# Rapport du TP

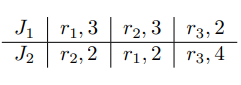
## Introduction

Le but du problème à résoudre dans ce projet est de trouver un ordonnancement autorisé d'une liste de tâches de telle sorte que la durée de l'ordonnancement soit la plus courte possible. On complètera d'abord les types de représentation qui serviront pour la suite. Ensuite on verra le développement de différentes méthodes pour résoudre le problème du job shop. Les méthodes seront testées sur des instances de problèmes connus que l'on comparera avec la meilleure solution connu actuellement pour chaque instance. On verra les résultats des solveurs développés ainsi que leur analyse. On verra aussi les méthodes exhaustives par le biais de 4 questions. Enfin nous concluront après avoir comparé les solveurs entre eux et l'impact des paramètres de ces solveurs.

## Manipulation de représentations

**Pour la solutions en représentation par numéro de job donnée, calculez (à la main) les dates de début de chaque tâche**

L'instance du problème aa11 est la suivante :

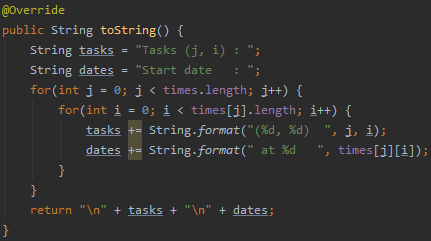


Une solution (sous forme de numéro de jobs) est : [0 1 1 0 0 1].

Les dates de début au plus tôt sont :

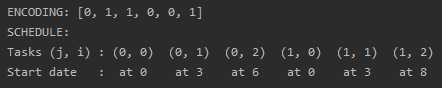
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tâche (j, i) | (0, 0) | (1, 0) | (1, 1) | (0, 1) | (0, 2) | (1, 2) |
| Date au plus tôt | 0 | 0 | 3 | 3 | 6 | 8 |

**Implémentez la méthode toString() de la classe Schedule pour afficher les dates de début de chaque tâche dans un schedule.**

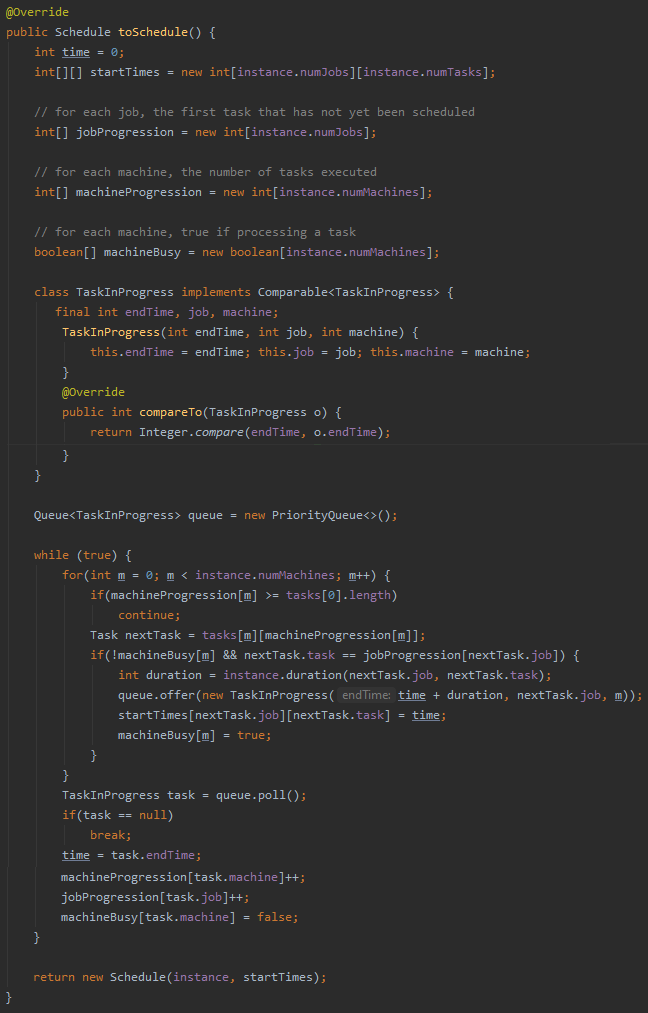


**Vérifiez que ceci correspond bien aux calculs que vous aviez fait à la main.**

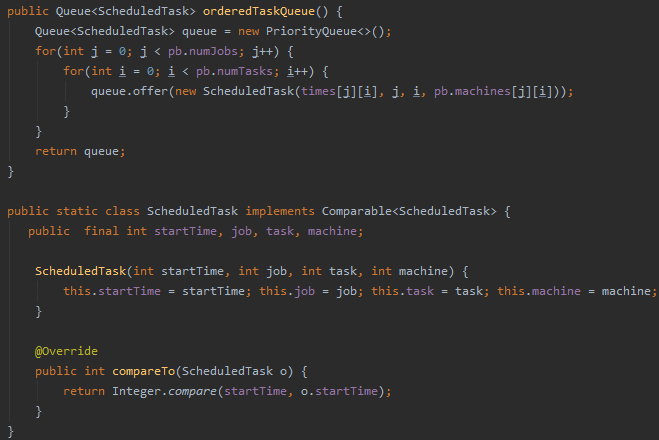
Ça affiche ça et ça correspond bien (même si c'est pas affiché dans le même ordre).

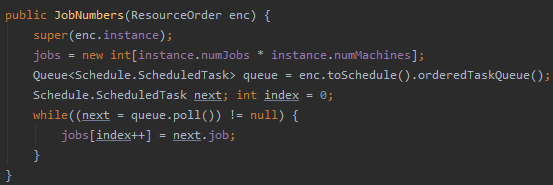


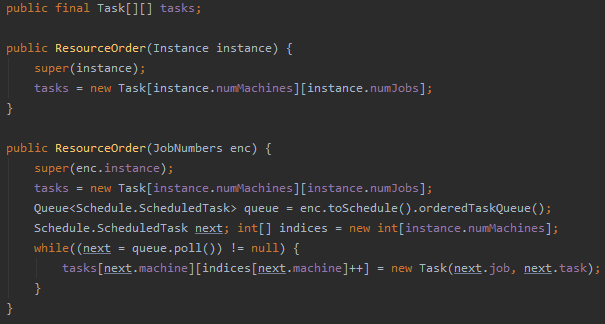
**Créez une classe ResourceOrder, implémentez toSchedule(), et ajoutez des tests dans EncodingTests**



**Changement de représentation entre ResourceOrder et JobNumbers**







## Heuristique gloutonne

J'avais codé l'algorithme avant que le sujet ne soit modifié, j'ai décidé de le laisser comme ça et de rajouter SRPT et LRPT afin de ne pas prendre de retard sur le reste des TP. Il y aura donc des différences par rapport au sujet modifié. En particulier la première version qui était demandée pour l'algo glouton devait sélectionner la tâche la plus prioritaire parmi les tâches possibles au temps t, et non pas parmi toutes les tâches possibles. Ça correspond donc à l'amélioration demandée dans le nouveau sujet.

**Explications sur l'algo que j'ai choisi :**

Il y a une queue des tâches candidates au temps t qui sert à récupérer la tâche de plus haute priorité (basée sur un binary heap). Pour l'initialisation, on place la première tâche de chaque job dans la queue des candidates (elles sont possibles au temps 0 car aucune machine n'est utilisée, mais elles ne seront pas forcément toutes possibles à la fois).

Ensuite à chaque itération de l'algorithme on va :

* lancer le maximum de tâches de la candidate queue (en prenant la plus prioritaire en premier et en vérifiant que les suivantes soient toujours possibles), si une tâche de cette queue n'est plus possible car la machine sur laquelle elle devait s'effectuer est prise, on va remettre cette tâche comme candidate pour la prochaine itération de l'algorithme (pour cela on utilise une autre queue qu'on swapera avec la première pour des raisons de performances), lorsqu'on lance une tâche on la place dans la running queue qui est une priority queue qui trie par date de fin au plus tôt, on va aussi construire la solution en fixant ce choix
* on utilise la running queue pour déterminer la prochaine tâche qui se finira et on met à jour le temps t, on va donc libérer toutes les machines qui sont maintenant disponibles au temps t (il y a d'autres tâches qui sont encore dans la running queue car elles ne finissent pas au temps t) et on va bufferizer les tâches qui viennent de finir pour la prochaine étape, mais on va aussi récupérer les tâches pending qui attendaient pour une machine qui vient de se libérer et on va les mettre dans la candidate queue (voir ci-dessous)
* chaque tâche finie va rendre potentiellement candidate la tâche suivante du même job, c'est ce qu'on va vérifier à cette étape, mais si la tâche suivante du job n'est pas possible à cause d'une machine prise, alors on va stocker cette tâche comme tâche pending dans un buffer pour cette machine là, et dès que cette machine sera libérée, la tâche sera ajoutée comme candidate (voir ci-dessus), on sépare en 2 étapes en passant par un buffer de finishing tasks afin de respecter l'algorithme glouton du sujet original car si on faisait tout à la même étape la disponibilité des machines serait différente, donc on libère d'abord toutes les machines qu'on peut avant d'ajouter les tâches qui sont devenues candidates à la candidate queue

L'algorithme se termine quand il n'y a plus de running tasks, la solution est construite avec la représentation ResourseOrder qu'on convertit en schedule. J'ai rajouté l'exit cause NotProvedOptimal car c'est un algorithme glouton. Pour prendre en compte les mode de priotiry SRPT et LRPT on calcule au début les remaining time de chaque job et on les mets à jour à chaque tâche finie.

**Résultats**

Il y a la qualité de la solution trouvée par l'algorithme glouton en pourcentage supplémentaire par rapport à la meilleure solution, et il y a le temps d'exécution de l'algorithme. Dans toute la suite du rapport j'ai fait les tests avec les instances de la01 à la40 et de ta61 à ta70. J'ai par contre observé que suivants quelles instances on demandait de tester en premier dans la liste d'instances passée en paramètre, les temps d'exécution étaient très différents: les premières instances étaient toujours longues alors que les suivantes de plus en plus rapide. Je pense que c'est du à la phase d'initialisation de la jvm ou aussi le fait que le just in time compiler mais du temps avant de tout optimiser comme il le veut. J'ai donc mis le code qui lance l'algo sur les instances dans une boucle qui fait 10 itérations, les résultats sont donc affichées 10 fois mais comme prévu les temps d'exécutions baisse rapidement avant de se stabiliser. Voici donc les résultats de la 1ère et de la 10ème itération :

**Priority SPT (1ère iteration)**

instance size best runtime makespan ecart

ta61 50x20 2868 26 3674 28,1

ta62 50x20 2869 9 3690 28,6

ta63 50x20 2755 10 3339 21,2

ta64 50x20 2702 7 3447 27,6

ta65 50x20 2725 5 3403 24,9

ta66 50x20 2845 3 3576 25,7

ta67 50x20 2825 2 3431 21,5

ta68 50x20 2784 3 3451 24,0

ta69 50x20 3071 16 3833 24,8

ta70 50x20 2995 2 3817 27,4

la01 10x5 666 0 751 12,8

la02 10x5 655 0 821 25,3

la03 10x5 597 0 672 12,6

la04 10x5 590 0 711 20,5

la05 10x5 593 1 610 2,9

la06 15x5 926 0 1200 29,6

la07 15x5 890 0 1034 16,2

la08 15x5 863 0 966 11,9

la09 15x5 951 0 1045 9,9

la10 15x5 958 0 1049 9,5

la11 20x5 1222 0 1496 22,4

la12 20x5 1039 0 1305 25,6

la13 20x5 1150 1 1474 28,2

la14 20x5 1292 0 1427 10,4

la15 20x5 1207 1 1331 10,3

la16 10x10 945 1 1265 33,9

la17 10x10 784 0 924 17,9

la18 10x10 848 0 981 15,7

la19 10x10 842 0 940 11,6

la20 10x10 902 0 1000 10,9

la21 15x10 1046 0 1324 26,6

la22 15x10 927 0 1180 27,3

la23 15x10 1032 0 1162 12,6

la24 15x10 935 0 1171 25,2

la25 15x10 977 0 1449 48,3

la26 20x10 1218 0 1624 33,3

la27 20x10 1235 0 1804 46,1

la28 20x10 1216 1 1610 32,4

la29 20x10 1152 0 1556 35,1

la30 20x10 1355 1 1792 32,3

la31 30x10 1784 0 2001 12,2

la32 30x10 1850 0 2204 19,1

la33 30x10 1719 1 1901 10,6

la34 30x10 1721 0 2025 17,7

la35 30x10 1888 1 2145 13,6

la36 15x15 1268 1 1799 41,9

la37 15x15 1397 0 1655 18,5

la38 15x15 1196 1 1407 17,6

la39 15x15 1233 0 1534 24,4

la40 15x15 1222 0 1476 20,8

AVG - - 1,9 - 22,1

**Priority SPT (10ème itération)**

instance size best runtime makespan ecart

ta61 50x20 2868 1 3674 28,1

ta62 50x20 2869 2 3690 28,6

ta63 50x20 2755 1 3339 21,2

ta64 50x20 2702 1 3447 27,6

ta65 50x20 2725 1 3403 24,9

ta66 50x20 2845 1 3576 25,7

ta67 50x20 2825 1 3431 21,5

ta68 50x20 2784 2 3451 24,0

ta69 50x20 3071 2 3833 24,8

ta70 50x20 2995 3 3817 27,4

la01 10x5 666 0 751 12,8

la02 10x5 655 0 821 25,3

la03 10x5 597 0 672 12,6

la04 10x5 590 0 711 20,5

la05 10x5 593 0 610 2,9

la06 15x5 926 1 1200 29,6

la07 15x5 890 0 1034 16,2

la08 15x5 863 0 966 11,9

la09 15x5 951 0 1045 9,9

la10 15x5 958 0 1049 9,5

la11 20x5 1222 0 1496 22,4

la12 20x5 1039 0 1305 25,6

la13 20x5 1150 0 1474 28,2

la14 20x5 1292 0 1427 10,4

la15 20x5 1207 0 1331 10,3

la16 10x10 945 0 1265 33,9

la17 10x10 784 0 924 17,9

la18 10x10 848 0 981 15,7

la19 10x10 842 0 940 11,6

la20 10x10 902 0 1000 10,9

la21 15x10 1046 1 1324 26,6

la22 15x10 927 0 1180 27,3

la23 15x10 1032 0 1162 12,6

la24 15x10 935 0 1171 25,2

la25 15x10 977 0 1449 48,3

la26 20x10 1218 0 1624 33,3

la27 20x10 1235 0 1804 46,1

la28 20x10 1216 0 1610 32,4

la29 20x10 1152 0 1556 35,1

la30 20x10 1355 0 1792 32,3

la31 30x10 1784 1 2001 12,2

la32 30x10 1850 1 2204 19,1

la33 30x10 1719 1 1901 10,6

la34 30x10 1721 1 2025 17,7

la35 30x10 1888 0 2145 13,6

la36 15x15 1268 0 1799 41,9

la37 15x15 1397 0 1655 18,5

la38 15x15 1196 0 1407 17,6

la39 15x15 1233 0 1534 24,4

la40 15x15 1222 0 1476 20,8

AVG - - 0,4 - 22,1

L'écart avec le makespan est le même à chaque itération (c'est normal il trouve les même résultats) par contre les premières instances sont beaucoup plus rapides à résoudre lors de la 10ème itération que lors de la première (en fait dès la 4ème itération les résultats étaient similaires à la 10ème). Ces tests ont été fait avec le mode de priorité SPT.

Maintenant voyons les résultats de la 10ème itération avec d'autres paramètres pour le mode de priorité.

**Priority LPT**

instance size best runtime makespan ecart

ta61 50x20 2868 2 4087 42,5

ta62 50x20 2869 2 4029 40,4

ta63 50x20 2755 2 4153 50,7

ta64 50x20 2702 2 4025 49,0

ta65 50x20 2725 1 4013 47,3

ta66 50x20 2845 2 4001 40,6

ta67 50x20 2825 2 4110 45,5

ta68 50x20 2784 2 4026 44,6

ta69 50x20 3071 2 4212 37,2

ta70 50x20 2995 4 4190 39,9

la01 10x5 666 0 822 23,4

la02 10x5 655 0 990 51,1

la03 10x5 597 0 825 38,2

la04 10x5 590 0 818 38,6

la05 10x5 593 0 693 16,9

la06 15x5 926 0 1125 21,5

la07 15x5 890 1 1069 20,1

la08 15x5 863 0 1035 19,9

la09 15x5 951 0 1183 24,4

la10 15x5 958 0 1121 17,0

la11 20x5 1222 0 1467 20,0

la12 20x5 1039 0 1240 19,3

la13 20x5 1150 0 1230 7,0

la14 20x5 1292 0 1434 11,0

la15 20x5 1207 1 1612 33,6

la16 10x10 945 1 1229 30,1

la17 10x10 784 0 1082 38,0

la18 10x10 848 0 1114 31,4

la19 10x10 842 0 1062 26,1

la20 10x10 902 0 1272 41,0

la21 15x10 1046 0 1451 38,7

la22 15x10 927 0 1315 41,9

la23 15x10 1032 0 1302 26,2

la24 15x10 935 0 1245 33,2

la25 15x10 977 2 1354 38,6

la26 20x10 1218 0 1643 34,9

la27 20x10 1235 1 1741 41,0

la28 20x10 1216 1 1844 51,6

la29 20x10 1152 0 1720 49,3

la30 20x10 1355 1 1953 44,1

la31 30x10 1784 0 2245 25,8

la32 30x10 1850 0 2437 31,7

la33 30x10 1719 1 2314 34,6

la34 30x10 1721 1 2330 35,4

la35 30x10 1888 0 2457 30,1

la36 15x15 1268 0 1949 53,7

la37 15x15 1397 0 1944 39,2

la38 15x15 1196 0 1732 44,8

la39 15x15 1233 0 1653 34,1

la40 15x15 1222 1 1840 50,6

AVG - - 0,6 - 34,9

**Priority SRPT**

instance size best runtime makespan ecart

ta61 50x20 2868 2 4074 42,1

ta62 50x20 2869 1 3828 33,4

ta63 50x20 2755 2 3522 27,8

ta64 50x20 2702 1 3664 35,6

ta65 50x20 2725 1 3700 35,8

ta66 50x20 2845 2 3872 36,1

ta67 50x20 2825 2 3732 32,1

ta68 50x20 2784 2 3823 37,3

ta69 50x20 3071 1 4190 36,4

ta70 50x20 2995 2 4233 41,3

la01 10x5 666 0 772 15,9

la02 10x5 655 0 835 27,5

la03 10x5 597 0 796 33,3

la04 10x5 590 0 786 33,2

la05 10x5 593 0 759 28,0

la06 15x5 926 1 1080 16,6

la07 15x5 890 0 1123 26,2

la08 15x5 863 0 940 8,9

la09 15x5 951 0 1146 20,5

la10 15x5 958 0 1113 16,2

la11 20x5 1222 0 1367 11,9

la12 20x5 1039 0 1303 25,4

la13 20x5 1150 0 1334 16,0

la14 20x5 1292 0 1599 23,8

la15 20x5 1207 0 1594 32,1

la16 10x10 945 0 1352 43,1

la17 10x10 784 0 1002 27,8

la18 10x10 848 0 1019 20,2

la19 10x10 842 0 944 12,1

la20 10x10 902 0 1233 36,7

la21 15x10 1046 0 1286 22,9

la22 15x10 927 0 1232 32,9

la23 15x10 1032 0 1235 19,7

la24 15x10 935 0 1156 23,6

la25 15x10 977 0 1301 33,2

la26 20x10 1218 0 1793 47,2

la27 20x10 1235 0 1557 26,1

la28 20x10 1216 0 1668 37,2

la29 20x10 1152 1 1650 43,2

la30 20x10 1355 0 1830 35,1

la31 30x10 1784 0 2195 23,0

la32 30x10 1850 0 2471 33,6

la33 30x10 1719 1 2275 32,3

la34 30x10 1721 1 2306 34,0

la35 30x10 1888 1 2328 23,3

la36 15x15 1268 0 1711 34,9

la37 15x15 1397 1 1781 27,5

la38 15x15 1196 1 1500 25,4

la39 15x15 1233 0 1548 25,5

la40 15x15 1222 0 1744 42,7

AVG - - 0,5 - 29,1

**Priority LRPT**

instance size best runtime makespan ecart

ta61 50x20 2868 2 3611 25,9

ta62 50x20 2869 1 3764 31,2

ta63 50x20 2755 2 3572 29,7

ta64 50x20 2702 2 3373 24,8

ta65 50x20 2725 2 3380 24,0

ta66 50x20 2845 3 3665 28,8

ta67 50x20 2825 2 3628 28,4

ta68 50x20 2784 1 3323 19,4

ta69 50x20 3071 2 3598 17,2

ta70 50x20 2995 1 3737 24,8

la01 10x5 666 0 772 15,9

la02 10x5 655 0 851 29,9

la03 10x5 597 0 741 24,1

la04 10x5 590 0 838 42,0

la05 10x5 593 0 615 3,7

la06 15x5 926 0 992 7,1

la07 15x5 890 0 1030 15,7

la08 15x5 863 0 965 11,8

la09 15x5 951 0 1018 7,0

la10 15x5 958 0 987 3,0

la11 20x5 1222 0 1260 3,1

la12 20x5 1039 0 1039 0,0

la13 20x5 1150 0 1170 1,7

la14 20x5 1292 0 1302 0,8

la15 20x5 1207 1 1424 18,0

la16 10x10 945 0 1152 21,9

la17 10x10 784 0 892 13,8

la18 10x10 848 0 998 17,7

la19 10x10 842 0 1013 20,3

la20 10x10 902 0 1225 35,8

la21 15x10 1046 0 1347 28,8

la22 15x10 927 0 1256 35,5

la23 15x10 1032 0 1178 14,1

la24 15x10 935 0 1149 22,9

la25 15x10 977 0 1283 31,3

la26 20x10 1218 0 1506 23,6

la27 20x10 1235 0 1552 25,7

la28 20x10 1216 0 1448 19,1

la29 20x10 1152 0 1488 29,2

la30 20x10 1355 1 1618 19,4

la31 30x10 1784 1 1932 8,3

la32 30x10 1850 1 1988 7,5

la33 30x10 1719 0 1873 9,0

la34 30x10 1721 0 2022 17,5

la35 30x10 1888 0 2224 17,8

la36 15x15 1268 0 1464 15,5

la37 15x15 1397 0 1628 16,5

la38 15x15 1196 0 1457 21,8

la39 15x15 1233 0 1538 24,7

la40 15x15 1222 0 1456 19,1

AVG - - 0,4 - 19,1

**Analyse des résultats**

Le makespan est le même à chaque itération car c'est une mesure qui ne dépend pas du temps d'exécution donc c'est normal. Le mode de priorité qui offre des résultats de meilleure qualité à l'air d'être le mode LRPT avec un makespan moyen de 19.1% par rapport à la meilleure solution connue. Ensuite le SPT est assez proche avec 22.1%, puis SRPT avec 29.1% et enfin LPT avec 34.9%. Les temps d'exécutions moyens pour les jeu de données testés vont de 0.4 à 0.6 millisecondes mais on ne peux pas faire un classement des mode de priorité car ces résultats fluctuent dans cet intervalle d'une exécution à l'autre.

## Méthode de la descente

L'algorithme utilise GluttonousSolver pour le point de départ comme indiqué dans le sujet. J'utilise une méthode clone() que j'ai rajouté qui permet de cloner un ResourceOrder.

Pour la méthode blocksOfCriticalPath, on mémorise la machine actuelle ainsi que la première tâche et on parcours les tâches du critical path. Tant qu'on reste sur la même machine on compte le nombre de tâches consécutives, et dès qu'on change de machine c'est que l'on à finit un block, donc on le créé et on le stocke dans la liste de blocks qui est retournée à la fin de la méthode.

Pour la méthode neighbors, on créé soit un soit deux objets swap suivant si le bloc est composé de respectivement 2 ou plus de 2 taches.

Pour la méthode solve, on récupère en utilisant les méthodes précédentes les voisin de la solution initiale, puis pour chacun d'eux on vérifie si le voisin est meilleur que la meilleure solution connue actuellement, et si c'est le cas on la remplace. Si une solution meilleure trouvée parmi les voisins, on recommence en utilisant cette fois les voisins de la nouvelle solution.

Le paramètre de priorité du GluttonousSolver est PRIORITY\_SPT.

**Résultats**

Les résultats sont ceux de la 10ème itération (pour que le startup-time n'influence pas trop les résultats). On fait varier le mode de priorité utilisé pour trouver la solution initiale.

**Priority SPT**

instance size best runtime makespan ecart

ta61 50x20 2868 6 3633 26,7

ta62 50x20 2869 14 3655 27,4

ta63 50x20 2755 14 3326 20,7

ta64 50x20 2702 18 3394 25,6

ta65 50x20 2725 43 3363 23,4

ta66 50x20 2845 34 3538 24,4

ta67 50x20 2825 22 3369 19,3

ta68 50x20 2784 19 3441 23,6

ta69 50x20 3071 16 3825 24,6

ta70 50x20 2995 42 3706 23,7

la01 10x5 666 0 686 3,0

la02 10x5 655 1 685 4,6

la03 10x5 597 0 666 11,6

la04 10x5 590 0 702 19,0

la05 10x5 593 0 610 2,9

la06 15x5 926 1 963 4,0

la07 15x5 890 0 1034 16,2

la08 15x5 863 0 916 6,1

la09 15x5 951 0 975 2,5

la10 15x5 958 0 1049 9,5

la11 20x5 1222 0 1390 13,7

la12 20x5 1039 1 1039 0,0

la13 20x5 1150 0 1394 21,2

la14 20x5 1292 0 1427 10,4

la15 20x5 1207 0 1330 10,2

la16 10x10 945 2 1099 16,3

la17 10x10 784 1 905 15,4

la18 10x10 848 0 981 15,7

la19 10x10 842 0 909 8,0

la20 10x10 902 1 959 6,3

la21 15x10 1046 1 1257 20,2

la22 15x10 927 1 1169 26,1

la23 15x10 1032 1 1141 10,6

la24 15x10 935 0 1136 21,5

la25 15x10 977 2 1293 32,3

la26 20x10 1218 3 1519 24,7

la27 20x10 1235 3 1705 38,1

la28 20x10 1216 1 1541 26,7

la29 20x10 1152 1 1522 32,1

la30 20x10 1355 3 1666 23,0

la31 30x10 1784 1 1909 7,0

la32 30x10 1850 2 2198 18,8

la33 30x10 1719 1 1862 8,3

la34 30x10 1721 4 1975 14,8

la35 30x10 1888 2 2100 11,2

la36 15x15 1268 8 1694 33,6

la37 15x15 1397 2 1605 14,9

la38 15x15 1196 2 1395 16,6

la39 15x15 1233 1 1534 24,4

la40 15x15 1222 4 1377 12,7

AVG - - 5,6 - 17,1

**Priority LPT**

instance size best runtime makespan ecart

ta61 50x20 2868 53 4033 40,6

ta62 50x20 2869 41 3929 36,9

ta63 50x20 2755 33 4065 47,5

ta64 50x20 2702 146 3834 41,9

ta65 50x20 2725 78 3949 44,9

ta66 50x20 2845 22 3971 39,6

ta67 50x20 2825 44 4091 44,8

ta68 50x20 2784 25 3932 41,2

ta69 50x20 3071 15 4182 36,2

ta70 50x20 2995 58 4087 36,5

la01 10x5 666 0 790 18,6

la02 10x5 655 1 774 18,2

la03 10x5 597 0 727 21,8

la04 10x5 590 0 770 30,5

la05 10x5 593 0 693 16,9

la06 15x5 926 0 1010 9,1

la07 15x5 890 0 1058 18,9

la08 15x5 863 0 1018 18,0

la09 15x5 951 1 1088 14,4

la10 15x5 958 0 1069 11,6

la11 20x5 1222 0 1446 18,3

la12 20x5 1039 0 1231 18,5

la13 20x5 1150 0 1230 7,0

la14 20x5 1292 1 1375 6,4

la15 20x5 1207 0 1599 32,5

la16 10x10 945 1 1164 23,2

la17 10x10 784 1 942 20,2

la18 10x10 848 0 1097 29,4

la19 10x10 842 0 1002 19,0

la20 10x10 902 1 1217 34,9

la21 15x10 1046 1 1343 28,4

la22 15x10 927 0 1315 41,9

la23 15x10 1032 1 1189 15,2

la24 15x10 935 1 1186 26,8

la25 15x10 977 0 1352 38,4

la26 20x10 1218 1 1592 30,7

la27 20x10 1235 0 1738 40,7

la28 20x10 1216 2 1733 42,5

la29 20x10 1152 4 1677 45,6

la30 20x10 1355 3 1873 38,2

la31 30x10 1784 3 2191 22,8

la32 30x10 1850 3 2372 28,2

la33 30x10 1719 8 2207 28,4

la34 30x10 1721 5 2253 30,9

la35 30x10 1888 3 2324 23,1

la36 15x15 1268 2 1925 51,8

la37 15x15 1397 4 1847 32,2

la38 15x15 1196 8 1610 34,6

la39 15x15 1233 9 1534 24,4

la40 15x15 1222 4 1826 49,4

AVG - - 11,7 - 29,4

**Priority SRPT**

instance size best runtime makespan ecart

ta61 50x20 2868 27 4062 41,6

ta62 50x20 2869 25 3815 33,0

ta63 50x20 2755 63 3481 26,4

ta64 50x20 2702 83 3632 34,4

ta65 50x20 2725 47 3689 35,4

ta66 50x20 2845 91 3785 33,0

ta67 50x20 2825 33 3662 29,6

ta68 50x20 2784 63 3801 36,5

ta69 50x20 3071 17 4187 36,3

ta70 50x20 2995 51 4178 39,5

la01 10x5 666 0 703 5,6

la02 10x5 655 0 835 27,5

la03 10x5 597 0 796 33,3

la04 10x5 590 0 752 27,5

la05 10x5 593 0 731 23,3

la06 15x5 926 0 1079 16,5

la07 15x5 890 0 1083 21,7

la08 15x5 863 0 929 7,6

la09 15x5 951 0 1056 11,0

la10 15x5 958 0 1057 10,3

la11 20x5 1222 0 1347 10,2

la12 20x5 1039 1 1232 18,6

la13 20x5 1150 0 1324 15,1

la14 20x5 1292 2 1363 5,5

la15 20x5 1207 1 1507 24,9

la16 10x10 945 1 1337 41,5

la17 10x10 784 2 881 12,4

la18 10x10 848 1 1006 18,6

la19 10x10 842 2 944 12,1

la20 10x10 902 1 1168 29,5

la21 15x10 1046 0 1286 22,9

la22 15x10 927 8 1202 29,7

la23 15x10 1032 1 1193 15,6

la24 15x10 935 0 1156 23,6

la25 15x10 977 1 1260 29,0

la26 20x10 1218 4 1672 37,3

la27 20x10 1235 2 1534 24,2

la28 20x10 1216 1 1620 33,2

la29 20x10 1152 5 1525 32,4

la30 20x10 1355 2 1787 31,9

la31 30x10 1784 11 2110 18,3

la32 30x10 1850 3 2259 22,1

la33 30x10 1719 7 2195 27,7

la34 30x10 1721 6 2135 24,1

la35 30x10 1888 6 2220 17,6

la36 15x15 1268 6 1667 31,5

la37 15x15 1397 1 1781 27,5

la38 15x15 1196 10 1425 19,1

la39 15x15 1233 3 1527 23,8

la40 15x15 1222 2 1724 41,1

AVG - - 11,8 - 25,0

**Priority LRPT**

instance size best runtime makespan ecart

ta61 50x20 2868 18 3587 25,1

ta62 50x20 2869 25 3681 28,3

ta63 50x20 2755 121 3267 18,6

ta64 50x20 2702 38 3280 21,4

ta65 50x20 2725 30 3325 22,0

ta66 50x20 2845 56 3529 24,0

ta67 50x20 2825 121 3522 24,7

ta68 50x20 2784 36 3236 16,2

ta69 50x20 3071 45 3508 14,2

ta70 50x20 2995 40 3701 23,6

la01 10x5 666 0 735 10,4

la02 10x5 655 0 760 16,0

la03 10x5 597 0 701 17,4

la04 10x5 590 0 678 14,9

la05 10x5 593 0 615 3,7

la06 15x5 926 0 970 4,8

la07 15x5 890 0 1030 15,7

la08 15x5 863 1 894 3,6

la09 15x5 951 1 951 0,0

la10 15x5 958 0 958 0,0

la11 20x5 1222 1 1238 1,3

la12 20x5 1039 0 1039 0,0

la13 20x5 1150 0 1150 0,0

la14 20x5 1292 1 1292 0,0

la15 20x5 1207 0 1424 18,0

la16 10x10 945 1 1065 12,7

la17 10x10 784 0 892 13,8

la18 10x10 848 0 997 17,6

la19 10x10 842 1 1008 19,7

la20 10x10 902 0 1221 35,4

la21 15x10 1046 0 1347 28,8

la22 15x10 927 1 1180 27,3

la23 15x10 1032 0 1178 14,1

la24 15x10 935 1 1115 19,3

la25 15x10 977 3 1242 27,1

la26 20x10 1218 4 1367 12,2

la27 20x10 1235 2 1489 20,6

la28 20x10 1216 1 1396 14,8

la29 20x10 1152 5 1388 20,5

la30 20x10 1355 2 1517 12,0

la31 30x10 1784 3 1866 4,6

la32 30x10 1850 3 1939 4,8

la33 30x10 1719 3 1845 7,3

la34 30x10 1721 2 1924 11,8

la35 30x10 1888 7 2128 12,7

la36 15x15 1268 2 1424 12,3

la37 15x15 1397 5 1588 13,7

la38 15x15 1196 3 1455 21,7

la39 15x15 1233 4 1520 23,3

la40 15x15 1222 3 1456 19,1

AVG - - 11,8 - 15,0

**Analyse des résultats**

Le mode de priorité qui offre des résultats de meilleure qualité à l'air d'être le mode LRPT avec un makespan moyen de 15.0% par rapport à la meilleure solution connue. Ensuite le SPT est assez proche avec 17.1%, puis SRPT avec 25.0% et enfin LPT avec 29.4%. Les temps d'exécutions moyens pour les jeu de données testés vont de 5 à 12 millisecondes. On pourrait penser d'après les résultats ci-dessus que le mode SPT est deux fois plus rapide que les autres, mais en réalité chaque mode varie assez significativement d'une itération à l'autre entre ces deux bornes.

## Méthodes exhaustives

**Question 1)**

La taille de l'espace de recherche de l'instance ft06 si l'on utilise la représentation par numéro de job est de

**Question 2)**

En supposant que la génération et l’évaluation d’une solution prend une nanoseconde, le temps nécessaire pour explorer l’espace de recherche associé à la représentation par numéro de job est de environ 30,9 milliards de jours. Si l'on considère la représentation par ordre de ressource, il descent à environ 1612 jours. Si l'on considère la représentation ar date de début de chaque tâche, le temps est d'environ 4,62e+68 jours.

**Question 3)**

Il faut bien choisir la représentation qu'on utilise car cela a un impact déterminant sur ce que l'on pourra trouver comme solution. Il faut essayer de se rapprocher au plus proche du nombre de solutions réel possibles. Pour cela il faut trouver une représentation qui élimine les symétries (plusieurs représentations différentes pour la même solution) et aussi les solutions impossibles.

**Questions 4)**

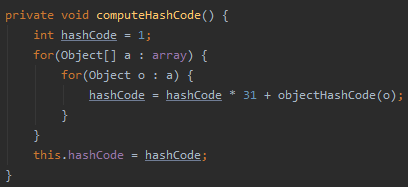
Les méthodes exhaustives sont assez limités en pratique à cause de la croissance souvent exponentielle de la taille de l'espace de recherche. Mais quand on peut l'utiliser elle peut être très utile car c'est une façon systématique de trouver la solution optimale, et on peut prédire en combien de temps on l'aura environ.

## Méthode du tabou

Pour la méthode taboo, l'algorithme ressemble pas mal à celui de la méthode descente mais on maintient une meilleure solution locale en plus. Pour tester si on a déjà vu une solution (solution tabou), j'utilise un HashSet car même avec les plus grosses instances de problèmes comme ta78 qui ont 2000 taches, il faudrait qu'on garde les solution tabou pendant plusieurs dizaines de milliers d'itération pour manquer de mémoire, et même avec un paramètre dureeTaboo=1000 on peut laisser travailler le solveur pendant aussi longtemps que nécessaire grâce au paramètre maxIteration. On utilise une queue dans laquelle on place les solutions tabou, et une fois que la queue atteint une taille égale à dureeTaboo, on va récupérer la première solution de la queue (qui à donc été trouvée il y a dureeTaboo itérations) et on va l'enlever du Set (et de la queue).

L'opération de swap pour trouver une solution voisine se faisait en O(1), et j'ai voulu conserver cette complexité dans le nouvel algorithme qui devait en plus recalculer le hashCode de la nouvelle solution après le swap et effectuer une recherche dans le HashSet. Cette dernière opération est déjà en O(1) (en moyenne si la fonction hashCode distribue assez bien les objets entre les buckets), par contre le hashCode d'une solution doit être calculé en prenant en compte toutes les taches, ce qui augmenterait le temps de la recherche. Pour éviter cela j'ai créé une classe HashCodeArrayWrapper qui va se charger de calculer (en O(n)), réutiliser et mettre à jour (en O(1)) le hashCode correspondant au contenu d'un tableau. Le wrapper propose des méthodes set et get afin d'accéder au tableau wrappé, afin qu'il puisse au passage modifier le hashCode de façon transparente pour l'utilisateur de la classe. Cela permet de swaper un bloc dans la solution puis de tester si la solution est une solution tabou en O(1).

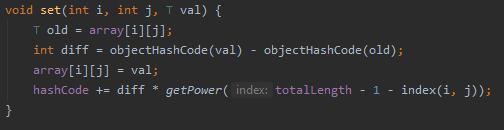
Le hashCode du tableau de taches d'une représentation en ResourceOrder est calculé la première fois de la façon suivante :



C'est une façon classique de calculer un hashCode pour une liste d'entiers (qu'on va appeler list dans la suite, les entiers de cette liste sont en fait les hashCode des objets Task du tableau). Grâce aux propriétés d'overflow des opération d'addition et de multiplication sur des int, le résultat du hashCode est le même que si l'on avait d'abord calculé le polynôme :

list[0]\*31^1000 + list[1]\*31^999 + ... + list[999]\*31^0

en utilisant autant de bits de stockage afin de ne pas faire d'overflow, puis qu'on garde les 32 bits de poids faible du résultat. Cela permet de savoir comment sera modifé le résultat du calcul du hashCode sans qu'on ai besoin de le recalculer entièrement : si l'on modifie par exemple le coefficient list[1] pour lui affecter le nombre 4 alors qu'il valait 2 avant, la valeur du polynôme sera augmentée de 2\*31^999. Heureusement on peux effectuer le calcul d'exponentiation en utilisant seulement 32 bits et en une opération : ont précalcule les 32 bits de poids faible de la 999ème puissance de 31, et toujours grâces aux propriétés des overflow, on a juste à multiplier ce reste par la différence entre la nouvelle valeur pour obtenir le même résultat que si l'on avait multiplié la différence par la vrai valeur de puissance.



J'ai testé par la suite avec ou sans l'accélération pour mettre à jour le hashCode (dont le temps de calcul varie de quelques opérations à plusieurs milliers pour les instances comme ta78), mais il s'avère que le temps total d'exécution du solveur ne varie pas significativement, ce qui laisse penser que le reste de l'algorithme qui doit notamment calculer des schedules ainsi que le premier hashCode d'une itération prenne plus de temps. L'accélération du calcul de hashCode n'est donc pas nécessaire pour cet algorithme et il est aussi rapide avec un simple HashSet.

**Résultats**

Les résultats présenté en détail sont ceux en utilisant le mode de priorité SPT pour trouver la solution initiale à l'aide du gluttonous solver. Les résultats moyens des autres modes seront dans un tableau.

**Solution rapide (avec comme paramètres : maxIteration=10 et dureeTaboo=5)**

instance size best runtime makespan ecart

ta61 50x20 2868 33 3609 25,8

ta62 50x20 2869 39 3645 27,0

ta63 50x20 2755 47 3286 19,3

ta64 50x20 2702 47 3392 25,5

ta65 50x20 2725 42 3345 22,8

ta66 50x20 2845 40 3538 24,4

ta67 50x20 2825 45 3358 18,9

ta68 50x20 2784 60 3438 23,5

ta69 50x20 3071 31 3820 24,4

ta70 50x20 2995 44 3708 23,8

la01 10x5 666 1 666 0,0

la02 10x5 655 1 685 4,6

la03 10x5 597 1 651 9,0

la04 10x5 590 2 681 15,4

la05 10x5 593 3 593 0,0

la06 15x5 926 1 963 4,0

la07 15x5 890 4 1011 13,6

la08 15x5 863 1 916 6,1

la09 15x5 951 0 951 0,0

la10 15x5 958 1 1041 8,7

la11 20x5 1222 1 1390 13,7

la12 20x5 1039 1 1039 0,0

la13 20x5 1150 1 1329 15,6

la14 20x5 1292 1 1374 6,3

la15 20x5 1207 1 1330 10,2

la16 10x10 945 2 1097 16,1

la17 10x10 784 2 899 14,7

la18 10x10 848 1 981 15,7

la19 10x10 842 1 909 8,0

la20 10x10 902 2 950 5,3

la21 15x10 1046 2 1214 16,1

la22 15x10 927 3 1165 25,7

la23 15x10 1032 3 1119 8,4

la24 15x10 935 3 1136 21,5

la25 15x10 977 2 1340 37,2

la26 20x10 1218 4 1519 24,7

la27 20x10 1235 3 1685 36,4

la28 20x10 1216 6 1504 23,7

la29 20x10 1152 3 1496 29,9

la30 20x10 1355 4 1665 22,9

la31 30x10 1784 4 1891 6,0

la32 30x10 1850 5 2153 16,4

la33 30x10 1719 5 1834 6,7

la34 30x10 1721 5 1951 13,4

la35 30x10 1888 6 2099 11,2

la36 15x15 1268 6 1694 33,6

la37 15x15 1397 4 1600 14,5

la38 15x15 1196 6 1395 16,6

la39 15x15 1233 12 1518 23,1

la40 15x15 1222 6 1377 12,7

AVG - - 11,0 - 16,1

**Solution de qualité (avec comme paramètres : maxIteration=100 et dureeTaboo=100)**

instance size best runtime makespan ecart

ta61 50x20 2868 337 3527 23,0

ta62 50x20 2869 314 3630 26,5

ta63 50x20 2755 376 3286 19,3

ta64 50x20 2702 329 3302 22,2

ta65 50x20 2725 314 3291 20,8

ta66 50x20 2845 294 3502 23,1

ta67 50x20 2825 334 3285 16,3

ta68 50x20 2784 406 3398 22,1

ta69 50x20 3071 362 3801 23,8

ta70 50x20 2995 297 3655 22,0

la01 10x5 666 2 666 0,0

la02 10x5 655 4 680 3,8

la03 10x5 597 2 638 6,9

la04 10x5 590 4 602 2,0

la05 10x5 593 1 593 0,0

la06 15x5 926 1 936 1,1

la07 15x5 890 3 927 4,2

la08 15x5 863 2 916 6,1

la09 15x5 951 5 951 0,0

la10 15x5 958 2 1034 7,9

la11 20x5 1222 6 1320 8,0

la12 20x5 1039 4 1039 0,0

la13 20x5 1150 5 1173 2,0

la14 20x5 1292 3 1374 6,3

la15 20x5 1207 8 1299 7,6

la16 10x10 945 18 1006 6,5

la17 10x10 784 15 830 5,9

la18 10x10 848 11 942 11,1

la19 10x10 842 11 875 3,9

la20 10x10 902 12 938 4,0

la21 15x10 1046 20 1175 12,3

la22 15x10 927 18 1049 13,2

la23 15x10 1032 15 1100 6,6

la24 15x10 935 19 1112 18,9

la25 15x10 977 15 1286 31,6

la26 20x10 1218 22 1477 21,3

la27 20x10 1235 22 1414 14,5

la28 20x10 1216 30 1434 17,9

la29 20x10 1152 25 1461 26,8

la30 20x10 1355 22 1497 10,5

la31 30x10 1784 39 1868 4,7

la32 30x10 1850 34 2054 11,0

la33 30x10 1719 26 1782 3,7

la34 30x10 1721 31 1881 9,3

la35 30x10 1888 20 2059 9,1

la36 15x15 1268 37 1675 32,1

la37 15x15 1397 40 1579 13,0

la38 15x15 1196 39 1379 15,3

la39 15x15 1233 54 1500 21,7

la40 15x15 1222 43 1313 7,4

AVG - - 81,1 - 12,1

**Autres modes de priorités**

Les résultats moyens des autres modes de priorité sont récapitulés dans le tableau suivant.

|  |  |  |
| --- | --- | --- |
|  | Runtime | Makespan |
| taboo\_fast\_spt | 11.0ms | 16.1% |
| taboo\_fast\_lpt | 11.5m | 27.8% |
| taboo\_fast\_srpt | 16.1ms | 23.3% |
| taboo\_fast\_lrpt | 9.5ms | 13.5% |
| taboo\_quality\_spt | 81.1ms | 12.1% |
| taboo\_quality\_lpt | 99.5ms | 20.0% |
| taboo\_quality\_srpt | 96.5ms | 18.7% |
| taboo\_quality\_lrpt | 69.5ms | 9.5% |

## Comparaison des différentes méthodes

On trouve dans le tableau suivant le récapitulatif des différentes méthodes utilisées pour les instances allant de la01 à la40 et de ta61 à ta70.

|  |  |  |
| --- | --- | --- |
|  | Runtime | Makespan |
| gluttonous\_spt | 0.4ms | 22.1% |
| gluttonous\_lpt | 0.6ms | 34.9% |
| gluttonous\_srpt | 0.5ms | 29.1% |
| gluttonous\_lrpt | 0.4ms | 19.1% |
| descent\_spt | 5.6ms | 17.1% |
| descent\_lpt | 11.7ms | 29.4% |
| descent\_srpt | 11.8ms | 25.0% |
| descent\_lrpt | 11.8ms | 15.0% |
| taboo\_fast\_spt | 11.0ms | 16.1% |
| taboo\_fast\_lpt | 11.5m | 27.8% |
| taboo\_fast\_srpt | 16.1ms | 23.3% |
| taboo\_fast\_lrpt | 9.5ms | 13.5% |
| taboo\_quality\_spt | 81.1ms | 12.1% |
| taboo\_quality\_lpt | 99.5ms | 20.0% |
| taboo\_quality\_srpt | 96.5ms | 18.7% |
| taboo\_quality\_lrpt | 69.5ms | 9.5% |

On voit que plus on descend dans le tableau, plus le makespan est bon (car on utilise des méthodes de méta-heuristique plus élaborés), et plus le temps de calcul augmente (pour la même raison, à l'exception du taboo solver paramétré sur fast qui met un runtime similaire au descent solver alors qu'il a un meilleur makespan).

Si le problème à résoudre est du même ordre de grandeur que les instances vu dans ce projet et que l'on veut la meilleure qualité possible, on peut utiliser le taboo\_quality\_lrpt, ou un taboo\_lrt avec encore plus d'itérations. Si le problème est vraiment petit et que l'on a suffisamment de puissance de calcul et de temps on peut utiliser la méthode exhaustive et ainsi avoir la solution optimale. Le gluttonous\_lrpt est à utiliser lorsque l'on veut rapidement une solution pas trop mauvaise, et cela peut être le cas même pour des très petites instances si par exemple il y a besoin de résoudre une grande quantité d'instances par seconde. Le paramètre LRPT à l'air d'être le meilleur choix pour le mode de priorité.

## Conclusion

Quand on a un problème à résoudre, au lieu de le résoudre particulièrement, on peut voir si il appartient à une classe de problème qui peut déjà être résolue avec des méthodes génériques. Le mot résolue veut dire que l'algorithme retournera une solution possible, ou une solution optimale, ou une solution assez bonne, suivant le type de problème. Cela va permettre d'avoir les grandes lignes de l'algorithme à utiliser, et il restera à utiliser les réprésentations propre au problème actuel pour l'implémenter.

Les méthodes de méta-heuristiques sont très générales (elle s'appliquent à beaucoup de problèmes, même s'il y a une partie de l'algo qui différe pour chaque problème), contrairement à une méthode de graphes par exemple qui peut servir à résoudre de nombreux problème mais seulement des problèmes de graphes. Cela à l'avantage de pouvoir passer du temps à améliorer les méthodes de résolution méta-heuristique car toutes les applications concrètes en bénéficieront. Il y a par contre un élément qui lui dépend fortement du problème considéré : l'heuristique (sauf dans le cas des méthodes exhaustives). Il va falloir coder particulièrement cette partie de l'algorithme, même s'il peut aussi y avoir des heuristiques déjà connu pour bien fonctionner pour des classes de problèmes. Cette façon de faire présente l'avantage de bénéficier à la fois de l'investissement mis dans la partie générique de l'algorithme, et à la fois de la qualité de l'heuristique particulière pour le problème actuel à résoudre qui permet de donner à l'algorithme général de précieuses informations pour le résoudre efficacement.