	14.4 0.7 58.77%	11.4 8.3 46%	1 3263.1 4.08%

On remarque d'abord comme attendu que c'est le résultat taboo qui est le meilleur. Cependant nous basions l'heuristique de descente et la métha heuristique taboo sur l'algorithme glouton et nous observons ici que random donne un meilleur résultat. Devrions-nous reconsidérer la base de nos méthodes ? La réponse est bien évidemment non. En effet, si random a ici donné un meilleur résultat, ce n'est pas toujours le cas, ça dépent du timeout. En revanche son temps d'exécution lui reste toujours le maximum qui lui est donné. Cela viendrait mettre en péril la rapidité des autres heuristiques qui ne pourraient alors pas fonctionner correctement si tout le temps qui leur est imparti est utilisé par random. C'en serait d'autant plus dommage que descente, mais même taboo dans certains cas, n'utilisent pas tous le temps nécéssaire. Enfin, quitte à explorer l'espace de recherche des solutions, autant le faire intelligemment (j'entends en suivant une logique) plutôt qu'aléatoirement.

6 Conclusion

Ce TP à permis de mettre en place différentes méthodes heuristiques. J'ai pu voir au travers d'un problème courant mais complexe à résoudre l'intérêt de ne pas rechercher une solution «à la main» ou du moins en brute-force quand cela demande des années mais d'utiliser de l'intelligence pour trouver une meilleure solution. Là ou certains diraient qu'il ne faut pas passer plus de temps à chercher une méthode qu'à trouver la solution, en quelques mois j'ai pu implémenter une méthode et trouver une solution à un problème qui en aurait demandé des puissances de dix.

Mais le vrai coeur de ce TP à été d'expérimenter une solution bien plus efficace à tout problème : les métha heuristiques. Tabou est l'une d'elle et est applicable à bien plus de problèmes que jobshop et dépasse les minimas locaux qui pouvait bloquer les heuristiques classiques telle que la méthode de descente. Je me suis même demandé s'il ne fallait pas que je l'applique à elle même pour trouver les meilleurs paramètres maxIter et duréeTaboo.