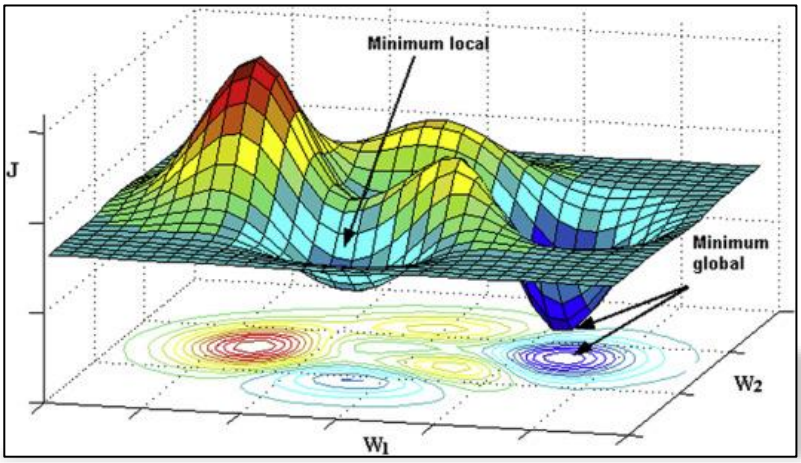
CLERMONT AUVERGNE INP Isima

Enseignant : Philippe Lacomme



Job-Shop et optimisation d'ordonnancement

TP2 OUTILS D'AIDE A LA DECISION

Anaël Courjaud | S1 ZZ2 F2 | 14/11/2022

Table des matières

I. Introduction

A. ENJEUX ET OBJECTIFS DU TP

1. Job-shop ? Flow-shop ?

Traduit de l'anglais – Un Job-Shop (*atelier de travail* en français) désigne généralement des petits systèmes de fabrication qui gèrent la production de travaux, c'est-à-dire des processus de fabrication personnalisés/sur mesure ou semi-personnalisés/sur mesure, tels que des commandes de clients de petite à moyenne taille ou des travaux par lots.

Typiquement, au sein de ces structures, on peut très vite se heurter à des problèmes d'ordonnancement qui vont drastiquement faire baisser la productivité s'ils ne sont pas traités correctement. On peut aussi parler de JSSP : *Job Shop Schedule Problem*.

Les problèmes d'ordonnancement en atelier proviennent du fait que des opérateurs doivent manœuvrer certaines machines pour usiner différentes pièces mais avec des ordres et pendant des durées différentes.

Par exemple, il peut arriver qu'un opérateur A se retrouve bloqué dans tout son processus de production car il lui manque un usinage court sur une de ses pièces mais que la machine M requise pour cet usinage court est monopolisée pendant très longtemps par un autre opérateur B qui est en train d'effectuer un usinage long. Une résolution possible de ce problème simple serait que l'opérateur B attende un peu pour laisser le temps à l'opérateur A d'effectuer son court usinage avec la machine M avant de se lancer dans son usinage long avec cette même machine.

En revanche la situation est bien plus complexe quand sont mis en jeu des dizaines de machines, de pièces et d'opérateurs. Les plus courageux d'entre nous vont certainement y laisser des plumes s'ils ne s'y prennent pas de la bonne manière. Et si par malheur on demandait à un ordinateur de calculer l'efficacité de l'intégralité des ordres de passage possibles et imaginables pour pouvoir sélectionner la meilleure solution, alors les temps de calculs s'accroîtraient de façon exponentielle pour chaque pièce, machine ou opérateur supplémentaire introduit dans le système. Pour résoudre correctement ce problème, il va falloir être plus malin que ça…

Pour commencer, il faut modéliser le problème pour pouvoir analyser facilement et de façon normée la situation par des algorithmes. C'est là qu'intervient le *flow-shop* :

Le flow-shop désigne un problème d'ordonnancement, typiquement rencontré dans les job-shops, qui définit un ensemble de n pièces, de m machines ainsi que trois types de contraintes :

- Les contraintes de gamme, toutes les pièces doivent passer sur toutes les machines.
- Les contraintes de ressource, une machine ne peut traiter qu'une seule pièce à la fois et une pièce ne peut pas être traitée par deux machines à la fois.
- Les contraintes d'ordre, une pièce ne peut pas passer sur les différentes machines dans n'importe quel ordre (On ne va pas commencer à peindre une planche avant de la raboter).

Il existe plusieurs formes simplifiées de flow-shop :

- Le flow-shop de permutation : Toutes les pièces passent dans le même ordre sur toutes les machines.
- Le flow-shop à deux machines.

- …

Les contraintes d'un flow-shop permettent de déduire que, pour un traitement algorithmique, on instancie le flow-shop avec n*m tâches (instanciation_n_m pour la suite du rapport). Une tâche est définie par une pièce, un numéro d'étape, une machine et une durée. Le couple (pièce ; machine) associé à une tâche est unique (i.e. : il n'existe pas de quadruplet (pièce ; étape ; machine ; durée) identiques), de même pour le couple (pièce ; étape). Dans mon code source, j'ai écrit « usinage » à la place de « tâche ». Pour la suite du rapport on adoptera également le terme « usinage ».

Dans ce TP, nous allons essayer d'implémenter un algorithme permettant de traiter n'importe quelle instance de flow-shop.

2. Que veut-on dire par « solution » ?

Dans un flow-shop, une solution induit un diagramme de Gantt respectant les contraintes initiales du flow-shop (pas d'usinages qui se chevauchent etc…). Cela veut dire que pour chacune des solutions, on trouve par le calcul sa durée d'exécution ainsi qu'un tableau associant à chacun des usinages sa date de début. Afin de construire le diagramme de Gantt, il suffira de lire ce tableau ainsi que le tableau des durées de chacune des tâches (donné à l'instanciation).

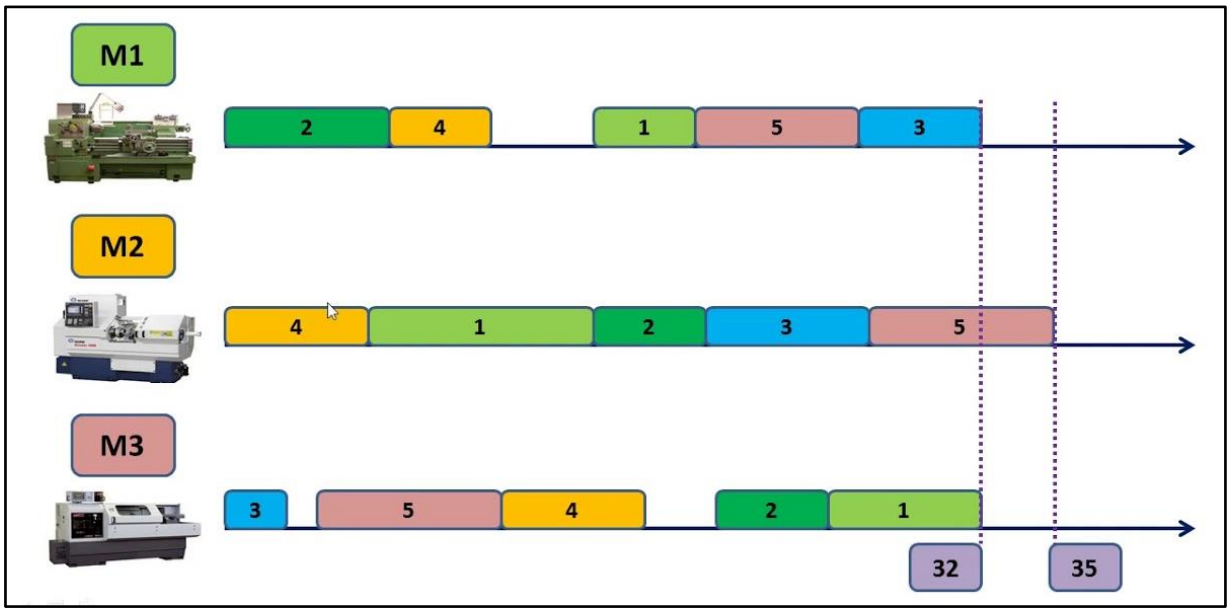


Figure 1 : Diagramme de Gantt d'une des meilleures solutions d'une instanciation n=5 m=3

Sur la figure 1, on compte bien n*m = 15 usinages (petits rectangles de couleur posés sur les flèches). On peut également lire 35 qui est la durée d'exécution de la solution. M1, M2 et M3 sont les machines et on peut compter 5 pièces différentes (petits rectangles de 1 à 5).

Visuellement, on peut facilement s'imaginer que, pour la même instanciation, il y a énormément de diagrammes de Gantt possibles. Ce qui confirme le fait qu'il y a énormément de solutions différentes par instanciation. Le but étant de trouver la meilleure possible, ce qui est le cas de la figure 1.

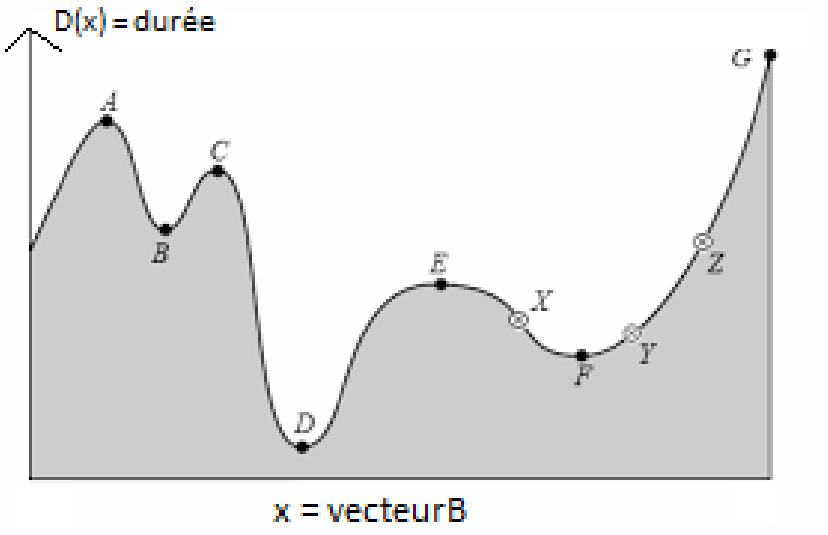
3. Définition de l'espace des solutions et principe des minimas locaux/globaux

Mais au final, que veut-on dire par "résoudre un flow-shop" ? Trouver une solution certes mais laquelle ? La plus efficace certes mais sous quelle forme ? Sous la forme d'un vecteur de Bierwirth. AH ! Voilà qui est intéressant, nous reviendront un peu plus tard sur les aspects techniques du vecteur de Bierwirth.

Ce qu'il faut en retenir, c'est qu'une solution est en fait donnée sous la forme d'un vecteur de Bierwirth (pour la suite du rapport vecteurB). Pour un flow-shop donné, il y a donc autant de solutions que de vecteursB.

Or, pour une instanciation_n_m, un vecteurB quelconque est de dimension n*m sachant que chaque dimension admet n valeurs distinctes. Il existerait donc n^(n*m) solutions mais en fait c'est moins que ça car il faut que chacune des n pièces soient présentes exactement m fois dans le vecteurB. Le nombre de solutions est donc un peu moins élevé que prévu mais l'ordre de grandeur reste valable.

On peut aisément se rendre compte que la représentation et la lecture graphique de la durée d'une solution en fonction de son vecteurB avec un axe par dimension est imbuvable. On va donc simplifier notre conception des vecteursB en les considérant avec une seule dimension. A cela on va appliquer une fonction D(x) qui va associer la durée d'exécution de la solution pour chaque vecteurB.



Ici on peut voir que les points B et F sont des minimums locaux et le point D est le minimum global. La solution associée au vecteurB correspondant au point D (solution D pour la suite) est donc la meilleure solution possible pour cette instanciation.

Pour un vecteurB donné, il suffit de permuter deux de ses valeurs (pas n'importe lesquelles, on verra ça un peu plus précisément plus tard) pour obtenir un de ses vecteursB voisins. Par exemple, on peut penser que le vecteurB correspondant au point F (vecteurB F) est le fruit d'une permutation au sein du vecteurB Y et de même pour le point Z.

On peut donc penser que pour trouver la solution F en partant de la solution Z, on a généré deux voisins de Z : Y et G. En calculant leurs durées, on s'est rendu compte que la solution Y est meilleure que la solution Z. Et de même en générant des voisins de Y, on s'est rendus compte que la solution F est la meilleure de cette « zone » : on a trouvé un minimum local. Cette procédure s'appelle un algorithme de descente et nous allons utiliser ce principe pour la fonction *rechercheLocale()* de mon code source.

4. Votre mission, si vous l'acceptez …

Notre objectif durant ce TP, va donc être de réaliser avec Visual Studio un programme en C++ qui va permettre de :

- Lire un fichier contenant les données d'instanciation d'un flow-shop rencontré dans un job-shop quelconque.
- Calculer la durée d'une solution quelconque donnée.
- En partant de cette solution, trouver son minimum local.
- Répéter ce processus avec un nombre donné (paramétrable) d'autres solutions aléatoires afin « d'explorer » tous les recoins de l'espace des solutions.
- Espérer tomber sur le minimum global lors de cette exploration ou, à défaut, trouver un minimum local qui s'en rapproche le plus possible.
- Faire tout ceci avec un temps de calcul raisonnable.

La suite du rapport va donc s'atteler à essayer de décrire le plus clairement possible la façon dont je m'y suis pris.

B. TOUR D'HORIZON DE MON CODE SOURCE

1. Principales structures

Les captures d'écran de code proviennent du fichier d'entête header.h.

Les informations nécessaires à définir une instance et un usinage sont stockées dans les types instance_t et usinage_t. (Rappel : une instance est composée de n*m couples (pièce ; étape) uniques : voir le dernier paragraphe de Job-shop ? Flow-shop ?). Dans la suite du rapport, pour parler de l'usinage correspondant au couple (pièce ; étape), on va parler de l'usinage de coordonnées (pièce ; étape).

```
typedef struct typeInstance {
    int nbrPieces;
    int nbrMachines;
    int machineUtilisee[NBRPIECESMAX][NBRMACHINESMAX];
    int dureeUsinage[NBRPIECESMAX][NBRMACHINESMAX];
} instance_t;

typedef struct typeUsinage {
    int numeroPiece;
    int numeroEtapePiece;
} usinage_t;
```

De même, j'ai créé un type solution_t dans lequel on va stocker un vecteurB sous la forme d'un tableau, puis, après évaluation du graphe avec ce vecteurB, on va stocker les données de la solution dans les structures correspondantes. Les noms des structures parlent d'eux-mêmes :

```
typedef struct typeSolution {
    int dureeSolution;
    int dateDebutUsinage[NBRPIECESMAX][NBRMACHINESMAX];
    /*
    Le pere d'un usinage désigne l'usinage correspondant à une certaine étape d'usinage d'une certaine piece dont
    la date de libération (achèvement) coïncide avec la date de début de l'usinage en question.
    Si un usinage débute à t=0, alors on admettra que son pere est tel que : numeroPiece = -1 et numeroEtapePiece = -1.
    Si l'usinage X d'une piece et l'usinage Y utilisant une machine se liberent en meme temps et que la piece et la
    machine sont tous les deux necessaires à un usinage Z, alors le pere de l'usinage Z sera l'usinage X.
    */
    usinage_t peresUsinages[NBRPIECESMAX][NBRMACHINESMAX];
    usinage_t pereUsinageTerminal;

    int vecteurB[TAILLEVECTEURBMAX];
} solution_t;
```

J'ai notamment créé la variable instance de type instance_t au tout début du main() avant de la remplir à l'aide de la fonction instanciation() car la majorité des fonctions du programme exploitent les données de l'instance.

Voici un résumé des fonctions définies et utilisées dans mon programme :

```
// lis un .txt pour instancier le flow-shop en remplissant la variable instance
void instanciation(std::string nom, instance_t& instance);

// génère aléatoirement un vecteurB compatible et l'enregistre dans la variable solution
void genererVecteurB(instance_t& instance, solution_t& solution);

// calcule les données de la solution induite par le vecteurB et l'enregistre dans solution
void evaluerSolution(instance_t& instance, solution_t& solution);

// Trouve la meilleure solution locale d'une solution et l'enregistre dans solution
void rechercheLocale(instance_t& instance, solution_t& solution);

// Dans vecteurB stocké dans solution : permute usinageCourant et le père de usinageCourant
void permutationDansVecteurB(solution_t& solution, usinage_t& usinageCourant);

// Génère nbrVoisinsGeneres voisins de solution et sélectionne la meilleure solution locale
// calculée pour chacuns d'eux avant de l'enregistrer dans solution.
void GRASP(instance_t& instance, solution_t& solution, int nbrVoisinsGeneres);

// vérifie si la solution d'un vecteurB a déjà  été calculée en amont ou pas
// Je n'ai pas réussi à implémenter cette fonction.
bool tester_double(solution_t& solution);
```

2. Fonction de hachage ?

Durant l'exécution de mon programme, il peut arriver qu'il se mette à calculer la solution associée à un vecteurB qui a déjà été calculé par le passé. C'est donc du temps d'exécution gaspillé puisqu'on sait qu'elle n'est pas une solution intéressante (une solution intéressante est une solution strictement améliorante de la meilleure solution courante). En revanche ce phénomène n'est pas nuisible lors de l'exécution de rechercheLocale() puisque même une solution inintéressante permet « d'accéder » à des solutions intéressantes.

J'ai donc prévu la fonction tester_double() que j'ai placé aux moments qui me semblaient opportuns. Cette fonction est censée empêcher l'exécution (en retournant « false ») d'un bout de code quand on sait que cette exécution donnera forcément une solution inintéressante. Malheureusement, cette fonction retourne actuellement toujours « true » car je n'ai pas réussi à lui implémenter une version satisfaisante.

Une fonction de hachage semble cependant être une bonne piste pour réussir l'implémentation de cette fonction.

II. Description algorithmique

A. INSTANCIATION DEPUIS UN FICHIER.TXT

Une instance_n_m est définie avec un fichier .txt comportant n+1 lignes utiles et m*2 colonnes utiles. La première ligne contient 2 nombres n et m séparés par un espace, respectivement le nombre de pièces et le nombre de machines.

Parmi les n prochaines lignes, chacune d'entre elles contient les informations relatives à la pièce qui lui correspond. Ligne 2 du .txt correspond à la pièce 0, la 3 à la 1 et ainsi de suite jusqu'à la ligne n+1 qui correspond à la pièce n-1.

Toutes les lignes (sauf la première) sont constituées de m couples (numéro de la machine utilisée, durée de l'usinage). L'ordre dans lequel sont écrit ces couples définit l'ordre dans lequel la pièce considérée va passer dans les machines. Par exemple le 2ème couple de la 2ème ligne définit la machine et la durée nécessaires pour l'usinage numéro 1 de la pièce numéro 0.

On obtient ainsi pour chaque couple de chaque ligne le quadruplet définissant un usinage (voir avant-dernier paragraphe de Job-shop ? Flow-shop ?Job-shop ? Flow-shop ?).Par exemple si, sur la ligne 8, le 5ème nombre est 4 et le 6ème nombre est 30, alors on lit (6, 3, 4, 30). Cela veut dire que pour la pièce 6, son 3ème usinage nécessite la machine 4 et a une durée de 30.

Lors de l'instanciation, pour chaque usinage, on va stocker la machine requise dans le tableau machineUtilisee[numéro de Pièce][numéro d'étape] et la durée nécessaire dans le tableau dureeUsinage[numéro de Pièce][numéro d'étape]. Ces deux tableaux sont stockés dans la variable instance de type instance_t. Ainsi pour chaque usinage de coordonnées (pièce ; étape), on va directement pouvoir récupérer le numéro de la machine requise ainsi que la durée nécessaire grâce à ces deux tableaux.

Voici un exemple de fichier .txt utilisé pour une instanciation (en l'occurrence, c'est l'instanciation LA19 de la OR-Library) :

```
10 10
2 44 3  5 5 58 4 97 0  9 7 84 8 77 9 96 1 58 6 89
4 15 7 31 1 87 8 57 0 77 3 85 2 81 5 39 9 73 6 21
9 82 6 22 4 10 3 70 1 49 0 40 8 34 2 48 7 80 5 71
1 91 2 17 7 62 5 75 8 47 4 11 3  7 6 72 9 35 0 55
6 71 1 90 3 75 0 64 2 94 8 15 4 12 7 67 9 20 5 50
7 70 5 93 8 77 2 29 4 58 6 93 3 68 1 57 9  7 0 52
6 87 1 63 4 26 5  6 2 82 3 27 7 56 8 48 9 36 0 95
0 36 5 15 8 41 9 78 3 76 6 84 4 30 7 76 2 36 1  8
5 88 2 81 3 13 6 82 4 54 7 13 8 29 9 40 1 78 0 75
9 88 4 54 6 64 7 32 0 52 2  6 8 54 5 82 3  6 1 26
```

B. GENERATION D'UN VECTEUR DE BIERWIRTH

J'utilise pour cela la fonction genererVecteurB(). Tant que le vecteurB en construction ne comporte pas n*m pièces, la fonction va réaliser des tirages aléatoires parmi les pièces puis vérifier si la pièce tirée n'est pas apparue déjà m fois avant de l'ajouter en dernière position du vecteurB le cas échéant.

La lecture d'un vecteurB est assez simple malgré les apparences : en fait chacun des éléments d'un vecteurB correspond à un usinage de coordonnées (pièce ; étape) avec pièce = numéro lu et étape = numéro d'apparition du numéro lu.

Voici un exemple vecteurB généré aléatoirement avec n = 10 et m = 5. On peut constater qu'il y a bel et bien exactement 5 exemplaire de chacune des 10 pièces.

```
VecteurB : 4 6 4 9 5 3 1 5 6 4 0 5 3 6 7 3 0 6 2 8 2 9
1 3 5 1 8 5 3 4 7 6 1 9 2 2 7 8 9 0 7 1 8 7 4 0 2 9 8 0
```

Prenons par exemple le 8ème élément de ce vecteurB : on peut lire que c'est un 5 or c'est le deuxième 5 qui apparaît dans ce vecteurB (le premier apparaît sur le 5ème élément). On peut donc dire que le 8ème élément correspond à l'usinage de coordonnées (pièce = 5 ; étape = 1).

C. EVALUATION DU GRAPHE

Cette fonction va, avec une instance donnée, calculer et remplir les données d'une solution correspondant à un vecteurB donné. Pour simplifier la notation, le terme solution fait référence à la structure rassemblant les données de la solution associée à un vecteurB.

1. Arc horizontal/disjonctif

Pour cela elle va parcourir le vecteurB, identifier pour chaque élément à quel usinage il correspond et ainsi construire les arcs disjonctifs ou horizontaux menant à chacun de ces usinages. Pour faire simple, un usinage nécessite que la pièce et la machine concernées soient disponibles au moment de son lancement. Généralement, on a un de ces deux éléments qui est disponible et on attend la libération de l'autre élément pour pouvoir lancer l'usinage. Dans ce cas, si la machine est disponible et que l'on attend la libération de la pièce pour lancer l'usinage, alors on parle d'arc horizontal. A l'inverse, si on attend la libération de la machine, alors on parle d'arc disjonctif. Il arrive de temps en temps que la machine et la pièce se libère exactement en même temps et qu'ainsi l'usinage se lance également en même temps. Par convention, on parle d'arc horizontal pour lever l'ambiguïté de ce cas-là.

2. Usinage père

L'utilité de déterminer si un usinage a été lancé via un arc horizontal ou disjonctif est d'ainsi pouvoir reconnaître le père de cet usinage. Le père d'un usinage A est un usinage dont l'achèvement a permis le lancement de l'usinage A. L'achèvement et le lancement de ces deux usinages sont donc forcément à la même date. Si un usinage de coordonnées (pièce ; étape) a été lancé avec un arc horizontal, alors on ne se pose pas de questions et le père est l'usinage (pièce ; étape - 1). Si en revanche un usinage (pièce ; machine) a été lancé via un arc disjonctif, alors le père est le dernier usinage traité par cette machine. Afin de retrouver facilement cet usinage, j'ai créé le tableau leastAffectationMachines[] de type usinage_t qui est mis à jour dès qu'un usinage est lancé. Enfin, si un usinage nécessite une pièce et une machine qui n'ont encore jamais été utilisée dans ce flow-shop, alors j'ai déterminé que son père est un père originel déclaré de façon conventionnelle et de coordonnées (pièce = -1 ; étape = -1). Une fois l'usinage père déterminé, on va stocker cet usinage dans le tableau peresUsinages[][] aux coordonnées de l'usinage A. Ce tableau est stocké dans la solution.

3. Dates et durées

A chaque lancement d'un usinage, le programme va mettre à jour le tableau dateDebutUsinage[][] (stocké dans la solution) en récupérant le date de début de l'usinage père et en y ajoutant sa durée nécessaire. L'usinage père originel débute à t = 0 et a une durée de 0.

Une fois le vecteurB entièrement parcouru, le programme va repérer la dernière machine en fonction et déterminer quelle pièce A elle est en train d'usiner. On sait qu'il s'agit de la dernière étape d'usinage de la pièce (donc l'étape numéro m-1 puisque la première étape est de numéro 0 et qu'il existe m étapes par pièces pour une instance_n_m). On sait donc qu'il s'agit de l'usinage de coordonnées (pièce = pièce A ; étape = m – 1) et que cet usinage est le père de l'usinage terminal. On va stocker cet usinage dans la variable pereUsinageTerminal de la solution.

L'usinage terminal est un usinage déclaré de manière conventionnelle dont la date de début est la date d'achèvement de son père et dont la durée est de 0. La lignée de pères dont descend l'usinage terminal constitue ce qu'on appelle le plus long chemin de la solution. On peut donc récupérer chacun des usinages constituant le plus long chemin grâce au tableau peresUsinages[][] et à la variable pereUsinageTerminal (stockés dans la solution). L'usinage terminal et son père sont toujours reliés par un arc horizontal.

Pour finir, on va stocker dans la variable dureeSolution de la solution la date d'achèvement de l'usinage terminal (qui est égale à la date d'achèvement de son père).

D. RECHERCHE D'UN MINIMUM LOCAL

Comme expliquée dans la partie « Définition de l'espace des solutions et principe des minimas locaux/globaux », on utilise pour la recherche d'un minimum local un algorithme de descente en partant du vecteurB donné initialement. On va donc commencer par évaluer le vecteurB initial avant d'essayer de trouver des voisins proches en permutant des valeurs du vecteurB d'une certaine manière. Il faut en fait remonter le plus long chemin en partant du père de l'usinage terminal puis, dès qu'on pour la première fois sur un arc disjonctif, alors on va prendre l'usinage qui a été lancé via cet arc et le permuter avec son père. On génère ainsi un nouveau vecteurB qu'on va évaluer avant de comparer sa durée avec la durée du vecteurB initial. Si la durée est strictement plus longue, alors on remplace le vecteurB initial par ce nouveau vecteurB et on le considère comme le nouveau vecteurB initial et on recommence le processus en partant du père de l'usinage terminal. Sinon, alors on garde la vecteurB initial et on continue le parcours de son vecteurB jusqu'à tomber sur le prochain arc disjonctif et ainsi de suite… Ce processus s'arrête au moment où le parcours du vecteurB initial actuel tombe sur le père originel. Ainsi on sait qu'on est tombé sur un vecteurB correspondant à un minimum local.

Voici un exemple d'exécution de la fonction rechercheLocale() avec l'instance LA01 :

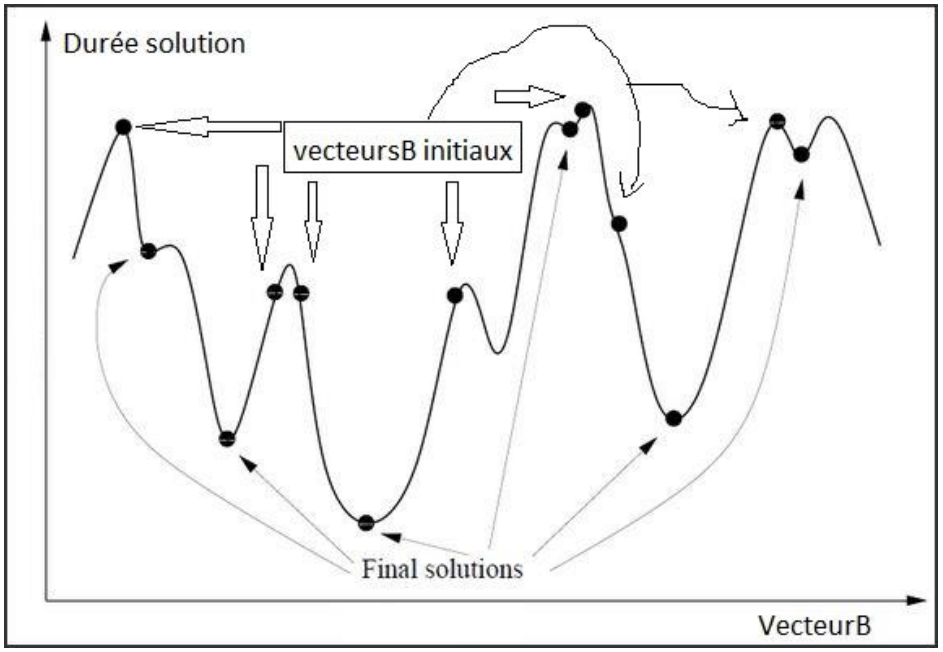
```
Fichier ouvert
VecteurB initial (aléatoire) : 27548547688417054172233881462265039536979910396001
Durée solution initiale : 1113 ; Durée solution voisine : 1113 --> je remplace !
Durée solution initiale : 1021 ; Durée solution voisine : 1021 --> je remplace !
Durée solution initiale : 1021 ; Durée solution voisine : 1039 --> je ne remplace pas.
Durée solution initiale : 1021 ; Durée solution voisine : 1021 --> je ne remplace pas.
Durée solution initiale : 1021 ; Durée solution voisine : 1021 --> je ne remplace pas.
Durée solution initiale : 990 ; Durée solution voisine : 990 --> je remplace !
Durée solution initiale : 990 ; Durée solution voisine : 1035 --> je ne remplace pas.
Durée solution initiale : 990 ; Durée solution voisine : 990 --> je ne remplace pas.
Durée solution initiale : 990 ; Durée solution voisine : 1025 --> je ne remplace pas.
Durée solution initiale : 944 ; Durée solution voisine : 944 --> je remplace !
Durée solution initiale : 944 ; Durée solution voisine : 947 --> je ne remplace pas.
Durée solution initiale : 944 ; Durée solution voisine : 944 --> je ne remplace pas.
Durée solution initiale : 940 ; Durée solution voisine : 940 --> je remplace !
Durée solution initiale : 940 ; Durée solution voisine : 944 --> je ne remplace pas.
Durée solution initiale : 930 ; Durée solution voisine : 930 --> je remplace !
Durée solution initiale : 930 ; Durée solution voisine : 960 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 886 --> je remplace !
Durée solution initiale : 886 ; Durée solution voisine : 960 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 904 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 886 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 886 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 915 --> je ne remplace pas.
Durée solution initiale : 886 ; Durée solution voisine : 886 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 878 --> je remplace !
Durée solution initiale : 878 ; Durée solution voisine : 952 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 900 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 878 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 878 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 915 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 878 --> je ne remplace pas.
Durée solution initiale : 878 ; Durée solution voisine : 886 --> je ne remplace pas.
Je suis tombé sur le père originel !
VecteurB final (minimum local) : 27548547685417084172328321468562039536979910369001
```

Le vecteurB obtenu n'est pas le vecteurB correspondant au minimum global (de durée 666 pour LA01) car on part d'un vecteur aléatoire qui fixe une zone de l'espace à explorer. En améliorant ce vecteur avec l'algorithme de descente, on explore les environs du point de départ ce qui a pour finalité de trouver un minimum local et non forcément le minimum global.

E. RECHERCHE DU MINIMUM GLOBAL

1. Principe

Lors d'une recherche locale, on va souvent se retrouver « bloqué » dans un minimum local, le programme n'arrivant pas à trouver un solution améliorante puisque tous ses voisins ont une durée plus longue (définition de minimum local). Il faut donc essayer de générer un maximum de vecteursB aléatoires distribués uniformément dans l'espace des solutions afin de pouvoir explorer un maximum de « recoins » de l'espace des solutions :



Ici, on voit bien que générer plusieurs vecteursB un peu partout dans l'espace des solutions a permis de trouver le minimum global.

On va donc commencer par générer beaucoup de vecteursB aléatoires, puis on va lancer la procédure GRASP() sur chacun d'eux. Cette procédure va retourner la meilleure solution trouvée pour un vecteurB et on va vérifier si elle est strictement améliorante de notre meilleure solution actuelle. Le cas échéant on va remplacer la meilleure solution actuelle par cette solution. Et ainsi de suite en espérant tomber sur le minimum global de l'espace des solutions.

2. La procédure GRASP

Voici une capture d'écran d'un article Wikipédia sur la procédure GRASP() :

Greedy randomized adaptive search procedure (GRASP) est une métaheuristique, c'est-à-dire un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile (au sens de la théorie de la complexité) pour lesquels on ne connaît pas de méthode classique plus efficace.

Introduite dans l'article Feo and Resende (1989), cette métaheuristique produit une solution réalisable. Elle est exécutée un certain nombre de fois et la meilleure solution trouvée est gardée. Pour produire une solution, deux phases sont exécutées l'une à la suite de l'autre : la première consiste en une phase de construction qui est suivie d'une phase de recherche locale.

Fonctionnement [modifier | modifier le code]

Comme le dit le nom de l'algorithme, la phase de construction combine les méthodes gloutonnes et aléatoires. La construction d'une solution se déroule par étapes et à chacune de celles-ci, l'ensemble des morceaux de solution qu'il est possible d'ajouter est placé dans une liste appelée RCL (Restricted Candidate List). Cette liste est triée, c'est la partie gloutonne de l'algorithme. Mais ce n'est pas nécessairement le meilleur morceau qui est ajouté à la solution courante. On tire aléatoirement parmi les meilleurs possibilités le morceau à ajouter, c'est la partie aléatoire de l'algorithme.

Grâce à la partie aléatoire, la phase de construction permet donc de varier la forme des solutions générées mais celles-ci sont quand même de bonne qualité, puisque le choix aléatoire se fait parmi un ensemble de bons candidats. La recherche locale s'applique sur la solution réalisable résultante de la phase de construction afin de voir s'il est encore possible d'améliorer cette solution.

Dans mon code source, je ne pense pas avoir réussi à reproduire fidèlement leur conception de la phase de construction. En effet, je n'ai pas de liste triée de bonnes solutions etc… Les seules fonctionnalités de ma fonction GRASP consistent en la génération aléatoires de voisins du vecteurB initial (en permutant deux éléments aléatoires du vecteurB) avant d'opérer une recherche locale sur chacun de ces voisins. Ces recherches locales vont retourner des minimums locaux qu'on va comparer entre eux pour retourner le meilleur d'entre eux.

3. Choix des paramètres de recherche

Les deux variables nbrVecteursBGeneresAleatoirement et nbrVoisinsGeneresGRASP sont déclarés au début du main().On peut ainsi facilement adapter facilement les paramètres de recherche. Généralement, plus les paramètres de recherche sont de valeurs élevées, plus la meilleure solution trouvée sera optimisée. Voici un exemple de cette convergence sur l'instance LA20 :

```
Fichier ouvert
nbrVecteursBGeneresAleatoirement = 20 et nbrVoisinsGeneresGRASP = 10
La meilleure solution trouvée a une durée de 1030
```

```
Fichier ouvert
nbrVecteursBGeneresAleatoirement = 80 et nbrVoisinsGeneresGRASP = 40
La meilleure solution trouvée a une durée de 929
```

```
Fichier ouvert
nbrVecteursBGeneresAleatoirement = 500 et nbrVoisinsGeneresGRASP = 250
La meilleure solution trouvée a une durée de 922
```

Adapter les paramètres en fonction de la complexité de l'instance est également utile pour éviter des temps de calcul inutilement longs sur des instances très simples dont le minimum global est facile à trouver.

Par exemple, pour l'instance LA01, avec nbrVecteursBGeneresAleatoirement = 40 et nbrVoisinsGeneresGRASP = 15, le programme a trouvé presque instantanément une solution de durée 666 ce qui est la plus petite durée théorique de cette instance.

En revanche, pour l'instance LA20, j'ai réglé nbrVecteursBGeneresAleatoirement à 2000 et nbrVoisinsGeneresGRASP à 500 et le programme a mis 3 bonnes minutes à s'exécuter. Par contre, dès

la première exécution, il a trouvé une solution de durée 910, ce qui est une très bonne solution puisque la meilleure solution possible a une durée de 902.

III. Quelques résultats sur les séquences LAXX

Voici un tableau rassemblant les résultats obtenus après une exécution avec les instances La01-La20 :

Instances	Durée optimale théorique	Durée trouvée avec :	nbrVecteursBGeneresAleatoirement	nbrVoisinsGeneresGRASP
LA01	666	666	40	15
LA02	655	662	800	200
LA03	597	615	800	200
LA04	590	598	800	200
LA05	593	593	800	200
LA06	926	926	800	200
LA07	890	890	800	200
LA08	863	863	800	200
LA09	951	951	800	200
LA10	958	958	800	200
LA11	1222	1222	800	200
LA12	1039	1039	800	200
LA13	1150	1150	800	200
LA14	1292	1292	800	200
LA15	1207	1220	1000	500
LA16	945	977	1000	500
LA17	784	797	1000	500
LA18	848	855	1000	500
LA19	842	867	2000	500
LA20	902	910	2000	250

IV. Conclusion

Tout d'abord, j'aimerais identifier plusieurs points qui permettraient d'améliorer mon programme, à commencer par mon choix de générateur de nombres aléatoires. J'ai utilisé la fonction rand() de la librairie standard alors que c'est quasiment considéré comme un sacrilège depuis qu'on découvert le Mersenne Twister, en TP de simulation. *Message d'excuse à M. David Hill : Je vous prie de m'excuser, je ne sais pas ce qu'il m'a pris de faire ça …*. Même si j'ai obtenu des résultats très satisfaisants d'après moi, peut-être que ma version potentiellement incomplète de la fonction GRASP() a été nuisible à l'efficacité du programme à trouver les meilleures solutions. Ensuite, évidemment, de nombreuses optimisations de code sont envisageables afin de grappiller des économies de complexité de ci de là.

Pour finir, je pense que le programme de mon code source a réussi à trouver de bons résultats sur les instances LA01 à LA20 et cela dans des temps raisonnables malgré mon code qui pourrait être amélioré de bien des façons.