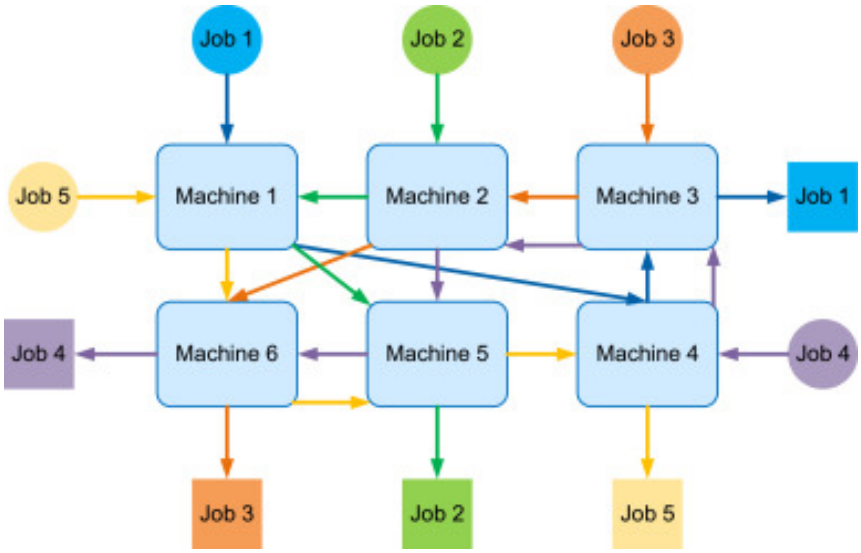
TP

# Aide à la décision - JS




CLIQUOT Théo
CHASSAGNOL Rémi

*Professeur:* LACOMME Philippe

November 3, 2022

# Contents

# 1 Introduction

## 1.1 Enoncé du problème

On cherche à résoudre un problème de **Job Shop**. Un **Job Shop** correspond à une séquence de passages dans des machines pour un ensemble de pièces. Chaque pièce passe dans chaque machines mais dans des ordres et pendant des durées différentes. Notre but est de minimiser le temps nécessaires pour que toutes les pièces ai finit leur processus.

## 1.2 Structure et Choix d'implémentation

Il nous faut stocker plusieurs informations tout du long de notre étude du job shop. Pour cela nous avons séparer nos données en 2 structures distinctes:

- Une structure `instance_t` qui va contenir:
  - Le nombre de Machines et de Pièces
  - Un tableau2D qui va nous renvoyer la machine correspondante à la pièce ($1^{er}$ argument) et au numéro d'ordre ($2^{nd}$ argument). (On peut linéariser le tableau afin d'avoir une meilleure optimisation.)
  - De la même façon, un tableau 2D qui va nous renvoyer cette fois-ci le temps que va passer la pièce n°i pour l'opération n°j (`T[i][j]`).
- Une structure `solution_t` avec:
  - Le coût de cette solution
  - Le temps pour une pièce et une opération donnée.
  - L'opération **père** pour une pièce et une opération donnée (l'opération `pere[i][j]` est l'opération qui précède l'opération de la pièces **i** sur la machine **j**).
  - Le **père** de notre élément étoile (le dernier élément, lorsque toute nos opérations sont finis).
  - Le **vecteur de Bierwith** qui a donné cette solution.
  - Le hash de cette solution. (utile pour le GRASP).

# 2 Implémentation

## 2.1 Lecture Dans un fichier

Nous avons déjà un choix de représentation des données du job shop si on suit la formalisation de brunel. Cette représentation est sous la forme:

- Une première ligne donnant le nombre de machines et de pièces
- Chaque ligne n°i suivante contient la suite d'opération (couple [machine coût]) pour la pièce n°i.

On peut donc récupérer toutes les informations de ce fichier avec une simple boucle `for` comme on peut le voir dans le code ci dessous:

```
fichier >> instance.nbPieces >> instance.nbMachines;

for (int i = 1; i <= instance.nbPieces; ++i) {
  for (int j = 1; j <= instance.nbMachines; ++j) {
    fichier >> machine;
    instance.machines[i][j] = machine + 1; // on compte     partir de 1
    fichier >> instance.coutPieceMachine[i][j];
  }
 }
}
```

## 2.2 Évaluer d'un graphe

L'évaluation nécessite d'avoir au préalable générer un **vecteur de Bierwith** avec la fonction dédiée. Le but de l'évaluation est de trouver les dates au plus tôt pour un vecteur de Bierwith donné, et donc de mettre à jours les tableaux `dates` et `pere` de `solution` et le coût de la solution. La fonction permet aussi de trouver le père du nœud ***** on en a besoin par la suite. Enfin, à la fin de l'évaluation, on calcule aussi le **hasch** de la solution, se **hash** sera utilisé par la suite.

Le fonctionnement de la fonctions est le suivant:

On traite les pièces suivant leur apparition dans le vecteur de Bierwith. On a donc une boucle qui parcours le vecteur. A chaque tour de boucle, on met à jour `pieceActuelle` qui correspond au numéro de la pièce courante, et `rangPiece` qui correspond au rang de la pièce. Pour trouver le rang, on utilise un tableau `occurencePiece` qui compte l'apparition des pièces.

```
// parcours du vecteur de bierwith et mise    jour de solution
for (int i = 1; i <= instance.nbMachines * instance.nbPieces; ++i) {
  pieceActuelle = solution.bierwith[i];
  rangPiece = ++occurencePiece[pieceActuelle];

  // ...
 }
```

Listing 1: Boucle principale de l'évaluation.

On commence par mettre à jours la date (et le père) suivant le lien horizontal. Ici, on modifie la date si $date(piecePrecedente) + cout(piecePrecedente) > date(pieceActuelle)$. A noter que le lien horizontal n'est mis à jours que pour les pièces de rang supérieur à 1 puisqu'une pièce de rang 1 n'a pas de père (le nœud **0** ne compte pas).

```
if (occurencePiece[pieceActuelle] > 1) {
  oldDate =
    solution.dates[pieceActuelle][rangPiece - 1]; // date du noeud pr c dent (
    horizontalement)

  if (oldDate + instance.coutPieceMachine[pieceActuelle][rangPiece - 1]
      > solution.dates[pieceActuelle][rangPiece]) {
    // mise    jours de la solution
  }
 }
```

Listing 2: Traitement de l'arc horizontal.

Ensuite on traite le lien disjonctif. Ici le principe est le même sauf qu'on regarde la pièce qui est passé en dernier sur la machine courante et la date est modifié si $pieceSurMachine + cout(pieceSurMachine) > cout(piece)$.

```
machineCourrante = instance.machines[pieceActuelle][rangPiece];

if (ordreSurMachine[machineCourrante][0] != 0) {
  numeroPieceMachine = ordreSurMachine[machineCourrante][0];
  rangPieceMachine = ordreSurMachine[machineCourrante][1];
  oldDate = solution.dates[numeroPieceMachine][rangPieceMachine];

  if (oldDate + instance.coutPieceMachine[numeroPieceMachine][rangPieceMachine]
      > solution.dates[pieceActuelle][rangPiece]) {
    // mise    jour de la solution
  }
 }
```

Listing 3: Traitement de l'arc disjonctif.

À la fin on met à jour la pièce sur la machine et on met à jours le coût de la solution si besoin.

```
// mise    jour de l'ordre de passage sur les machines
ordreSurMachine[machineCourrante][0] = pieceActuelle;
ordreSurMachine[machineCourrante][1] = rangPiece;

// si on traite le dernier noeud (*), on met    jour le cout de la solution
// et le p re du dernier noeud.
if (rangPiece == instance.nbMachines) {
  if (solution.dates[pieceActuelle][rangPiece]
      + instance.coutPieceMachine[pieceActuelle][rangPiece] > solution.count) {
    // mise    jours du cout et du p re de '*'
  }
 }
}
```

Listing 4: Mise à jour de la piece sur la machine et du coût

### 2.2.1 Calcul du hash

Dans les parties suivantes, on aura besoin de pourvoir tester si un solution a déjà été traitée, pour ce faire on aura besoin d'identifier les solution via un `hash`. La fonction de hachage suit la formule suivante:

$h = \sum_{d \in dates} (d^2 \% k)$ ou k est le nombre d'éléments de la table de hachage.

```
static inline int hachage(t_solution &solution, t_instance &instance) {
  unsigned long int h = 0;
  unsigned long int dateCourrante;

  for (int i = 1; i <= instance.nbPieces; ++i) {
    for (int j = 1; j <= instance.nbMachines; ++j) {
      dateCourrante = solution.dates[i][j];
      h = (h + (dateCourrante * dateCourrante)) % k;
    }
  }

  if (h > k)
    exit(-1);
  return h;
}
```

La fonction est `inline` pour avoir un code plus rapide.

## 2.3 Recherche locale

### 2.3.1 Raisonnement

Le but de cette fonction est de nous retourner la meilleure solution au niveau locale, c'est à dire qu'on va prendre en argument une solution actuelle, et on va chercher toutes les modification locales possibles améliorant notre solution. Les modifications locales correspondent à une permutation entre 2 opérations sur la même machine mais avec des pièces différentes (et on ne s'intéresse qu'au chemin critique vu que c'est celui ci qui nous limite, le chemin critique est trouvé en parcourant les pères, et ceux en commençant par notre noeud étoile).

La logique de cette algorithme est donc de parcourir notre chemin critique en sens inverse (jusqu'à arriver au noeud 0, le noeud de début), et durant le parcours du chemin, si on à un arc disjonctif, on va évaluer de nouveau notre solution en permutant cette fois-ci les deux opérations à l'origine de cette arc. Si la solution ainsi trouvé est meilleur ont repart du noeud de fin (étoile) et on recommence. Sinon (arc non disjonctif ou pire coût) on continue notre parcours du chemin critique.

### 2.3.2 Implémentation

Pour pouvoir permuter nos 2 noeuds dans le vecteur, on doit d'abord connaître leur position dans le vecteur de bierwith, C'est le but de notre fonction `recherchePosition` qui va nous prendre la pièce et le range de notre noeud courant et précédent et va nous renvoyer leur position dans le vecteur de bierwith. Pour cela on parcours le vecteur en sens inverse et dès qu'on voit notre pièce, on décrémente son rang. Dès qu'on atteint 0, on à trouvé notre index. Une optimisation apporté est de séparer le while on deux, en effet une fois qu'on à trouvé notre noeud courant, plus besoin de le chercher on peut se concentrer seulement sur le précédent.

```
while (!trouve) { // cour
  if (solution.bierwith[i] == pieceCour) {
    if (rangICour == rangCour) {
      indexCour = i;
      trouve = true;
    }
    rangICour -= 1;
  }
  if (solution.bierwith[i] == piecePred) {
    rangIPred -= 1;
  }
  --i;
 }
trouve = false;
while (!trouve) { // pred
  if (solution.bierwith[i] == piecePred) {
    if (rangIPred == rangPred) {
      indexPred = i;
      trouve = true;
    }
    rangIPred -= 1;
  }
  --i;
```

Une fois cette fonction écrite, nos parties se résume à:

```
// Si arc disjonctif
if (machineCour == machinePred) {
  solutionPrime = solution;

  rechercheePosition(solution, instance, pieceCour, rangCour, rangPred,
        piecePred, indexCour, indexPred);
  permuter(solutionPrime, indexCour, indexPred);

  // evaluation de la nouvelle solution
  evaluer(solutionPrime, instance);

  // si on am liore la solution
  if (solutionPrime.count < solution.count) {
    // On repart du noeud   toile   et on change notre solution
  } else {
    // On avance
  }
 } else {
  // On avance.
 }
```

## 2.4 GRASP

Le but de cet algorithme est de s'approcher au maximum de la meilleur solution du problème. Pour ce faire, on suit le schéma suivant:

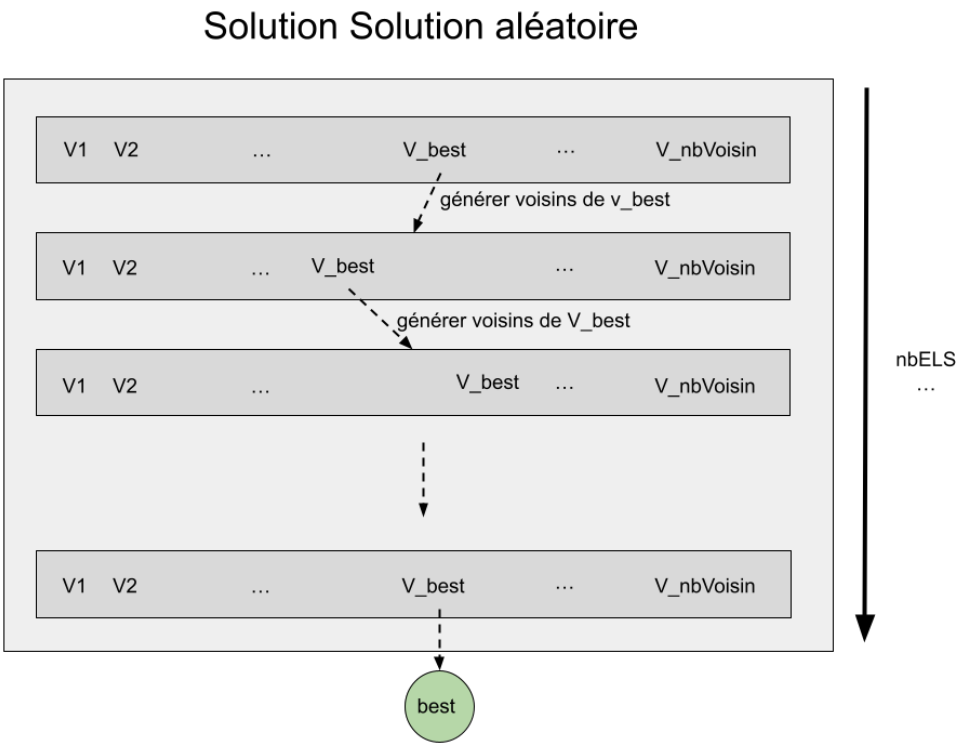



Figure 1: Principe du GRASP.

On commence par générer une solution aléatoire. Utiliser une table de hachage pour s'assurer que la solution que l'on traite n'a jamais été explorée.

```
do {
  genererVecteurBierwith(solution, instance);
  evaluer(solution, instance);
  rechercheLocale(solution, instance, 1000000);
 } while (T[solution.h] == 1);
T[solution.h] = 1; // marquage de la solution
```

Listing 5: Recherche d'une solution aléatoire non traitée.

Ensuite on génère `nbVoisin` voisins de la solution trouvée précédemment. Pour chaque voisin, on l'évalue, on fait une recherche locale, si le voisin n'est pas traité, on test si le coût est plus faible que celui de la meilleure solution trouvée jusqu'ici, on met à jour la meilleur solution.

```
for (int j = 1; j <= nbELS; j++) {
  int         nbv = 0;
  t_solution tmpBest;
  tmpBest.count = inf;

  // g n ration des voisins de la solution de d part + sauvegarde du
  // meilleur
  while ((nbv < nbVoisin && iter < 1000) || nbv == 0) {
    voisin = genererVoisin(solDepartELS, instance);
    evaluer(voisin, instance);
    rechercheLocale(voisin, instance, 1000000);
    if (T[voisin.h] == 0) {
      if (voisin.count < tmpBest.count)
  tmpBest = voisin;
      nbv++;
      T[voisin.h] = 1;
    }
    iter++;
  }
  solDepartELS = tmpBest; // on repart de la meilleur solution

  // sauvegarde du meilleur
  if (tmpBest.count < bestSolution.count)
    bestSolution = tmpBest;
 }
```

A noter que le nombre de voisin trouvé n'est incrémenté uniquement lorsque le voisin généré n'a pas été traité. De ce fait, on laisse une condition de sécurité pour éviter une boucle infinie dans le cas ou on arrive pas à générer `nbVoisin`.

On répète les étapes précédentes `nbIter` fois et on met à jour `bestSolution` à chaque fois. À la fin, `solution` prend la valeur de la meilleur solution trouvée.

# 3 Conclusion

## 3.1 Présentation des résultats

Voici un exemple des résultats qu'on obtient:

Table 1: Tableau des résultats

| Num de test | Résultat optimal | Résultat obtenu |
|---|---|---|
| la01 | 666 | 666 |
| la02 | 655 | 663 |
| la03 | 597 | 619 |
| la04 | 590 | 598 |
| la05 | 593 | 593 |
| la06 | 926 | 926 |
| la07 | 890 | 890 |
| la11 | 1222 | 1222 |
| la16 | 945 | ??? |
| la20 | 902 | ??? |

## 3.2 Analyse de ces derniers

On remarque que nos solutions sont très proches, et si on relance de façon aléatoire on obtient les bons résultats la plupart du temps. Cependant pour le cas du `la16` et `la20` le temps d'exécution est extrêmement long. Il nous faudrait faire des optimisation plus complexes pour avoir des temps de calculs plus raisonnables sur ces deux solutions.

Comme dit précédemment on peut linéariser les tableaux 2D afin d'avoir une meilleur optimisation, au prix d'une implémentation et d'une compréhension plus compliqué. Cependant, on constate tout de même que nos algorithmes sont relativement efficaces et trouve de bonnes solutions pour des problèmes simple relativement rapidement.