DENIZOT Hugo F2 17/10/2022

Compte Rendu TP2 Job Shop

Sommaire :

1.Explication du sujet :

Dans ce compte rendu nous allons voir comment trouver grâce à des algorithmes le cout minimum d'un job shop. Pour ceci nous allons passer par différentes procédures qui auront tous un rôle à jouer dans l'optimisation de ces algorithmes. Il y aura dans ce rapport les algorithmes de ces fonctions et dans un fichier à côté les fonctions codées.

2. Présentation des fonctions nécessaires à l'optimisation

a) Génération d'un graphe à partir d'une séquence :

Pour commencer notre projet va donc avoir besoin d'une fonction qui va créer des vecteurs de Bierwitz. Ces vecteurs sont constitués des numéros de pièces, qui sont répétés autant de fois qu'il y a de machine. Ils sont tirés aléatoirement et permettent de changer l'ordre des pièces et des machines. Tout le projet repose sur ces vecteurs. Selon le vecteur le passage des machines par les pièces peut être totalement différent et donc changer le temps mis pour chaque pièce pour être fabriqués. Pour un même schéma et une même situation on peut donc avoir 2 vecteurs différents qui n'accorderont pas le même ordre et donc avoir 2 temps différents.

b) Evaluation d'un graphe :

La partie évaluation va consister, à partir du vecteur tiré précédemment, de reconstituer le graphe et d'évaluer pour chacun de ses sommets la date à laquelle il commence. Il donne aussi pour chaque sommet les pères de celui-ci afin de pouvoir par la suite remonter le graphe. Grâce à ceci on peut en fin de fonction calculer le cout du graphe pour le plus long chemin.

c) Conception d'une recherche locale efficace :

Pour ce qui est de la recherche locale, le but de la fonction est, d'à partir d'une solution (et donc d'un vecteur de Bierwitz), de réaliser un nombre d'itérations donnés en partant de la droite du graphe et en permuttant des pièces dans le vecteur de Bierwitz. Ces permutations permettent de changer les liens et donc de voir si il ne serait pas mieux de changer quelques liens dans le graphe. A chaque changement on recalcule le cout et si le cout de la nouvelle solution est plus petit on garde la nouvelle solution. On s'arrête après le nombre d'itérations, ou alors lorsque l'algo est remonté jusqu'à la gauche du graphe.

d) Conception d'un GRASP() :

Pour la conception d'un GRASP on va utiliser ce qu'on appelle des voisins. On va partir d'un vecteur et d'une solution de base. Et à partir de cette solution on va essayer de calculer le cout minimal en plusieurs itérations. On va tirer différents vecteurs et pour chaque vecteur on va tirer des vecteurs voisins (ce sont des vecteurs ou seulement 2 pièces changent). Après avoir tirés un nombre de voisins déterminés dès le début de la fonction, on va calculer pour chaque vecteur le cout du chemin qu'il propose et voir lequel est le meilleur. Plus le nombre d'itérations de cette fonction est grand, plus le cout minimal retourné devrait converger le vrai cout minimal. Pour gagner du temps et éviter de recalculer des vecteurs déjà calculés on va passer par une fonction de hachage qui permettra de savoir si le vecteur a déjà été évalué ou non mais cela sera expliqué dans la suite du compte rendu.

3.La description des algorithmes utilisés :

a) Procédure création de séquences :

Pour créer une séquence l'algorithme se déroule de la forme suivante :

```
Entree instance T, solution s
Cpt tableau d'entier

nbRep=T.n*T.m;
Pour i allant de 1 à T.n faire
    cpt[i]=T.m;
Fait;
j=1
Tant que nbRep>0 faire
    valide=false;
    tant que (!valide) faire
        alea= random(1,T.n);
        si (cpt[alea]>0) alors
            V[j]=alea;
            cpt[alea]=cpt[alea]-1;
            valide=true;
        Fsi;
        j++;
        nbrep--;
    Ftq
Ftq
```

Figure 1 : Procédure création de séquences

b) Procédure evaluation d'un graphe :

Pour l'évaluation l'algorithme se déroule de la forme suivante :

```
Entrée instance T et Solution s.
n'=T.n*T*m;

Pour i=1 à n' faire
    j=V[i];
    C[j]=C[j]+1;
    si( C[j]>1 ) alors
        m=ST[j][C[J]-1];
        si( m+T.P[j][C[j]-1] > S.ST[J] ) alors
            s.ST[j][C[j]]:=m+T.P[j][C[j]-1];
        Fsi;
    Fsi;
    mach=m[j][c[j]];
    si PM[mach]!=(0,0) alors
        (p,q)=PM[mach];
        date=s.ST[p][q];
        Si( date+T.p[p][q] > s.ST[j][C[j]] ) alors
            s.ST[j][C[j]]=date+T.p[p][q];
        Fsi;
    Fsi;
PM[mach]=(j,C[j]);
```

Figure : Procédure évaluation d'un graphe

c) Procédure Recherche Locale :

L’algorithme de la recherche locale est le suivant :

```
Entrée : V et V’ des vecteurs de Bierwitz, nbIter un nombre d’itération ;
    Evaluer(V) ;
    I=* (tuple de fin) ;
    J= Père(I) ;
    Stop=0 ;
    Tant que (stop = 0) faire :
        Cpt++ ;
        Si (machine de i = machine de j) alors :
            Chercher(posI) ;
            Chercher(posJ) ;
            V’=Permuter(V,posI,posJ) ;
            Evaluer(V’) ;
            Si coût de V’> coût de V alors
                I=J ;
                J=Père(I) ;
            Sinon
                V=V’ ;
                I=* ;
                J=Père(I) ;
            Fsi ;
        Sinon
            I=J;
            J=Père(I) ;
        Fsi ;
        Si(I==0) ou (cpt==nbIter) alors :
            Stop=1 ;
        Fsi ;
    Ftq ;
```

Figure 3 : Procédure recherche locale

Les fonctions Chercher retourne les dernières positions des numéros de pièces données en paramètre, Permuter, inverse les nombres aux deux positions passées en paramètre, et Père retourne le tuple père du tuple passé en paramètre.

d) Procédure GRASP() :

L'algorithme de la fonction GRASP est le suivant :

```
Entrée : instance T; solution s; entier : nb_element,nb_ite,nb_voisin;

Pour i allant de 1 à nb_ite faire
    Jusqu'à ce que tab_hash[s.hash]==0 faire
        generer(S);
        evaluer(S);
        rechercheLocal(S);
    Fjusqu'a;

    pour j allant de 1 à nb_element faire
        nb_voisins_courant=0;
        tant que (nb_voisins_courant!=nb_voisin)
            generer_voisin(voisin,V);
            rechercheLocale(voisin)

            si(tab_hash[voisin.hash]==1) alors
                //rien;
            sinon
                nb_voisin_courant++;
            Fsi;
        Ftq
    Fait
Fait
```

Figure 4 : Procédure GRASP

Après le tout on obtient donc une estimation du coût minimum de l'instance.

Générer voisins va générer un voisin du vecteur V en permutant aléatoirement deux nombres différents. Je ne trouve pas utile de mettre l'algo.

e)Procédure tester_double() :

Il n'y pas vraiment de procédure tester_double qui existe mais pour tester les doubles on utilise une table de hachage rempli de 0 ou de 1. Après une fonction de Hachage, chaque vecteur retourne un entier spécifique. Le code va ensuite vérifier dans la table de hachage si la valeur est 0 le programme va traiter le vecteur et changer la valeur à 1, et si la valeur est déjà 1 le programme ne va pas traiter le vecteur.

La fonction de hachage est la formule mathématique suivante :

Numéro de hachage= $\sum_i^n \sum_j^m (STij^2)\%k$ (k étant la taille de la table de hachage).

4.Etude sur une dizaine de séquences :

J’ai donc essayé mon code sur une dizaine de séquence de la matrice la01 donné sur le site :
http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt

On peut voir que ma fonction Grasp à un problème, pour une dizaine de test voici mes résultats avec comme paramètre :

-nbElements=10

-nb_Iter=1000

nbVoisins=15

On peut donc voir que la fonction ne retourne pas toujours les mêmes valeurs, j’ai essayé avec 10000 itérations ce n’était toujours pas

suffisant et au-dessus mon ordi commençait à peiner pour donner les dernières valeurs.

Je n’arrive donc pas à savoir si c’est que je ne fais pas assez d’itérations pour trouver les bons endroits de la courbe ou si ma fonction n’est pas bonne.