Enseignant : David HILL

Simulation ZZ2 - TP 4
Rabbit Population Growth

Ao XIE & Chloé BERTHOLD - Novembre 2022

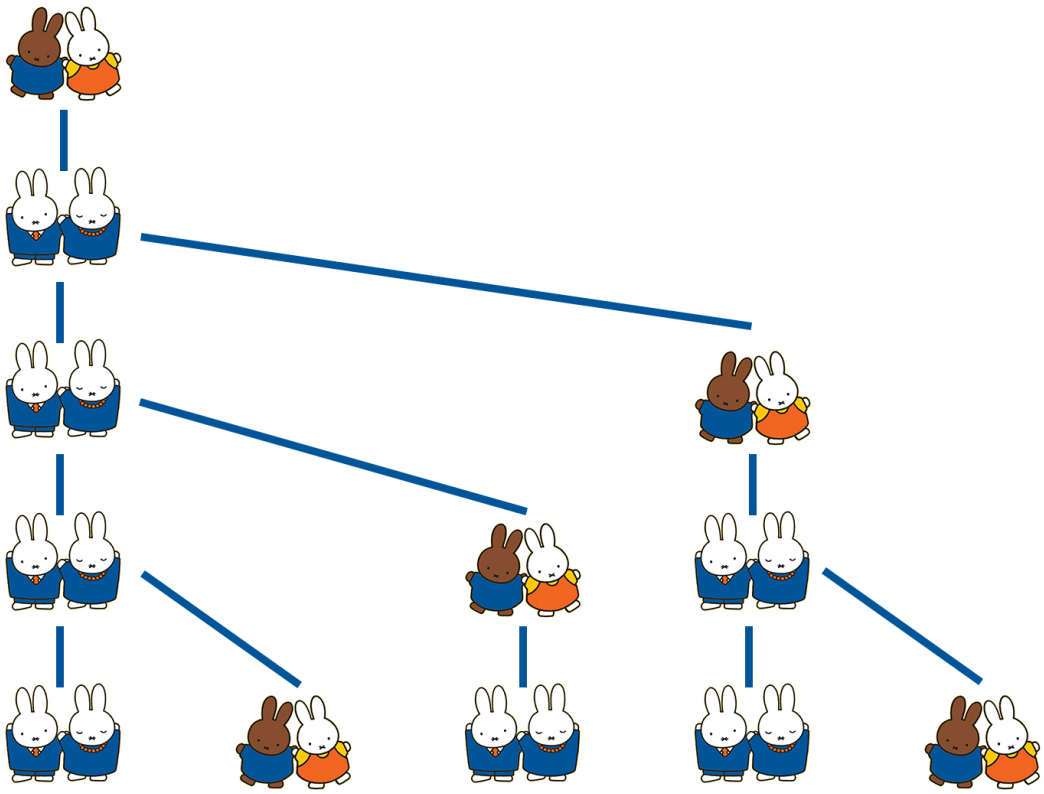

Table des matières

Table des figures

1 Présentation générale

1.1 Rappel des objectifs

La question de l'ordonnancement de l'atelier est cruciale pour l'usine[1]. Le problème d'ordonnancement (Job Shop Schedule Problem, JSSP) est donc au cœur des systèmes d'atelier. Il s'agit d'un problème d'ordonnancement complexe typique qui peut être décrit comme un ordonnancement rationnel de la séquence de traitement des tâches en fonction d'objectifs de production uniques ou multiples et des conditions environnementales de l'atelier, à condition que les contraintes soient satisfaites. La question est de déterminer les heures de début et de fin de chaque processus en fonction de l'ordre obtenu, c'est-à-dire n pièces à traiter sur m machines[2].

Les contraintes à respecter lors de l'usinage de ces pièces sont les suivantes :

— Une pièce ne peut pas être usinée sur plus d'une machine en même temps.
— Une machine ne peut traiter qu'une seule pièce à la fois.
— Le processus d'usinage de la pièce doit satisfaire aux exigences du parcours de la pièce.
— Une fois qu'une pièce a été usinée dans une machine, le processus est ininterrompu.

Ici, les algorithmes permettant le calcul du flux de travail le plus efficace sont écrit en C++ et il est considéré qu'il n'y a aucun temps de transport entre les étapes de l'usinage des pièces. nous avons écrit un algorithme en pipeline pour calculer le flux de travail le plus efficace en l'absence de temps de transport en utilisant le langage C++.

1.2 Organisation du code source

Ce TP est divisé en quatre fichiers :

— Le fichier *Header.h* qui regroupe les signatures de toutes les fonctions codées.
— Le fichier *Source.cpp* où sont implémentées les fonctions définies dans le fichier *header.h*.
— Le fichier *job_ shop.cpp* qui contient la fonction main ainsi qu'un exemple de l'exécution des différentes fonctions sur un graphe 10×10.
— Le fichier *la01.txt* qui contient la description d'un graphe 10×10.

Il huit fonctions sont incluses dans le programme, à savoir lire_fichier(), bierwith(), verifier_vecteur(), evaluer(), recherche_locale(), hashage(), permut() et grasp().

Dans ces fonctions, la fonction lire_fichier() permet de lire une structure de graphe dans un fichier et de la stocker dans une instance en paramètre, la fonction bierwith() permet de générer un vecteur de Bierwith aléatoirement dans le champ adapte de la structure passée en paramètre, la fonction verifier_vercteur() permet de vérifier que le vecteur de Bierwith de la solution passée en paramètre est bien construite et termine l'exécution si ce n'est pas le bonne situation, la fonction hashage() permet de calculer le hash de la solution passée en paramètre et la fonction permut() permet de réaliser une permutation simple du vecteur de Bierwith donne dans la structure de solution donnée en paramètre.

Les fonction evaluer(), recherche_locale() et grasp() sont expliquées plus en détails plus bas.

2 Fonctions de développement

Dans cette section, nous expliquons et analysons les points nécessaires dans la résolution d'un problème de Job-Shop.

2.1 La génération d'un graphe à partir d'une séquence

Pour traiter un problème de Job-Shop, il faut être capable de générer un graphe à partir d'une séquence le décrivant. Pour cela, il faut définir une norme pour ces séquences qui soit simple à lire et à comprendre. Ici, nous suivons la norme des fichiers LAX.

Un fichier est alors composé comme suit :

— Une première ligne ne contenant que le nombre de pièces et le nombre de machines, dans cet ordre et séparés par un espace.
— Une série de n lignes où n est le nombre de pièces. Chaque ligne se compose de plusieurs couples "machine temps d'usinage" pour représenter la séquence d'opération à suivre pour la pièce numéro i si i est le numéro de la ligne (sans compter la première ligne).

La fonction *lire_fichier()* permet de convertir ces séquences en un graphe.

2.2 L'évaluation du graphe

2.2.1 Principe

Une fois le graphe récupéré, il faut trouver une solution qui satisfasse toutes les contraintes de précédence entre les opérations d'une même pièce. Pour cela, on construit aléatoirement un vecteur de Bierwith qui permet de créer un ordre topologique des différentes opérations avec la garantie de ne jamais créer de cycle. Grâce à ce vecteur, il est alors possible d'évaluer le coût de la solution associée ainsi que de trouver les dates de début de toutes les opérations. Pour cela, il faut associer à chaque sommet du graphe (c'est-à-dire à chaque opérations) une marque qui réellement est la date de début estimée de l'opération, puis appliquer un algorithme du plus long chemin. En effet, si on utilise un algorithme du plus court chemin, on ne respecte pas les contraintes de précédence dans le graphe. Pour chaque sommet n'ayant pas de contrainte de précédence n'ayant pas encore été traitée, on considère tous ses fils (c'est-à-dire toutes les opérations pour qui il est une contrainte de précédence) et, si la marque du sommet considéré plus le temps associé à ce sommet est supérieure à la marque temporaire du fils, alors on garde cette nouvelle valeur comme marque pour le fils.

On procède donc par propagation. Ainsi, à partir d'un graphe et d'un vecteur de Bierwith, on peut trouver la solution correspondante.

2.2.2 La procédure *evaluer()*

La fonction *evaluer()* permet de calculer les dates de début de toutes les opérations et indique leur père dans les champs de la structure de solution passée en paramètre par rapport au graphe donné en paramètre et à un vecteur de Bierwith déterminé à l'avance.

L'algorithme est le suivant :

Algorithm evaluer()

Input: Structure du graphe considere; Structure de la solution;

```
1: for i = 1 → Nombredepieces * Nombredemachines do
2:     j ← s.vb[i]
3:     c[j] ← c[j] + 1
4:     if c[j] ≥ 1 then
5:         m ← s.t[c[j]]
6:         if PlusGrandCout then
7:             Enregistrements
8:         end if
9:     end if
10:    mach ← g.mach[j][c[j]]
11:    if AuMoinsUnePieceEstPassee then
12:        Enregistrements
13:        if PlusGrandDate then
14:            Enregistrements
15:        end if
16:    end if
17: end for
18: for i = 1 → NombreDePieces do
19:    if PlusGrandeDateFinale then
20:        Enregistrements
21:    end if
22: end for
```

FIGURE 1 – Procédure Evaluer

Avec :

- s : la structure de la solution où
  - .vb désigne le vecteur de Bierwith qui lui est associé
  - .t[pièce][opération] désigne la date de début de l'opération en deuxième facteur pour la pièce renseignée en premier facteur.
- c : un tableau d'entiers permettant de compter le nombre d'opérations traitées pour chaque pièce
- g : la structure du graphe considéré avec .mach[pièce][opération] donne la machine occupée pendant l'opération renseignée pour la pièce renseignée.

2.3 La recherche locale

2.3.1 Principe

Ne calculer qu'une solution n'est pas très efficace, on a de grandes chances de tomber sur un coût très élevé alors qu'il existe de bien meilleures solutions. C'est pourquoi il est important d'effectuer une recherche locale : une fois une solution sélectionnée, on s'en sert pour descendre le plus possible et aller chercher un minimum local. Il est crucial que cette recherche locale soit efficace car elle sera ensuite utilisée énormément de fois pour trouver le minimum global.

La recherche locale est effectuée en modifiant légèrement les arcs disjonctifs de la solution calculée de sorte à trouver une solution voisine très proche. Si cette solution est meilleure que la première, on la prend en compte et on réitère la recherche locale sur celle-ci. Pour cela, il faut remonter le vecteur de Bierwith associé à la solution pour trouver deux opérations de pièces différentes que l'on peut inverser de place dans l'ordre topologique. Ce changement n'affecte alors que l'ordre de passage sur une machine.

2.3.2 La procédure *recherche_ locale()*

La fonction *recherche_ locale()* permet de réaliser une amélioration de type descente à partir d'une première solution calculée grâce au vecteur de Bierwith renseigné dans la structure de solution donnée en paramètre.

L'algorithme est le suivant :

```
Algorithm recherchelocale()
Input: Structure du graphe considere; Structure de la solution; Nombre max d'iteration;
 1: if SolutionPasOptimale & !MaxIterationsAtteintes then
 2:     if MachinesOccupeesDifferentes then
 3:         PasseAuSuivant
 4:     else
 5:
 6:         while !PositionsTrouvees do
 7:             if !PositionITrouvee & s.vb[posi] == piece[i] then
 8:                 opi++
 9:                 if BonneOperationDecrite then
10:                     posi ← indice
11:                 end if
12:             end if
13:
14:             if !PositionITrouvee & s.vb[posi] == piece[j] then
15:                 opj++
16:                 if BonneOperationDecrite then
17:                     posj ← indice
18:                 end if
19:             end if
20:             indice++
21:         end while
22:
23:         v_prime = Permutation(v, posi, posj)
24:         EvaluerLaNouvelleSolution(v_prime)
25:         if CoutPlusEleve then
26:             Continuer avec v
27:         else
28:             Recommencer avec v_prime
29:         end if
30:     end if
31:
32:     iteration++
33: end if
```

FIGURE 2 – Procédure de Recherche Locale

2.4 La méta-euristique

2.4.1 Principe

La recherche locale permet d'obtenir un coût final beaucoup plus satisfaisant qu'une simple évaluation, cependant elle ne donne qu'un minimum local et pas le minimum global. C'est pourquoi il est nécessaire de mettre en place un algorithme qui va chercher le minimum global, ici l'algorithme Grasp. Pour cela, on cherche un minimum local à partir duquel on génère un certain nombre de voisins (permutations aléatoires du vecteur de Bierwith associé) pour parcourir l'espace et découvrir de nouveau minimum locaux, en mémorisant toujours la meilleure solution.

2.4.2 La procédure *grasp()*

Greedy Random Adaptive Search Procedure (GRASP) est une métathéorique, Il s'agit d'un algorithme d'optimisation classique et efficace. L'algorithme est divisé en deux parties, la première étant la phase de construction et la seconde la recherche locale[3]. Nous avons vu la deuxième partie dans la fonction recherche_locale().La partie principale de l'algorithme consiste à parcourir le graphe et à enregistrer la solution optimale pour chaque sommet. L'algorithme spécifique est le suivant :

Algorithm GRASP()
Input: Structure du graphe considere; Structure de la solution; Nombre max d'iteration; Nombre de voisins a chercher; Nombre max de recherche des voisins;

```
1: for iteration = 0 → nbMaxIterations do
2:     s = Minimum local aléatoire pas encore visité
3:     for i = 1 → Nombre max de recherche des voisins do
4:         voisins ← 0
5:         while voisins != nombre de voisins à chercher do
6:             Faire une permutation du vecteur de Bierwith de s
7:             Recherche locale sur le nouveau vecteur
8:             if Le minimum local n'a pas été visité then
9:                 if Le coût est meilleur que les autres voisins then
10:                    Meilleur voisin = minimum local trouvé
11:                end if
12:                Voisins + 1
13:            end if
14:
15:            if Meilleur voisin est le meilleur de tous depuis le début then
16:                best = meilleur voisin
17:            end if
18:        end while
19:    end for
20: end for
21: Return best
```

FIGURE 3 – Procédure Grasp

3 Étude de séquences

Nous allons à présent mettre les algorithmes à l'épreuve sur une batterie de séquences différentes et étudier les résultats. Pour cela, les séquences *la01* jusqu'à *la10* sont utilisées.

3.1 Évaluation simple

Comme expliqué plus haut, n'évaluer qu'une seule fois une solution n'est pas très efficace et peut donner des résultats très insatisfaisants.

Voici un tableau récapitulant une moyenne du coût total des solutions sur 1 000 expériences indépendantes pour les dix graphes pris en exemples :

Fichier utilisé	Moyenne sur 1 000 expériences	Solution optimale
la01	1166	666
la02	1141	655
la03	1043	597
la04	1089	590
la05	955	593
la06	1481	926
la07	1455	890
la08	1464	863
la09	1559	951
la10	1481	958

FIGURE 4 – Tableau récapitulatif des moyennes obtenues avec 1 000 évaluations

On remarque effectivement que les valeurs obtenues ne sont pas très bonnes voire très éloignées des valeurs optimales.

3.2 Recherche locale

La recherche locale permet de trouver un minimum local et donc d'obtenir des résultats beaucoup plus intéressants. Le tableau suivant présente une moyenne du coût total des minimums locaux explorés sur 1 000 expériences indépendantes pour les dix graphes pris en exemples :

Fichier utilisé	Moyenne sur 1 000 expériences	Solution optimale
la01	1166	666
la02	1141	655
la03	1043	597
la04	1089	590
la05	955	593
la06	1481	926
la07	1455	890
la08	1464	863
la09	1559	951
la10	1481	958

FIGURE 5 – Tableau des moyennes obtenues avec 1 000 recherches locales et le GRASP

La recherche locale en elle-même donne effectivement des meilleurs résultats mais reste insuffisante, c'est pourquoi il faut utiliser un algorithme comme le GRASP pour se rapprocher au maximum voire atteindre la solution optimale.

[1]

Références

[1] Tony C Scott and Pan Marketos. On the origin of the fibonacci sequence. *MacTutor History of Mathematics*, 23, 2014.