Outils d'aide à la décision TP3
Job Shop

Arquillière Mathieu - Zangla Jérémy

Table des matières

Table des figures

Liste des Algorithmes

1 Optimisation d'un Job Shop

1.1 Evaluation d'un graphe

On cherche ici à optimiser un problème de Job Shop. C'est un problème composé de n jobs, chacun ayant une séquence sur m machines. Par exemple prenons un problème à 3 jobs et 3 machines avec les séquences suivantes :

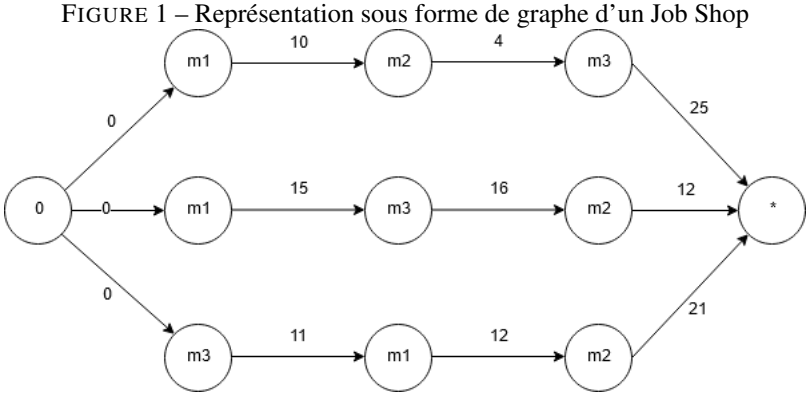
$$P_1 : (m_1, 10) \rightarrow (m_2, 4) \rightarrow (m_3, 25)$$

$$P_2 : (m_1, 15) \rightarrow (m_3, 16) \rightarrow (m_2, 12)$$

$$P_3 : (m_3, 11) \rightarrow (m_1, 12) \rightarrow (m_2, 21)$$

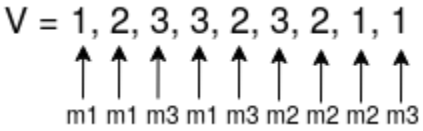
On modélise un Job Shop sous la forme d'un graphe. Celui-ci possède $n \times m$ sommets (n représentant le nombre de jobs et m le nombre de machines).

FIGURE 1 – Représentation sous forme de graphe d'un Job Shop



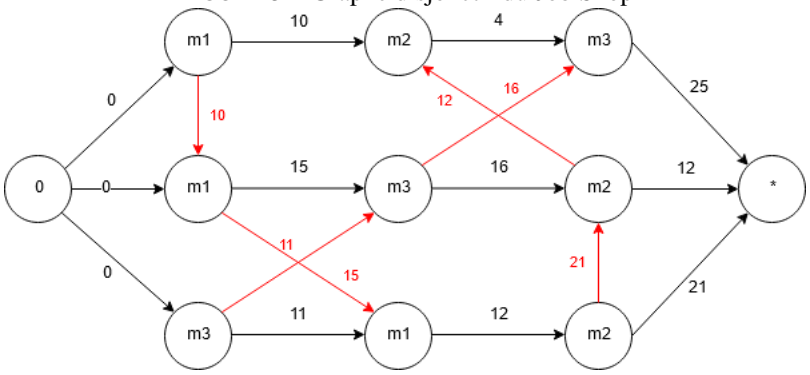
Pour obtenir une solution au problème, on utilise un vecteur de Bierwirth. En parcourant ce vecteur, on peut identifier les opérations correspondant à chaque élément et déduire la machine associé.

FIGURE 2 – Exemple de vecteur de Bierwirth



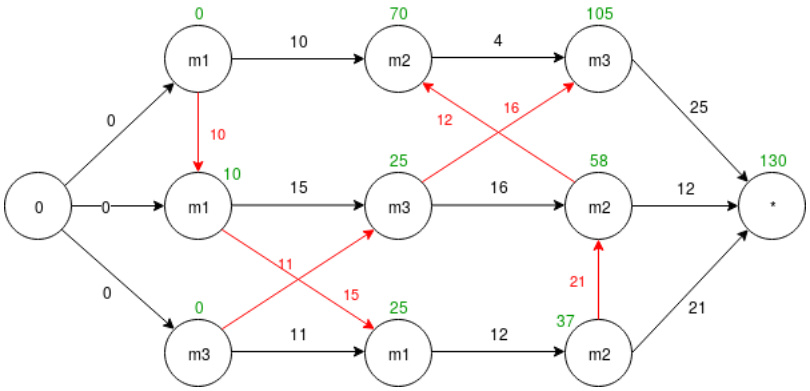
Cela permet de donner un ordre dans lequel sont exécutées les tâches. A partir de ça, on trouve les arcs disjonctifs entre les sommets du graphe et on obtient un graphe disjonctif.

FIGURE 3 – Graphe disjonctif du Job Shop



Enfin on effectue un algorithme de plus court chemin pour obtenir les *starting times* de chaque tâche.

FIGURE 4 – Graphe disjonctif du Job Shop avec les *starting times*



Pour créer l'algorithme correspondant à ces étapes, il faut d'abord définir les structures de données dont on aura besoin.

- *St* : un tableau de taille $n \times m$ pour contenir les *starting times*
- *pred* : un tableau de taille $n \times m$ pour contenir les précédents
- *P'* : un tableau de taille $n \times m$ contenant les "poids", les temps d'exécution de chaque tâche
- *U'* : un tableau de taille $n \times m$ contenant les séquences de chaque job
- *N* : un tableau de taille m contenant le nombre de fois qu'on trouve la machine $i \in 1,...,m$ pour trouver le prédecesseur
- *MP* : un tableau de taille m contenant la tâche passée en dernier sur la machine $i \in 1,...,m$

On a ensuite l'algorithme *évaluer* permettant d'obtenir les *starting times* et les prédecesseurs à partir d'un vecteur de Bierwirth.

```
1  pour i = 0 à N faire
2      j = V[i];
3      N[j] = N[j] + 1;
4      pos = T[j][N[j]];
5      si N[j] > 1 alors
6          prec = pos - 1;
7          si St[prec] + P'[prec] > St[pos] alors
8              St[pos] = St[prec] + P'[prec];
9              Pred[pos] = prec;
10         fin
11     fin
12     machine = U'[pos];
13     si MP[machine]! = 0 alors
14         prec = MP[machine];
15         si St[prec] + P'[prec] > St[pos] alors
16             St[pos] = St[prec] + P'[prec];
17             Pred[pos] = prec;
18         fin
19     fin
20     MP[machine] = pos;
21 fin
```

Algorithme 1 : Algorithme évaluer

1.2 Recherche Locale

Dans cette partie l'objectif est d'améliorer une solution. Pour cela, on effectuera des changements sur le chemin critique d'une solution existante. On parcourt en partant de la fin le chemin critique et lorsque deux tâches qui se suivent sur ce chemin sont sur la même machine, on tente d'inverser les deux éléments correspondants dans le vecteur de bierwirth et de recalculer une nouvelle solution, qu'on garde si elle s'est améliorée. Pour éviter que cet algorithme ne se finisse pas, on intègre un nombre maximum d'itérations à ne pas dépasser, dans notre cas *iter_max* est de 15.

```
1  i = NT;
2  j = pred[i];
3  tant que j ≠ 0 et iteration < iter_max faire
4      si U'[j] == U'[i] alors
5          Calculer position de i et j;
6          V' = V;
7          tmp = V'[pos_i];
8          V'[pos_i] = V'[pos_j];
9          V'[pos_j] = tmp;
10         evaluer(S');
11         si S' est meilleur que S alors
12             S = S';
13             V = V';
14             i = NT;
15             j = pred[i];
16         sinon
17             i = j;
18             j = pred[j];
19         fin
20     sinon
21         i = j;
22         j = pred[j];
23     fin
24 fin
```

Algorithme 2 : Algorithme de recherche locale

1.3 GRASP (Greedy Randomized Adaptive Search Procedure)

Cet algorithme (toujours visant à optimiser au maximum la solution trouvée) itere un certain nombre de fois la méthode de chercher 5 "voisins" d'une solution donnée et on garde la meilleure pour la prochaine itération. On obtient les voisins en échangeant des éléments du vecteur de Bierwirth, dans notre implémentation, la recherche de voisins boucle tant que l'on a pas trouvé 5 voisins différents et pas déjà utilisé (afin de ne pas bloquer l'algorithme, on itere cela un nombre maximum de fois, en l'occurence 25). De la même façon que la recherche locale, on met en place un maximum d'iterations (100 dans notre implémentation).

```
1  pour i = 1 à iter_max faire
2      Choisir au hasard V (vecteur de Bierwirth);
3      Avec de vecteur on obtient une solution S;
4      tant que on n'a pas obtenu 5 voisins différents faire
5          V = generer_un_voisin(S);
6          → permutation de 2 éléments dans le vecteur;
7          → evaluer et recherche locale;
8          Tester le voisin avec la table de hash;
9      fin
10     nS : nouvelle solution → meilleur des 5 voisins;
11 fin
```

Algorithme 3 : Algorithme GRASP

Pour savoir si les voisins trouvés sont bien différents et que l'on a pas déjà trouvé cette solution, on utilise une table de hachage. Cela consiste à trouver un identifiant unique (ou presque) à chaque solution et d'avoir un tableau indicé avec ces identifiants qui permet de savoir si la solution a déjà été trouvée ou non.

Formule de hachage :

$$h(S) = \sum_{i=0}^{n} (St_i)^2 \% k$$

n : nombre de sommets du graphe de la solution
k : taille du tableau de hash (suffisament grand).

2 Résultats

On utilise les instances de la OR-Library afin de réaliser nos tests.

2.1 Evaluer

FIGURE 5 – Résultats de notre implémentation de *evaluer* avec LA01.txt

```
Nombre de machines : 5
Nombre de pièces : 10
Temps total : 1069
Vecteur : ( 4 7 8 6 4 6 7 3 10 2 3 8 1 10 4 7 1 7 3 7 9 9 10 10 9 9 9 4 4 6 6 1 8 6 10 3 8 2 2 3 2
     2 1 5 5 8 1 5 5 5 )
Prédécesseurs :  ( 26 37 28  7  4 17 39  7  8  9 31 46 12 19 50 -1 16 12 45 19 15  4 22 23 24 16
    26 43 28 20 -1 31  1 33 34 -1  6 49 30  8 47 13 42 43 44 32 46 34 42 29 )
Starting times : ( 131 213 516 860 915 132 808 860 876 902 69 223 321 645 704  0 77 321 579 645
    716 915 949 1013 1032 77 131 437 516 722  0 69 152 239 326  0 153 487 784 876 302 363 412 437
    481 146 223 326 412 608 )
Chemin limitant (ordre inverse) : 25 <- 24 <- 23 <- 22 <- 4 <- 7 <- 39 <- 30 <- 20 <- 19 <- 45 <-
    44 <- 43 <- 42 <- 13 <- 12 <- 46 <- 32 <- 31
```

2.2 Evaluer puis recherche locale

FIGURE 6 – Résultats de notre implémentation de *recherche locale* avec LA01.txt

```
Nombre de machines : 5
Nombre de pièces : 10
Temps total : 1032
Vecteur : ( 4 7 8 6 4 6 7 3 10 2 3 8 1 10 4 7 1 7 3 7 9 9 10 10 9 9 9 4 6 6 6 1 8 4 10 3 8 2 2 3 2
     2 1 5 5 8 1 5 5 5 )
Prédécesseurs :  ( 26 37 28  7  4 17 39  7  8  9 31 46 12 19 50 -1 16 12 45 30 15  4 22 23 24 16
    26 43 28 29 -1 31  1 33 34 -1  6 49 20  8 47 13 42 43 44 32 46 34 42 29 )
Starting times : ( 131 213 516 823 878 132 771 823 839 865 69 223 321 645 704  0 77 321 579 670
    716 878 912 976 995 77 131 437 516 608  0 69 152 239 326  0 153 487 747 839 302 363 412 437
    481 146 223 326 412 608 )
Chemin limitant (ordre inverse) : 25 <- 24 <- 23 <- 22 <- 4 <- 7 <- 39 <- 20 <- 30 <- 29 <- 28 <-
    43 <- 42 <- 13 <- 12 <- 46 <- 32 <- 31
```

2.3 GRASP

Pour tester *GRASP*, on l'exécute 20 fois sur chaque instance.

Instance de Job-Shop	Temps moyen (ms)
LA01.txt	14.0759
LA02.txt	12.2875
LA03.txt	11.7334
LA04.txt	12.4472
LA05.txt	14.5207
LA06.txt	15.2889
LA07.txt	15.0473
LA08.txt	15.1696
LA09.txt	15.6707
LA10.txt	15.0829
LA11.txt	18.7042
LA12.txt	18.5126
LA13.txt	18.678
LA14.txt	18.2111
LA15.txt	18.6646
LA16.txt	14.1486
LA17.txt	13.5904
LA18.txt	14.5861
LA19.txt	13.0146
LA20.txt	13.8519