Chapitre 5 : Métaheuristiques pour le problème de flow-shop à deux machines avec des temps d'exécution et de latence quelconques

5.1 Introduction

Nous présentons dans ce chapitre deux métaheuristiques, l'algorithme de recherche avec tabous (*RT*) et un algorithme génétique (*AG*), pour la résolution du problème de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Une comparaison entre les deux méthodes est ensuite entreprise.

5.2 Algorithme de recherche avec tabous

L'algorithme que nous avons conçu est comme suit. Nous avons tout d'abord généré d'une manière aléatoire une solution initiale. En fait, il s'agit de créer une séquence aléatoire d'entrée sur la première machine : c'est notre pré-solution. Ensuite, dans une seconde étape, on génère le voisinage de cette pré-solution initiale. La taille du voisinage est égale au nombre de jobs traités. Notons que ce voisinage est généré à partir d'un ensemble de permutations sur cette pré-solution initiale. Les permutations se font entre deux jobs choisis aléatoirement. Chaque couple de jobs choisis est sauvegardé dans une *liste taboue*. La taille de cette liste est égale à 7. Le choix de cette valeur est justifié par un ensemble de tests que nous avons effectué. De plus, cette *liste taboue* est gérée comme une liste "circulaire". Après avoir généré le voisinage, on entame l'étape suivante, celle du calcul du *makespan* en prenant en compte des temps de latence. Les tests nous ont conduits aussi à faire itérer ce processus 600 fois. Notons que le passage de 600 à 700 itérations n'a pas amélioré les résultats, alors que celui de 500 à 600 les a nettement améliorés sans pour autant consommer beaucoup plus de temps d'exécution. Le critère d'aspiration consiste à vider le tiers de notre *liste taboue* lorsque cette dernière est remplie deux fois de suite.

L'algorithme de *RT* a été codé dans le langage C++ sous l'environnement Visual Studio .NET 2003. L'algorithme a été exécuté sur une machine Pentium IV à 3.06 GHZ d'horloge et avec 512 Mo de mémoire vive.

Le Tableau 5-1 présente les résultats trouvés pour l'ensemble de toutes les instances : 10 instances pour chaque classe, soit 70 instances au total. La deuxième colonne représente le temps moyen d'exécution. Le *makespan* moyen, le meilleur et le pire sont indiqués respectivement à la troisième, quatrième et cinquième colonne.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,3	545,1	418	725
15	11,5	791,4	703	899
20	23,7	1043,1	955	1116
30	74,3	1576,0	1397	1726
40	157,3	2085,1	1910	2202
50	295,2	2584,4	2397	2876
60	495,9	3052,4	2820	3384

Tableau 5-1 : Résultats générés par RT simple

5.3 Amélioration de RT

Nous introduisons dans ce qui suit des modifications dans le processus de *RT* afin d'améliorer les résultats trouvés.

5.3.1. Solution initiale

La première amélioration que nous pouvons effectuer est de démarrer *RT* à partir d'une bonne solution, au lieu de commencer par une solution générée aléatoirement. Nous générons cette solution à l'aide de l'heuristique *NEH* décrite au Chapitre 6. Le Tableau 5-2 présente les résultats appliqués aux mêmes instances précédentes avec une solution initiale générée par *NEH*.

Certes il y a une augmentation de la consommation temporelle qui varie de 1,4 % à 7,2 %, mais nous avons amélioré les résultats moyens de 1 %. De même, les meilleures et pires solutions ont été améliorées respectivement de 1,21 % et 1,18 % par rapport à *RT* simple pour les séquences à 60 jobs.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,4	546,4	422	725
15	11,8	789,3	705	890
20	24,3	1039,6	956	1112
30	75,4	1566,4	1380	1723
40	165,2	2077,7	1911	2202
50	307,2	2565,9	2378	2822
60	531,9	3032,0	2786	3344

Tableau 5-2 : Résultats générés par RT avec solution initiale

5.3.2. Diversification avec restart

Un restart consiste à effectuer, lors de la recherche, un démarrage à partir d'une nouvelle solution. Il sera ainsi possible d'explorer un autre espace du domaine de solutions. Cette technique représente l'une des techniques utilisées dans la littérature pour appliquer la diversification dans l'algorithme de recherche avec tabous.

- **Utilisation de NEH**

La deuxième amélioration que nous pouvons effectuer est d'introduire de la diversification et ce, en calculant à l'aide de l'algorithme de *NEH* la séquence associée à la meilleure trouvée dans le voisinage. Cette opération s'effectue à chaque 50 itérations. Ainsi, *RT* effectue un nouveau départ en commençant la recherche à partir d'une nouvelle séquence. Le Tableau 5-3 présente les résultats appliqués aux mêmes instances en utilisant la diversification avec *NEH*.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	4,6	545,7	417	725
15	12,6	788,6	706	890
20	27,0	1040,2	956	1111
30	86,7	1565,7	1373	1723
40	194,1	2075,1	1911	2202
50	374,6	2563,9	2378	2822
60	669,2	3030,5	2786	3344

Tableau 5-3 : Résultats générés par RT avec restart NEH

Cette amélioration a augmenté les temps d'exécution de 5% pour les petites instances à 15 % pour les plus grandes. Le *makespan* moyen a été amélioré pour tous les

ensembles d'instances, excepté ceux de 20 jobs. Nous avons pu améliorer les meilleures solutions trouvées pour les séquences de 10, 15 et 30. Le pire *makespan* des instances de 30 jobs a été lui aussi amélioré. Pour le reste, nous avons trouvé les mêmes valeurs.

- **Utilisation de NEH et Priority**

La troisième amélioration consiste à introduire de la diversification et ce, en calculant à l'aide de *NEH* ou *Priority* la séquence associée à la meilleure trouvée dans le voisinage. Cette opération s'effectue à chaque 50 itérations. Le choix de *NEH* ou *Prioriy* se fait de manière aléatoire en favorisant *NEH* qui consomme moins de temps. Nous utilisons ce qu'on appelle la technique de roulette pour effectuer ce choix [Dorigo, 1995]. Ainsi, *RT* effectue un nouveau départ en commençant la recherche à partir d'une nouvelle séquence. Le Tableau 5-4 présente les résultats appliqués aux mêmes instances que ci-dessus, en utilisant la diversification avec *NEH* et *Priority*.

Nombre de jobs N	***Temps moyen***	***Makespan moyen***	***Meilleur makespan***	***Pire makespan***
10	4,9	544,0	417	724
15	12,2	788,0	706	890
20	25,7	1039,1	955	1110
30	82,9	1566,0	1375	1723
40	182,5	2077,0	1909	2202
50	341,8	2563,4	2378	2822
60	583,7	3030,7	2786	3344

Tableau 5-4 : Résultats générés par RT avec restart NEH et Priority

Nous pouvons remarquer d'après ce même tableau que les temps moyens d'exécution se sont améliorés de 4 % pour les instances de petites tailles et de 12% pour les instances de 60 jobs. Le *makespan* moyen a été amélioré pour les instances de 10, 15, 20 et 50 jobs. Pour le reste le *makespan* moyen n' pas changé. Le meilleur *makespan* a été amélioré pour les séquences de 20 et 40 jobs. Les pires *makespan* ont été amélioré pour les instances de 10 et 20 jobs.

5.3.3. Intensification

Une intensification est une procédure qui consiste à intensifier la recherche dans une zone bien déterminée de l'espace des solutions. Nous utilisons pour cela la recherche locale (*RL*).

- **Utilisation de RL**

L'intensification consiste, dans notre cas, à appliquer toutes les 50 itérations une *RL* sur la meilleure solution trouvée dans le voisinage. *RL* consiste à trouver la meilleure séquence, suite à des permutations améliorantes. Ainsi, une solution est gardée si elle ne détériore pas la précédente. Les positions choisies pour les permutations sont tirées aléatoirement. *RL* s'arrête après avoir effectué 100 permutations. Le Tableau 5-5 présente les résultats appliqués aux mêmes instances en utilisant *RL*. Notons que dans ce cas, nous n'avons pas utilisé d'intensification.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	5,5	546,4	422	725
15	13,7	790,1	706	891
20	26,9	1040,8	956	1114
30	81,2	1566,1	1373	1723
40	174,4	2076,8	1911	2202
50	320,3	2565,3	2378	2822
60	534,0	3032,8	2791	3344

Tableau 5-5 : Résultats générés par RT avec RL

Comparé au Tableau 5-4, les temps d'exécution se sont améliorés pour les grandes instances, avec une diminution de 8 % de ceux-ci. Le *makespan* moyen a été détérioré de 0,07 % pour les instances de 50 et 60 jobs et moins de 0,5 % pour le reste. Les pires *makespan* ont été aussi détériorés pour les instances de 10, 15, 20 et 30 jobs, mais sont égaux à ceux de 40, 50 et 60 jobs. Le meilleur *makespan* des séquences de 30 jobs a été amélioré, ceux de 15 et 50 sont restés inchangés, alors que le reste, ils ont été détériorés.

- **Utilisation de RL et de la diversification avec NEH et Priority**

La dernière amélioration que nous avons effectuée consiste à utiliser à la fois la diversification et l'intensification. La diversification consiste dans ce cas à utiliser la

méthode de *restart* avec *NEH* et *Priority*. Après plusieurs tests, nous avons décidé de lancer la *RL* serait lancé toutes les 100 itérations et la diversification toutes les 30 itérations. Le Tableau 5-6 présente les résultats associés à cette dernière version de *RT*.

Nombre de jobs N	*Temps moyen*	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	5,9	543,9	417	724
15	14,6	787,8	706	890
20	29,2	1038,9	955	1109
30	86,4	1565,0	1370	1723
40	184,2	2076,7	1908	2202
50	337,0	2565,1	2378	2822
60	600,3	3030,5	2786	3344

Tableau 5-6 : Résultats générés par RT avec intensification et diversification

Comparée aux résultats du Tableau 5-5, cette version de *RT* consomme plus de temps (5% pour les petites instances et 10 % pour les plus grandes). Cependant, la valeur de tous les *makespan* moyens s'est améliorée. Il en est de même pour les pires valeurs du makespan où on les améliore trois fois et de cinq fois pour les meilleures valeurs. Pour les autres cas, on égale les précédents résultats.

5.4 Algorithme génétique

Dans cette section, nous décrivons, dans un premier temps, le contexte expérimental de l'algorithme génétique implémenté avec la plateforme *ParadisEO* [Cahon *et al.*, 2004]. Ensuite, nous détaillons l'implémentation de cet algorithme. Nous terminons cette section par la présentation des résultats obtenus de la simulation, que nous avons effectuée sur cet algorithme, et par la comparaison de ces derniers à ceux de l'algorithme avec tabous.

5.4.1. Contexte expérimental : ParadisEO

Nous avons utilisé la plateforme *ParadisEO* [Cahon et *al.*, 2004] pour implémenter l'algorithme génétique que nous avons conçu pour la résolution du problème de *flow-shop* à deux machines et avec des temps de latence. *ParadisEO* est un cadre de travail mettant en œuvre des librairies ANSI-C++ pour la conception et l'élaboration de métaheuristiques parallèles et hybrides dédiées à la résolution mono et multi-objectifs des

problèmes d'optimisation combinatoires. Ce cadre de travail est formé de quatre librairies:

- *Evolving Object* (*EO*) : Cette librairie est à la base de tout le projet. Elle fournit des outils pour élaborer des métaheuristiques basées sur des populations (Algorithme génétique, Programmation génétique, etc.).
- *Parallel Evolving Object* (*PEO*) : Cette librairie fournit des outils pour mettre ne œuvre des métaheuristiques parallèles et distribuées. Elle a été créée à la base pour assurer la parallélisation de la librairie EO de manière transparente et robuste. Nous pouvons trouver notamment le modèle en île, le modèle à parallélisation d'évaluation, le modèle cellulaire, etc.
- *Move Object* (*MO*) : La librairie MO permet d'implémenter des métaheuristiques à base de solution telles que la recherche avec tabous (*RT*), le recuit simulé (*RS*) ou encore le *hill climbing* (*HC*).
- *Multi Objectif Evolving Object* (*MOEO*) : Cette librairie permet d'implanter des métaheuristiques multi-objectifs avec plusieurs modèles tels que MOGA, NSGA-II, SPEA2, IBEA, etc. Elle permet aussi d'effectuer des calculs de performances des métaheuristiques.

Pour les besoins de notre travail, nous n'avons utilisé que la bibliothèque *EO*. Pour cela nous avons implémenté notre algorithme sur la même machine (512 Mo de RAM et 3,6 GHZ) mais sous un environnement *SuSe 10.2* avec *gc++ 4.6*. Nous l'avons exécuté sur le même ensemble des 70 instances générées précédemment pour l'algorithme de recherche avec tabous.

5.4.2. Implémentation de l'algorithme génétique

Les individus sont représentés par l'ensemble de jobs formant une séquence. Ainsi un individu représente une séquence d'exécution des différents jobs et la taille d'un individu est égale aux nombre de jobs dans une séquence. Après avoir effectué plusieurs tests, la taille de la population, la plus appropriées, est fixée à 40. Les individus seront générés aléatoirement. Le calcul du *makespan* de chaque individu représente notre évaluation. La sélection se fait d'une manière aléatoire. Nous avons gardé aléatoirement 40 individus entre les parents et les enfants générés. Le remplacement se fait d'une

manière élitiste. A chaque génération, il est produit 20 enfants issus des 40 parents. Les tests ont justifiés l'utilisation de l'opérateur de croisement COX au dépend de OX, PMX ou encore LX [Mühlenbein, 1993]. La probabilité du croisement est de 100%. Les 20 enfants générés sont issus de ce croisement. Nous avons aussi utilisé une *RL* comme opérateur de mutation. Pour un enfant sur quatre générés, nous avons appliqué une *RL* qui effectue un ensemble de permutations amélioratives. Cette recherche locale s'arrête après avoir effectué $N \times 20$ permutations où N est le nombre de jobs traités. Pour pouvoir comparer la recherche avec tabous et l'algorithme génétique, le critère d'arrêt de ce dernier est le temps consommé par la recherche avec tabous pour la même instance.

5.4.3. Comparaison des algorithmes RT et AG

Le Tableau 5-7 illustre les résultats de notre AG sur les 70 instances de données générées précédemment. La deuxième colonne représente le *makespan* moyen des 10 instances de chaque classe. La troisième et quatrième colonne représente, respectivement, le meilleur et le pire *makespan* trouvés.

N	*Makespan moyen*	*Meilleur makespan*	*Pire makespan*
10	544,7	423	724
15	788,6	706	884
20	1041,1	965	1113
30	1565,5	1373	1721
40	2076,9	1912	2214
50	2565,5	2396	2826
60	3031,3	2786	3344

Tableau 5-7 : Résultats AG

Au vue de ces résultats, nous pouvons tirer les conclusions suivantes : *RT* est meilleur qu'AG du point de vue de la valeur moyenne du *makespan*. Il en est de même 5 fois sur 7 concernant le pire *makespan*. Les résultats de *AG* sont comparés à la dernière version de *RT* avec de la diversification utilisant *NEH* et *Priority* et de l'intensification utilisant *RL*. Nous pouvons donc conclure que *RT* domine *AG* pour cet ensemble de données.

La Figure 5-1 présente les temps moyen d'exécution des quatre bornes supérieures ainsi que de l'algorithme de recherche avec tabous. Si les temps d'exécution

de *RT* et de *UB4* explosent avec l'accroissement du nombre de jobs, ceux de *UB1*, *UB2* et *UB3* restent relativement constants. En effet, *UB1* ne dépasse pas le tiers de la seconde, *UB2* ne dépasse pas le dixième de la seconde, et quant à *UB3*, elle ne dépasse pas 25 secondes en moyenne.

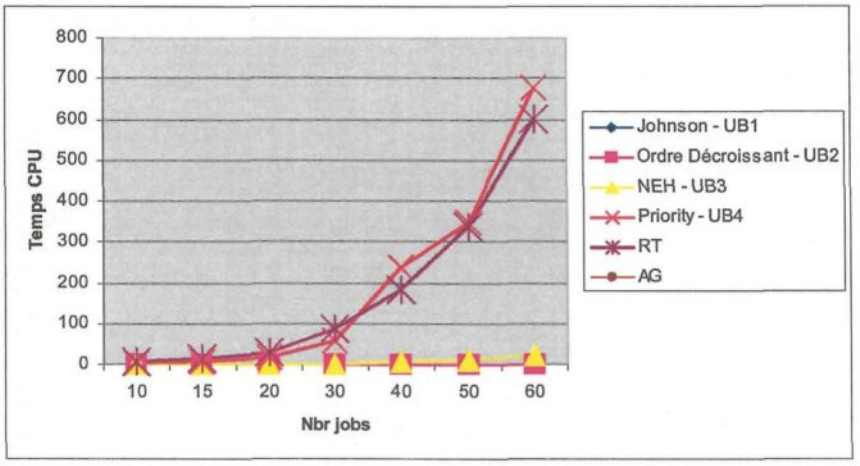

Figure 5-1: Comparaison des temps d'exécution RT et UB

La Figure 5-2 présente les déviations moyennes des quatre bornes supérieures ainsi que *RT*. La première remarque que nous pouvons faire est que la déviation moyenne diminue avec l'accroissement du nombre de jobs des séquences. Nous pouvons aussi distinguer deux groupes formés respectivement par *UB1* et *UB2* d'un côté et de *UB3*, *UB4* et *RT* d'un autre. Notons que le deuxième groupe donne de meilleurs résultats et que *UB3*, *UB4* et *RT* se confondent aux alentours de 2,5 % de déviation par rapport à *LB1* pour les séquences à 60 jobs. Nous pouvons aussi remarquer que *RT* domine les quatre bornes en termes de déviation.

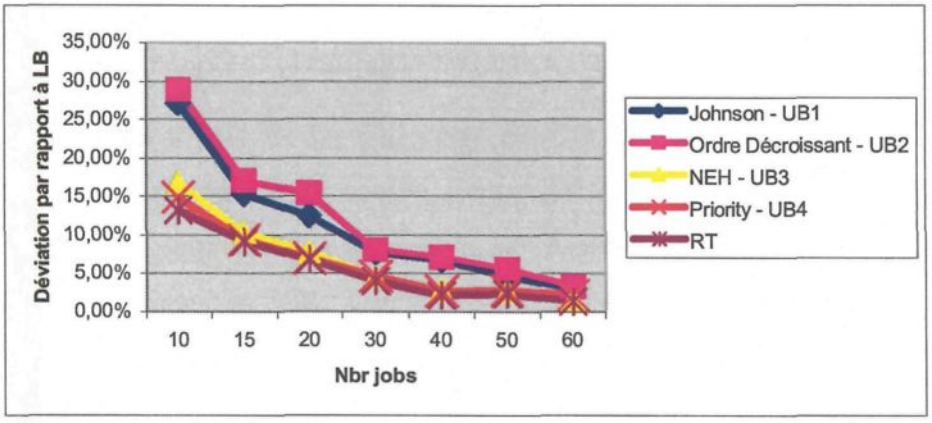

Figure 5-2: Comparaison des déviations moyennes par rapport à LB1

Conclusion

Nous avons étudié dans ce mémoire le problème de *flow-shop* à deux machines avec des temps de latence. Dans un premier temps, nous avons traité le problème avec des temps d'exécution unitaires avant de nous pencher sur les temps d'exécution quelconques. Les problèmes de type *flow-shop* ont été traités depuis la fin des années cinquante, notamment avec les travaux de [Johnson, 1954]. Après les vagues du *fordisme* et du *taylorisme*, les chaînes de production de type « ateliers » ont commencé à apparaître progressivement pour représenter de nos jours plus de la moitié des systèmes industriels des entreprises manufacturières [Pinedo, 2002]. Plusieurs variantes de ce modèle ont été utilisées dans l'industrie et dans les études théoriques. La pratique a cependant montré que les temps de latence liés aux jobs doivent être pris en considération étant donnée leur importance dans l'industrie d'aujourd'hui. En effet, Panwalkar et *al.* [1973] ont reporté que 19 % des activités industrielles subissent des temps de réglage et de transport liés aux jobs.

Après un chapitre introductif, nous avons présenté dans le Chapitre 2 les problèmes d'ordonnancement et décrit les différents modèles existant dans la littérature. Nous avons aussi motivé notre étude à l'aide d'un exemple de *flow-shop* à deux machines prenant en compte les temps de latence. Le critère que nous avons considéré dans notre travail est celui du *makespan*. Après y avoir introduit brièvement la *NP*-complétude, nous y avons survolé, à la fin de ce chapitre, les principales méthodes de résolution des problèmes d'ordonnancement, à savoir i) essayer de trouver un algorithme pseudo-polynomial ou de démontrer qu'il est *NP*-difficile au sens fort, *ii*) utiliser une méthode exacte, *iii*) utiliser une méthode approximative ou *iv*) utiliser une relaxation. Dans un deuxième temps, nous avons détaillé deux méthodes de résolution: l'approche exacte et approximative. Concernant la première approche, nous nous sommes concentrés sur la méthode de *branch and bound*. Cette méthode se base sur le calcul de bornes supérieures et inférieures à certains nœuds de l'arbre de recherche créé, afin d'élaguer certaines de ses branches au plus tôt et accélérer ainsi l'exécution de l'algorithme. Dans certains cas, il est aussi possible d'y introduire des règles de dominances pour mieux encore élaguer au plus

tôt certaines branches. Concernant l'approche approximative, nous avons abordé les heuristiques constructives, amélioratives ainsi que les métaheuristiques. Contrairement aux méthodes exactes, et pour des problèmes *NP*-difficiles, les heuristiques constituent une approche appropriée pour trouver de « bonnes » solutions en un temps « raisonnable ». Parmi les métaheuristiques, que nous avons discutées, nous avons détaillé l'algorithme de recherche avec tabous et les algorithmes génétiques.

Dans le Chapitre 3, nous avons décrit le problème du *flow-shop* et donné quelques propriétés pour le cas à deux machines avec des temps de latence. Nous avons fait également une revue de littérature de la résolution du problème du *flow-shop*. Cette revue de la littérature a concerné les méthodes heuristiques et la méthode de *branch and bound* pour la résolution d'un problème de *flow-shop*. Nous y avons ainsi décrit certaines heuristiques dont celles que nous avons utilisées, à savoir la méthodes de Johnson [1954], CDS [Campell et *al.*, 1970], Palmer [1965] et *NEH* [Nawaz et *al.*, 1983]. La revue de la littérature concernant la résolution du problème du *flow-shop* avec la méthode de *branch and bound* a, quant à elle, pu nous donner une idée sur les différentes bornes supérieures et inférieures utilisées ainsi que les règles de dominance.

Pour la résolution du problème de minimisation du *makespan* d'un problème de *flow-shop* à deux machines avec des temps de latence à l'aide de la méthode de *branch and bound*, nous avons considéré dans un premier temps, au Chapitre 4, le cas des temps d'exécution unitaires. Nous y avons présenté des bornes inférieures et supérieures ainsi que des règles de dominances pour l'implémentation d'un algorithme de *branch and bound*. Notons que cette partie est entièrement basée sur les travaux de [Moukrim et Rebaïne, 2005] et [Yu, 1996]. Dans un deuxième temps, nous sommes passés aux temps d'exécution quelconques. Dans ce cas, nous nous sommes servis de bornes inférieures de Yu [Yu, 1996], et avons conçu des bornes supérieures, qui sont des versions modifiées de l'algorithme de Johnson [1954], CDS [Campbell et *al.*, 1970], règles de priorité de Palmer [1965] et *NEH* [Nawaz et *al.*, 1983]. Nous avons tout d'abord évalué les heuristiques. Ensuite, nous avons appliqué l'algorithme de *branch and bound* en utilisant seulement les bornes inférieures avant d'y incorporer les bornes supérieures finalement. Les résultats se sont nettement améliorés du point de vue du nombre de nœuds visités et du meilleur *makespan* trouvé.

Le Chapitre 5 décrit la résolution du problème cité ci-dessus avec un algorithme de recherche avec tabous et un algorithme génétique. Concernant l'algorithme de recherche avec tabous, nous l'avons démarré avec une solution initiale générée aléatoirement, une liste taboue à 7 éléments et un voisinage de taille égal au nombre de jobs dans une séquence. Un voisinage contient des solutions issues de la solution courante après y avoir effectué une permutation de jobs. Une solution initiale générée avec *NEH* [Nawaz *et al.*, 1983], une diversification avec restart en utilisant *NEH* et la règle de priorité de Palmer [1965] et une intensification en utilisant de la recherche locale améliorative sont les différentes améliorations que nous avons ajoutées à l'algorithme de recherche avec tabous. Pour chaque amélioration, nous avons pu remarquer une amélioration des résultats. L'algorithme génétique a été implanté avec la plateforme *ParadisEO* [Cahon et *al.*, 2004] avec une population de 40 individus, un croisement COX, une mutation avec de la recherche locale améliorative, une sélection aléatoire et un remplacement élitiste. Le critère d'arrêt de l'algorithme génétique correspond au temps consommé par la recherche avec tabous pour la même instance. Les résultats ont montré que l'algorithme de recherche avec tabous domine clairement l'algorithme génétique.

Certes, l'algorithme de *branch and bound* a généré d'assez bons résultats, pour les instances de petites tailles, mais l'élaboration de meilleures bornes inférieures et supérieures pourraient améliorer les résultats. La conception de règles de dominances produira des répercussions positives sur la qualité les résultats. Concernant l'algorithme de recherche avec tabous, le passage vers un voisinage plus varié, voire multiple, pourrait améliorer les résultats. Une hybridation avec l'algorithme génétique, ou d'autres heuristiques, est aussi envisageable dans des travaux futurs de recherche.