TP2 - Recherche Operationnelle

Rapport - BELLEC Louison et BOUVARD Alexandre

Table of Contents

Descriptif

Le but du Job Shop est de trouver l'agencement optimisé des pièces et machines pour réduire les coûts.
On commence par générer un vecteur de Bierwirth qui a pour propriété de donner un ordre des pièces sur les machines, assuré sans cycles.

Evaluation du graphe

On calcule les dates de début d'un vecteur pour obtenir le coût final à partir d'un vecteur de Bierwirth. Cette fonction est importante car elle permet de facilement comparer deux vecteurs. On peut ainsi savoir lequel a le coût le plus faible.

Generation d'un graphe

On génère le graphe dans la partie père ou l'on trace nos arcs disjonctifs et horizontaux. Cela nous permet de remonter facilement le chemin critique pour ensuite tenter de l'améliorer.

Conception d'une recherche locale

La recherche locale à pour but de remonter le chemin critique en inversant les arcs disjonctifs pour tenter d'optimiser la solution. Elle permet très souvent d'améliorer le cout initial du vecteur généré par l'évaluation.
On inverse les arcs disjonctifs 2 à 2 et on vérifie si il y a une amélioration. On essaye ainsi sur tous les arcs afin d'obtenir la solution la plus optimisé a partir d'un veteur de base.

Conception d'un GRASP

Le GRASP à pour but d'optimiser le JobShop en générant un grand nombre de solutions. En effet, elle génère plusieurs voisins de la solutions actuelle aléatoirement. Puis elle effectue la recherche locale sur chaque voisin afin d'avoir la meilleure solution possible pour ce voisin. Enfin, elle stockent le meilleur des vecteurs au final.
Creer X voisins randomisé pour un vecteur permet de garder une diversité des solutions et donc d'améliorer plus le vecteur. La recherche locale seule n'est pas assez puissante car elle ne modifie que les arcs disjonctifs du chemin critique.

Remarques sur l'optimisation du JS

L'aléatoire joue une grande part dans la génération du vecteur de Bierwirth de base. Il est donc possible d'avoir une solution suboptimale des le début. La recherche locale permet d'améliorer cette solution de base. Le GRASP va génerer de nombreux voisins et donc réduire l'influence de l'aléatoire de la premiere solution.

Il est important de noter que de choisir une seed particulière et donc l'aléatoire de base pour la génération du vecteur ou la séquence des swaps sur le GRASP permet d'améliorer grandement (ou inversement) le résultat.

Description algorithmique

Nous allons décrire en francais de manière très abstraite le fonctionnement des differents algorithmes utilisés.
Les details d'implémentation peuvent être lu directement dans le code.

Evaluer

Evaluer à pour but de calculer le coût d'un vecteur de Bierwirth, de créer le chemin critique et de renvoyer la pièce et l'opération finale

```
POUR chaque élement du vecteur de Bierwirth:

    SI ce n'est pas la première opération:
        Le pere est l'operation précedente de la pièce

        SI la date de fin du pere est supérieure à la date de début du fils:
            On décale la date de début du fils à celle de la fin du père
            On met le père en tant que père dans l'opération critique

            SI la date de fin du fils est plus élevée que le cout:
                Le coût devient la date de fin du fils
                On sauvegarde la nouvelle pièce et opération de fin
            FSI

        FSI
    FSI

    // il y a déjà eu une opération sur cette machine
    SI on est pas le premier sur la machine:
        Le père est la pièce et l'opération qui était précedement sur cette machine

        SI la date de fin du père est supérieure à la date de début du fils:
            On décale la date de début du fils à celle de la fin du père
            On met le père en tant que père dans l'opération critique

            SI la date de fin du fils est plus élevé que le coût:
                Le coût devient la date de fin du fils
                On sauvegarde la pièce et l'opération de fin
            FSI

        FSI
    FSI

    On sauvegarde que cette pièce et cette opération sont passés sur la machine
FPOUR
On renvoi la pièce et l'opération de fin
```

Recherche Locale

Cette fonction evalue un vecteur et essaye ensuite de l'améliorer en effectuant
des permutations (swap) sur le chemin critique.

```
On évalue une première fois le vecteur
TANTQUE on est pas au premier élèment ou au nombre maximum d'iterations:
    On copie le vecteur de base dans un nouveau vecteur
```

```
    On recherche le pere et le fils pour la meilleure opération limitante

    // père est différent de fils donc c'est une pièce différente
    SI ce n'est pas un arc horizontal:
        On fait une permutation (swap) de père et fils sur le nouveau vecteur

        On evalue le nouveau vecteur

        SI le coût du nouveau vecteur est supérieur au coût du vecteur de base:
            On garde le nouveau vecteur
            On garde l'opération et la pièce limitante
        SINON
            On avance au père du vecteur de base
        FSI
    SINON
        On avance au père du vecteur de base
    FSI
FTANTQUE
```

GRASP

Le GRASP va évaluer un vecteur, générer plusieurs voisins, appliquer la recherche locale a chacun d'entre eux et garder le meilleur. Il va recommencer sur plusieurs itérations. Un vecteur est voisin d'un autre s'il sont les même à un swap près.

```
On applique la recherche locale à notre vecteur de base
POUR le nombre d'itérations:
    On génère un nombre donné de voisins
    POUR chaque voisin:
        On effectue une recherche locale
        On stocke le meilleur de ces voisins (au plus faible coût)
    FPOUR
    SI le meilleur vecteur local est meilleur que le meilleur vecteur global:
        Le meilleur vecteur local devient le global
    FSI
FPOUR
On retourne le meilleur vecteur global (au plus faible coût)
```

Hash Set

Lorsque on creer nos voisins, il faut être sur que on ne creer pas 2 fois le meme voisin. Pour ce faire, cette fonction genere un voisin qui n'existe pas deja en utilisant un Hash Set.

```
On prend deux pièces aléatoirement dans le vecteur de Bierwith
On s'assure qu'elles sont différentes
On échange ces deux pièces dans notre vecteur
On génère le hash a partir de ce nouveau vecteur

SI le hash n'est pas present dans le HashSet:
    On l'ajoute et on retourne le nouveau vecteur
SINON
    On recommence la procedure
FSI
```

Détails d'implementation de "génèrer hash":
Cette fonction transforme un tableau d'entier en une chaine de caractere utilisee comme hash.

```
string genererHash(int[] tableau)
    POUR chaque élement du tableau:
        On concatène sa valeur dans un string
    FPOUR
    On retourne le string (qui est le hash unique du tableau)
F
```

Etude

Pertinence de la recherche locale

Nous avons fait une étude sur les dix premiers LA pour voir la pertinence de la recherche locale.
Pour cela, nous évaluons 10 vecteurs d'un coté, et nous faisons 10 recherches locales de l'autre.
Nous avons également comparé le GRASP en faisant simplement un évaluer pour chaque voisin avec le GRASP qui utilise la recherche locale.

Voici nos résultats :

```
LA01
Eval : 1313 || Local Search : 957
GRASP w/o LS: 688 || GRASP LS: 666

LA02
Eval : 1197 || Local Search : 1197
GRASP w/o LS: 751 || GRASP LS: 717

LA03
Eval : 1075 || Local Search : 803
```

```
GRASP w/o LS: 695 || GRASP LS: 631

LA04
Eval : 1198 || Local Search : 838
GRASP w/o LS: 641 || GRASP LS: 619

LA05
Eval : 932 || Local Search : 811
GRASP w/o LS: 593 || GRASP LS: 593

LA06
Eval : 1626 || Local Search : 1151
GRASP w/o LS: 926 || GRASP LS: 926

LA07
Eval : 1498 || Local Search : 1158
GRASP w/o LS: 921 || GRASP LS: 919

LA08
Eval : 1400 || Local Search : 1072
GRASP w/o LS: 889 || GRASP LS: 869

LA09
Eval : 1649 || Local Search : 1402
GRASP w/o LS: 966 || GRASP LS: 951

LA010
Eval : 1416 || Local Search : 1285
GRASP w/o LS: 981 || GRASP LS: 958
```

On peut voir que la recherche local permet très souvent une bonne amélioration comparée à l'évaluation simple. Ensuite, le grasp fait de bonne choses sans recherche local mais on n'arrive jamais à la solution optimale.
Ainsi, on peut dire que la recherche locale est tout à fait pertinente car elle permet de grandement améliorer les solutions évaluées (elle donne quelques fois les mêmes résultats quand l'aléatoire du vecteur est très bon et améliore grandement le reste du temps). Elle est également pertinente car nécessaire à l'obtention des solutions optimales avec le GRASP.

Convergence de l'algorithme

Les séquences d'aléatoire jouent malgrés tout un rôle important dans la réussite de la solution. En fonction de la séquence d'aléatoire de départ on peut directement avoir de très bon ou très mauvais résultats. La recherche locale et en surtout le GRASP permettent d'avoir des résultats pertinents dans la majorité des cas.

Il serait approprié de faire tourner de nombreuses fois le GRASP avec différentes seed pour garder la meilleur à la fin.

Avec nos autres tests, nous pouvons conclure que:

- On constate qu'augmenter le nombre d'itération du GRASP n'est plus ou peu efficace après un certain point. Faire 500 ou 1000 iterations semble donner le même résultat.
  En revanche on observe de très nettes progrès sur les premières itération (10 à 100 par exemple).

- Faire varier le nombre de voisin peut avoir un très gros impact sur la réussite de l'algorithme. Un nombre trop faible de voisin semble être peu efficace tandis qu'un nombre trop élevé ne donne pas forcément non plus de bon résultats.
  Faire varier les voisins de 10 à 15 ou 20 semble donner les meilleurs résultats.

En utilisant la seed `1234` :

```
Avec 10 voisins, 10 itérations => 1013
Avec 10 voisins, 50 itérations => 967
Avec 10 voisins, 100 itérations => 967
Avec 10 voisins, 500 itérations => 967
Avec 10 voisins, 2000 itérations => 967

Avec 15 voisins, 10 itérations => 1068
Avec 15 voisins, 50 itérations => 921
Avec 15 voisins, 100 itérations => 907
Avec 15 voisins, 500 itérations => 907
Avec 15 voisins, 2000 itérations => 907
```

En utilisant la seed `1` :

```
Avec 10 voisins, 100 itérations => 1022
Avec 10 voisins, 2000 itérations => 1006

Avec 15 voisins, 100 itérations => 1022
Avec 15 voisins, 2000 itérations => 1006
```

Il semble donc important de faire varier ces differents paramètres : le nombre de voisins, la seed. Le nombre d'itération semble avoir une importance jusqu'à un certain point.

Le taux de convergence est assez fort au début puis à partir d'un certain nombre d'itérations il devient très difficile d'améliorer.

Si nous devions trouver la solution la plus optimisée pour un problème avec un temps de calcul plus long possible, nous garderions un nombre minimum d'itération (entre 400 et 500: permet d'avoir un bon résultat en un temps raisonnable). Et nous ferions varier les nombres de voisins (par exemple entre 10 et 20) sur pleins de seeds differentes. On peut faire tourner le programme pendant un long temps afin d'avoir de nombreuses solutions et donc d'obtenir la meilleure possible.