ISIMA

OUTILS D'AIDE A LA DECISION

Job Shop

Mise en place d'un algorithme mémétique pour la résolution d'un problème type Job Shop.

BARBESANGE Benjamin – GARÇON Benoît

26/11/2015

Table des matières

Table des illustrations

INTRODUCTION

Ce projet s'inscrit dans le cursus de seconde année à l'ISIMA. Le but est d'implémenter la résolution d'un problème NP-difficile comme le Job Shop grâce à des métaheuristiques comme un algorithme mémétique.

Le problème du Job Shop est le suivant : nous avons un nombre m de produits à usiner par n procédés sur n machines distinctes. Ces n procédés pour chaque produit doivent être effectués dans un ordre très précis. Pour chaque produit cet ordre peut être différent. Chaque procédé est effectué en un temps $c_{i\ j}$ et ne peut être concomitant à un autre procédé sur la même machine.

L'objectif est donc de trouver un ordre de passage sur les machines permettant d'usiner chaque produit le plus rapidement possible en respectant les contraintes. On veut donc la date de fin au plus tôt du travail.

Le problème étant que le Job Shop n'est pas un simple problème. Il existe en effet $(n!)^m$ combinaisons d'ordre ce qui devient très vite ingérable informatiquement parlant. Ce problème est en effet un problème NP-difficile et ne peut se résoudre en un temps dit polynomial.

C'est pourquoi dans ce projet nous allons nous atteler à développer une heuristique permettant d'atteindre en un temps polynomial une valeur approchée de la valeur optimale.

I – ETUDE DU PROBLEME

Un Job Shop peut être résolu de manière naïve en générant tous les ordonnancements possibles et en cherchant dans ces résultats la valeur de makespan la plus petite. Mais ceci n'est pas réalisable.

Nous allons donc utiliser une méthode étudiée en cours : les (méta)-heuristiques. Ceci nous permettra d'obtenir un résultat proche de la valeur optimale voire, la valeur optimale elle-même.

A – Génération d'une solution

Avant tout nous allons définir ce qu'est une solution. Pour notre problème une solution sera un graphe disjonctif orienté avec les dates de début au plus tôt de chaque opération.

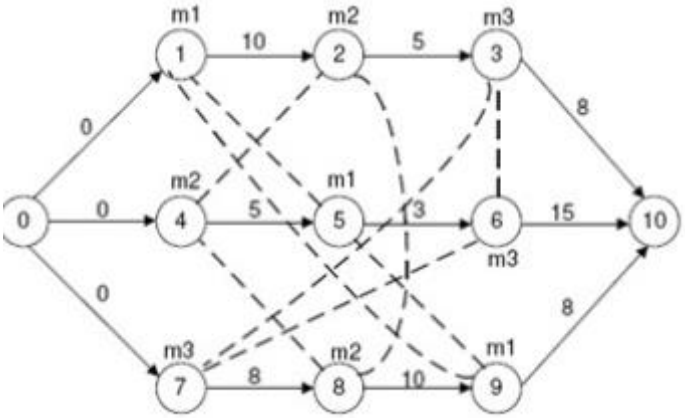

Figure 1 - Exemple de graphe disjonctif (P. Lacomme)

Ce graphe est assez lourd à représenter c'est pourquoi nous allons utiliser la méthode du vecteur de Bierwirth vu en cours. Ce vecteurs va être de taille n*m et contiendra m fois tous les entiers entre 1 et n. Le premier de chaque entier k correspond à la première tâche du job k, le second au second, etc.

Ainsi pour l'évaluation nous allons juste parcourir et évaluer les sommets dans l'ordre d'un vecteur de Bierwirth. Ceci assurera les contraintes pour l'ordre des tâches d'un même job et évitera d'avoir des cycles. L'évaluation sera enfantine : pour chaque sommet dans l'ordre de Bierwirth la date de début de la tâche sera égale au maximum entre la date de disponibilité de la machine et la date de fin de la dernière tâche sur le job.

On obtient ainsi une solution réalisable du problème.

B – Amélioration de cette solution

Maintenant que nous tenons une solution réalisable il convient de déterminer si elle peut être améliorée. En effet dans l'espace des solutions, nous allons pouvoir faire une recherche des minimas locaux. Cette recherche s'effectue en modifiant certains arcs disjonctifs de notre graphe. En effet les arcs disjonctifs ne peuvent être changés. Or on

a montré en cours que seules les modifications sur les arcs présents sur le chemin critique de la solution réalisable peuvent générer de meilleures solutions.

Dans notre vecteur de Bierwirth, un arc disjonctif correspond en fait à deux entiers différents consécutifs. La recherche locale va donc consister à rechercher sur ce chemin critique un échange qui donnera une évaluation meilleure et ce tant qu'on trouve mieux (ou alors tant qu'une limite n'est pas atteinte).

A la fin de cette recherche nous obtenons un minima local qui pourrait être la valeur optimale mais qui a très peu de chances de l'être, c'est pourquoi nous devons mettre en place un algorithme complémentaire.

C - Fabrication de meilleurs solutions

Maintenant que nous savons trouver les minimas locaux, nous allons pouvoir essayer de générer à partir de ceux-ci de meilleures solutions. En effet nous allons utiliser les solutions générées pour trouver des solutions voisines et ainsi pouvoir découvrir d'autres minimas locaux pour se rapprocher de la valeur optimale.

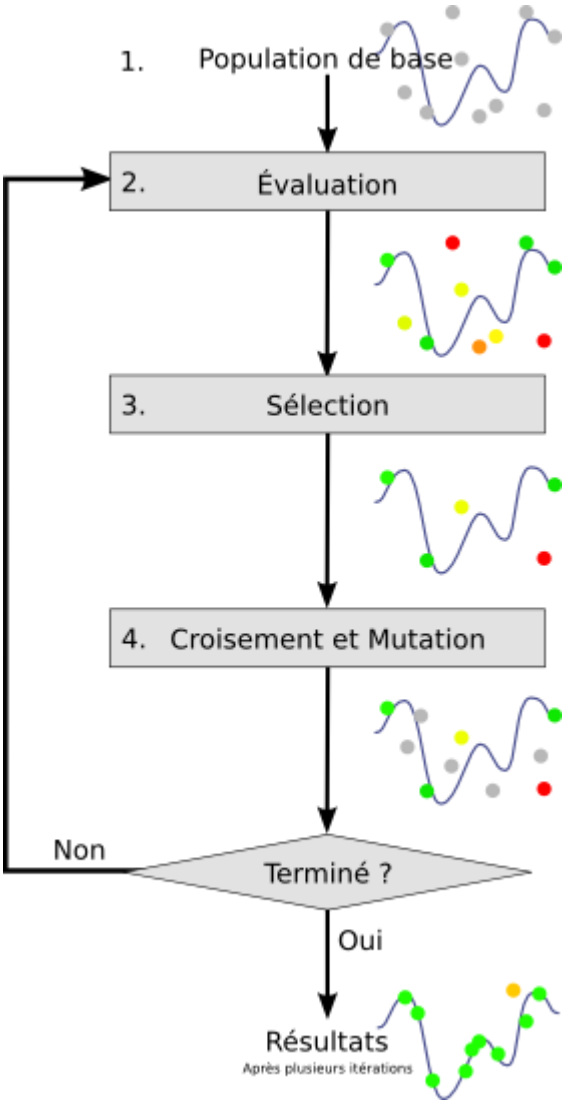

Figure 2 - Algorithme génétique

Pour constituer ces nouveaux « voisins » nous allons utiliser un algorithme dit génétique. Inspiré de la théorie de l'évolution, cet algorithme consiste à prendre des solutions parmi les meilleurs et à les croiser avec d'autres afin de créer de nouvelles solutions. On va évaluer toutes les solutions enfant et les ajouter à notre population de solutions. Ensuite nous ne garderons que les meilleures solutions et supprimerons les moins bonnes (comme dans le règne animal).

Ainsi en générant de nouvelles solutions et en ne gardant que les meilleures, nous sommes sûrs qu'à chaque itération notre meilleur makespan est au moins aussi bon que celui de l'itération précédente.

Comme nous pouvons voir sur le schéma (2), l'algorithme génétique est vraiment très naturel. La différence avec notre méthode est que pour chaque solution enfant générée, nous allons effectuer une recherche locale permettant d'obtenir des résultats accélérés : c'est un algorithme mémétique.

II – PRESENTATION DE LA SOLUTION

A – Evaluation du vecteur de Bierwirth

Présentation

Un vecteur de Bierwirth va permettre d'identifier l'ordre de passage de nos objets dans les machines. Ce vecteur se lit conjointement avec le graphe et fournit un ordre topologique. Il est beaucoup plus pratique de travailler avec ce genre de vecteur qu'avec la structure de graphe.

Algorithme

```
Pour chaque objet dans le vecteur Faire
      Recherche sur quelle machine il doit s'executer

      Si date de disponibilité du job < date de disponibilité machine
Alors
            La date de début du job est celle de disponibilité de la
machine
            Mise à jour de la date de disponibilité machine en
ajoutant la durée du job
            Mise à jour de la date de disponibilité du job
            Mise à jour du pointeur father
      Sinon
            La date de début du job est celle à laquelle il est
disponible
            Mise à jour de la date de disponibilité machine qui
correspond à la date de fin de ce job
            Mise à jour de la date de disponibilité du job
            Mise à jour du pointeur father
      Fin Si

      Mise à jour du pointeurs vers le précédent
      Mise à jour du dernier job de la machine
      Mise à jour de la prochaine machine pour ce job

Fin Pour

Rechercher et retourner le makespan
```

Implémentation

La méthode *Data::evaluer()* se trouve dans le fichier *Data.cpp*.

B – Recherche locale

Présentation

Cette méthode va permettre d'améliorer la solution que nous avons trouvée avec le vecteur de Bierwirth. Nous allons chercher à échanger un sommet du chemin critique avec un autre sommet du vecteur, dans le but de faire diminuer le makespan.

La structure de Job manipulée ici va contenir un pointeur father, qui va nous permettre de trouver le chemin critique une fois l'évaluation du vecteur effectuée. Il faudra simplement remonter à partir du dernier élément se terminant pour le constituer et améliorer la solution en effectuant des permutations.

Algorithme

```
Tant que le makespan est amélioré ET que l'on ne dépasse pas le nombre d'itérations Faire
	Tant que parcours du chemin critique
		Rechercher un arc disjonctif
		Le permuter avec un arc du chemin critique
		Evaluer le nouveau vecteur obtenu

		Si makespan est amélioré Alors
			Garder ce vecteur et cette solution
			Ne pas continuer sur le chemin critique
		Sinon
			Continuer sur le chemin critique
		Fin Si
	Fin Tant que
Fin Tant que

Retourner le makespan
```

Implémentation

La méthode *Data::rechercheLocale()* se trouve dans le fichier *Data.cpp* et fait appel à la méthode *Data::amélioration()*.

C - Algorithme de suppression des doublons

Présentation

Etant donné que notre vecteur de Bierwirth va être généré de manière aléatoire, il est possible que dans certains cas nous obtenions le même vecteur et donc la même solution. C'est un cas que nous devons alors éliminer.

Pour ce faire, nous allons utiliser une méthode basée sur un tableau de présence. Nous initialisons un tableau de très grande taille (de l'ordre du million). Lorsque nous souhaitons consulter si une solution a déjà été envisagée, nous utilisons une fonction de hashage sur la solution de la manière suivante : nous faisons la somme des carrés des dates de début de chaque job de la solution, modulo la taille du tableau de présence.

Implémentation

La suppression des doublons est directement implémentée dans le constructeur de la classe *Population* pour des raisons de rapidité.

D - Algorithme génétique

Présentation

Cet algorithme va permettre de s'approcher de la solution optimale du problème. On dispose d'une population triée par ordre croissant de makespan. On génère le même nombre d'individus que la population par croisement. On retrie la population puis on élimine les individus en trop dans la population pour conserver uniquement les meilleurs.

Algorithme

```
Tant que l'on ne dépasse pas le nombre d'itérations Faire
      Pour i de 1 à taille de la population Faire
            Choisir P1 dans les 10% de la population
            Choisir P2 dans les 90% restants
            Effectuer le croisement de P1 et P2 -> C
            Effectuer la recherche locale sur C
            Ajouter C à la population
      Fin Pour

      Retrier la population
      Retirer les individus en trop pour revenir au nombre initial

      Si on a pas amélioré la solution 10 fois consécutives Alors
            On regénère aléatoirement les 90% de la population qui est
"mauvaise"
      Fin Si
Fin Tant que

Retourner le makespan
```

Implémentation

L'algorithme génétique est implémenté dans le fichier *Data.cpp*. Par défaut, on utilise 100 itérations ainsi qu'une population de 100, si la méthode est appelée sans paramètres.

III – RESULTATS ET PERFORMANCES

A - Présentation du programme

Le programme présent sous le nom de prog peut prendre 1, 2 voire 3 arguments. Le premier doit être le nom du fichier contenant le graphe à traiter, par défaut ce fichier est "INSTANCES/la01.dat". Le second est le nombre d'itérations à effectuer pour l'algorithme mémétique, par défaut cette valeur est fixée à 100. Le dernier n'est autre que la taille de la population de solutions toujours pour l'algorithme mémétique, sa valeur par défaut est fixée à 100.

Ainsi un exemple d'utilisation du programme est :

./tp2 INSTANCES/la05.dat 400 100

Le programme n'affiche pas ici l'ordonnancement final obtenu, en effet nous préférons nous concentrer sur le makespan et la vitesse d'exécution plutôt que d'afficher un ordonnancement sans réel intérêt dans cette étude toutefois il suffirait d'un simple display_all() pour changer cela.

B - Tests et analyse

Nous avons effectués des tests sur les 19 premiers fichiers présents dans le répertoire INSTANCES. Chaque fichier est traité par l'algorithme génétique avec au maximum 1000 réplications et une population de base de 100 individus. Dans le tableau ci-dessous sont répertoriées les informations de tests.

Fichier3	Solution	Optimale	Ecart (%)	Itérations	Temps (s)
la1.dat	666	666	0%	1000	9.19
la2.dat	738	655	13%	1000	9.64
la3.dat	680	597	14%	1000	10.53
la4.dat	643	590	9%	1000	10.02
la5.dat	593	593	0%	1000	9.48
la6.dat	926	926	0%	1000	12.9
la7.dat	944	890	6%	1000	13.83
la8.dat	925	863	7%	1000	13.79
la9.dat	951	951	0%	1000	12.18
la10.dat	958	958	0%	1000	11.84
la11.dat	1222	1222	0%	1000	19.13
la12.dat	1059	1039	2%	1000	17.3
la13.dat	1150	1150	0%	1000	16.96
la14.dat	1292	1292	0%	1000	17.87
la15.dat	1363	1207	13%	1000	22.23
la16.dat	1059	945	12%	1000	12.67
la17.dat	853	784	9%	1000	11
la18.dat	902	848	6%	1000	11
la19.dat	943	842	12%	1000	12.38

Comme nous pouvons l'observer, en majorité, nous atteignons la solution optimale. De plus le temps de calcul semble assez raisonnable compte tenu qu'il été effectué sur nos machines virtuelles.

Dans le cas où nous n'atteignons pas la solution optimale, nous observons néanmoins que nous ne sommes pas éloignés de plus de 10% en moyenne de celle-ci, ce qui n'est pas une mauvaise chose.

Il est possible que dans certains cas, l'algorithme génétique ne suffisent plus à obtenir la solution optimale. Dans certains cas la valeur optimale n'est pas atteinte, en effet cela est dû au fait que la population n'est au fil des itérations pas assez diversifiées et la population est piégée dans un grand minimum local. Comme nous l'avons vu en cours, ce phénomène est tout à fait normal, la solution pour y remédier est aussi très simple. Lorsque la valeur optimale de makespan de la population n'augmente pas au bout d'un nombre r d'itérations il faut renouveler une partie celle-ci avec un peu d'aléatoire.

En effet on voit bien qu'au fil des itérations le makespan minimum n'augmente jamais car on garde toujours le meilleur. On ne peut donc avoir qu'au moins aussi bien que ce que l'on a à l'itération n-1.

Nb itérations	10	20	50	100	200
Makespan	742	742	737	685	666

Tableau 1 - Evolution de l'algorithme mémétique sur la01.dat en fonction du nombre d'itérations

Cette courbe est décroissante mais ne pourra jamais dépasser la valeur optimale bien sûr. Donc l'algorithme converge vers la valeur optimale.

CONCLUSION

La finalité de ce projet est qu'il est très complet. En effet, nous avons dû réfléchir à l'organisation de nos différents algorithmes pour qu'ils collaborent et atteignent au mieux la solution optimale ou en tout cas, s'en approcher au mieux.

Nous avons donc mis en place une solution trouvée parmi les métaheuristiques qui n'est autre qu'un algorithme génétique amélioré par des recherches locales. Cette méthode nous a permis de trouver dans bien des cas une valeur très approchée de la valeur optimale et même ladite valeur optimale pour d'autres problèmes. Les métaheuristiques sont donc une approche qui permet en un temps très raisonnable d'obtenir des résultats qui aurait mis des temps quasi infinis pour être calculés de façon exacte.

Concernant la rapidité de l'exécution nous sommes ici dans une échelle polynomiale ce qui offre une rapidité d'exécution infiniment plus élevée que la résolution naïve des problèmes NP. Les techniques algorithmiques introduites dans ce projet ont, elles aussi, permise l'accélération du processus de détermination du résultat à une échelle inférieure. En effet, la méthode du vecteur de Bierwirth permet de représenter un graphe complexe dans un simple vecteur et la table de hashage pour la reconnaissance des doublons permet de savoir en complexité O(1) si un graphe menant à une solution identique a déjà été testé.

En conclusion de ce projet, nous avons montré l'efficacité pratique de l'algorithme mémétique qui converge vers la solution optimale. Il faudrait alors se pencher sur des formes plus avancées du Job Shop pour tenter de reproduire cette méthode sur des contraintes supplémentaires par exemple.