École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
Fax +33 (0)2 47 36 14 22
polytech.univ-tours.fr

Département Informatique
4e année
2013 - 2014

Rapport - Projet de Sciences de la Décision

Heuristique pour un problème d'ordonnancement et de tournées

Encadrant
M. Jean-Charles BILLAUT
jean-charles.billaut@univ-tours.fr

Université François-Rabelais, Tours

Étudiants
Pierre-Antoine MORIN
pierre-antoine.morin@etu.univ-tours.fr
Armand RENAUDEAU
armand.renaudeau@etu.univ-tours.fr

DI4 2013 - 2014

Table des matières

Table des figures

Liste des tableaux

Introduction

Ce mini-projet de Sciences de la décision est une continuation de notre ancien projet de Gestion de Flux de Production. Le but de celui-ci est d'améliorer l'heuristique que nous avions mis en place la dernière fois en appliquant de nouvelles méthodes de résolution.

Dans le premier chapitre de ce rapport, nous présenterons le problème abordé, à savoir $F_m \mid \sum T_j$ avec VRP (*Vehicle Routing Problem*). Ensuite, nous évoquerons les attentes de notre encadrant dans le cahier des charges. Dans le troisième chapitre, nous aborderons l'algorithme de l'heuristique. Apres ce chapitre, nous parlerons de comment nous avons implémenté notre heuristique dans un langage de programmation. Dans le dernier chapitre, nous présenterons les résultats que l'heuristique a produit sur de nombreuses instances ainsi qu'une analyse de ces résultats Enfin, nous terminerons par une conclusion personnelle à propos des enjeux de ce projet et de ce que nous en avons appris.

Présentation du problème

2.1 Description générale

Une entreprise est spécialisée dans la fabrication de produits. Le procédé de fabrication est unique, il correspond à la gamme suivante : tous les jobs doivent être traités successivement par toutes les machines, dans le même ordre. L'atelier de production (l'usine) est organisé en flowshop.

Après fabrication, l'entreprise assure la livraison de la marchandise jusqu'à un site choisi par le client. Pour ce faire, l'entreprise dispose d'un unique véhicule, de capacité supposée infinie. (au cours d'une tournée de livraison, on peut transporter une quantité de marchandise aussi grande que voulue)

Le carnet de commandes est constitué de tâches (jobs), pour lesquelles on connaît :
- la durée de passage sur chacune des machines ;
- la date à laquelle le client souhaite être livré ;
- le site sur lequel livrer la marchandise.

Les temps de trajet d'un site à un autre, y compris l'usine, sont tous connus.

Le retard d'un job est défini ainsi :
- Si la marchandise est livrée sur le site choisi par le client avant la date demandée, alors le retard est nul.
- En revanche, si cette date limite est dépassée, le retard est égal à la différence entre la date de livraison effective et la date de livraison demandée.

Quelle organisation en atelier et quelles tournées de livraison effectuer afin de minimiser la somme des retards ?

2.2 Formulation mathématique

Données caractérisant la taille d'une instance

n Nombre de jobs J_j = Nombre de sites S_a en plus de l'usine $U = S_0$
m Nombre de machines M_i

Données caractérisant un job J_j

$\forall i = 1\,..\,m$, $\forall j = 1\,..\,n$ $\quad p_{i\,j}$ *Processing time* du job J_j sur la machine M_i

$\forall j = 1\,..\,n$ $\quad d_j$ *Due date* du job J_j

Données caractérisant les durées de transport de marchandises

$\forall a = 0\,..\,n$, $\forall b = 0\,..\,n$ $\quad t_{a\,b}$ Durée de voyage pour aller du site S_a au site S_b

Remarque importante

Le job J_j doit être livré sur le site S_j. On suppose que tous les sites sont différents. En pratique, si les jobs J_a et J_b doivent être livrés sur le même site, on posera : $t_{a\,b} = t_{b\,a} = 0$.

Format des données

$\mathcal{P}$	J_1	$\cdots$	J_j	$\cdots$	J_n
$\mathcal{M}_1$	$p_{1\,1}$	$\cdots$	$p_{1\,j}$	$\cdots$	$p_{1\,n}$
$\vdots$	$\vdots$		$\vdots$		$\vdots$
$\mathcal{M}_i$	$p_{i\,1}$	$\cdots$	$p_{i\,j}$	$\cdots$	$p_{i\,n}$
$\vdots$	$\vdots$		$\vdots$		$\vdots$
$\mathcal{M}_m$	$p_{m\,1}$	$\cdots$	$p_{m\,j}$	$\cdots$	$p_{m\,n}$
$\mathcal{D}$	d_1	$\cdots$	d_j	$\cdots$	d_n

$\mathcal{T}$	U	S_1	$\cdots$	S_b	$\cdots$	S_n
U	**0**	$t_{0\,1}$	$\cdots$	$t_{0\,b}$	$\cdots$	$t_{0\,n}$
S_1	$t_{1\,0}$	**0**	$\cdots$	$t_{1\,b}$	$\cdots$	$t_{1\,n}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
S_a	$t_{a\,0}$	$t_{a\,1}$	$\cdots$	$t_{a\,b}$	$\cdots$	$t_{a\,n}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
S_n	$t_{n\,0}$	$t_{n\,1}$	$\cdots$	$t_{n\,b}$	$\cdots$	**0**

Objectif

$$minimiser \left(\sum_{j=1}^{n} T_j \right)$$

où la variable T_j représente le retard associé à la livraison du job J_j. $(T_j \geq 0)$

Cahier des charges

Les exigences de notre encadrant pour ce projet sont les suivantes :

1. Créer un algorithme d'une heuristique permettant la résolution d'un problème d'ordonnancement avec routing. Cette heuristique propose une solution initiale et, à l'aide de mutations effectuées sur la répartition des jobs en tournées de livraison, elle essayera d'améliorer cette dernière.
2. Implémenter l'heuristique en langage C.
3. Tester celle-ci sur plusieurs instances.
4. Analyser les résultats de cette heuristique pour voir quelle a été son efficacité, faire une critique et proposer des améliorations.

Nous allons maintenant passer à la prochaine partie qui est l'explication de l'algorithme de l'heuristique.

Algorithme utilisé pour l'heuristique

4.1 Recherche d'une solution initiale

L'algorithme de descente locale va effectuer des mutations à partir d'une solution initiale. Cette solution initiale est obtenue selon l'algorithme suivant :

```
/* Étape 1 : Tri des jobs selon EDD */
job[] ← trier_jobs_selon_EDD(instance)
/* Étape 2 : Initialisation, création de la première tournée */
créer_nouvelle_tournée()
ajouter_dans_dernière_tournée(job[1])
/* Étape 3 : Double simulation pour les autres jobs */
Pour j = 2 à n Faire
    retard_insertion ← simuler_insertion_dans_dernière_tournée(job[j])
    retard_création ← simuler_création_nouvelle_tournée(job[j])
    Si retard_création < retard_insertion Alors
        créer_nouvelle_tournée()
    Fin Si
    ajouter_dans_dernière_tournée(job[j])
Fin Pour
retourner solution_initiale
```

Algorithme 1: Recherche d'une solution initiale

Les jobs sont d'abord triés par date de livraison croissante (tri EDD). Ensuite, les jobs sont considérés un à un dans cet ordre, et la meilleure de ces deux alternatives est choisie :

- Le job considéré est ajouté à la dernière tournée de livraison.
- Le job considéré est inséré dans une nouvelle tournée de livraison.

L'algorithme de NEH est utilisé pour constituer les ordonnancements. L'algorithme de recherche du plus proche voisin est utilisé pour constituer les tournées de livraison.

Cet algorithme est une optimisation d'un algorithme précédemment implémenté par les étudiants au cours du premier semestre, à l'occasion du mini-projet GPF (Gestion de Production et des Flux).

4.2 Voisinage d'une solution

Dans toute solution, les jobs sont répartis dans différentes tournées de livraison. En numérotant ces tournées dans l'ordre chronologique, il est possible d'utiliser un tableau d'entiers pour associer un numéro de job et un numéro de tournée.

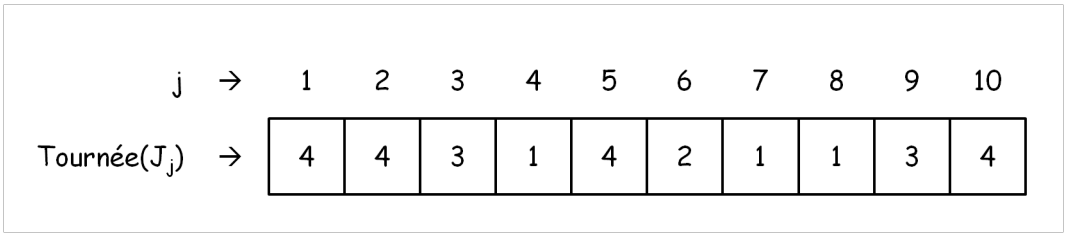

FIGURE 4.1 – Représentation de l'affectation des jobs aux tournées de livraison

Dans cet exemple, les 10 jobs sont répartis dans les tournées suivantes :

1. Tournée de livraison 1 : jobs 4, 7 et 8 ;
2. Tournée de livraison 2 : job 6 ;
3. Tournée de livraison 3 : jobs 3 et 9 ;
4. Tournée de livraison 4 : jobs 1, 2, 5 et 10.

Définition du voisinage d'une solution

Étant donné une solution appelée « solution mère », le voisinage de cette solution est un ensemble de solutions, appelées « solutions filles ». La répartition des jobs en tournées de livraison pour ces solutions filles suit la règle suivante :

1. Tous les jobs *sauf un* sont répartis dans les mêmes tournées de livraison que dans la solution mère.
2. Le *job restant* est inséré dans une autre tournée : s'il appartenait à la tournée k, il appartient désormais soit à la tournée $k-1$ (précédente), soit à la tournée $k+1$ (suivante).

Exemple

Considérons la solution précédente comme solution mère. En diminuant le numéro de la tournée associée au job 5, on obtient la solution fille suivante :

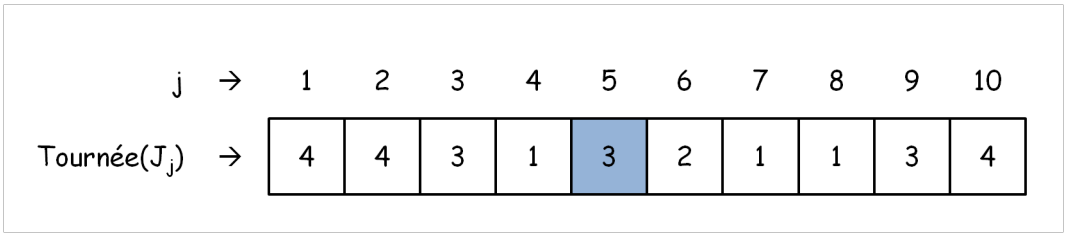

FIGURE 4.2 – Exemple de solution fille

Dans cet exemple, les 10 jobs sont répartis dans les tournées suivantes :

1. Tournée de livraison 1 : jobs 4, 7 et 8 ;
2. Tournée de livraison 2 : job 6 ;
3. Tournée de livraison 3 : jobs 3, 5 et 9 ;
4. Tournée de livraison 4 : jobs 1, 2 et 10.

4.3 Descente locale

L'algorithme utilisé est le suivant :

```
solution_mere ← trouver_solution_initiale(instance)
continuer ← vrai
Tant que continuer Faire
    continuer ← faux
    /* Génération de solutions filles */
    Pour k = 1 à nombre_mutations Faire
        solution_fille[k] ← générer_mutation(solution_mere)
        /* L'algorithme fait en sorte que toutes les solutions filles
        soient différentes de la solution mère et des autres solutions filles. */
    Fin Pour
    /* Sélection de la meilleure solution fille */
    Pour k = 1 à nombre_mutations Faire
        Si somme_retards(solution_fille[k]) < somme_retards(solution_mere) Alors
            solution_mere ← solution_fille[k]
            continuer ← vrai  /* Effectuer une nouvelle itération */
        Fin Si
    Fin Pour
Fin Tant que
retourner solution_mere
```

Algorithme 2: Algorithme de l'heuristique, descente locale à partir d'une solution initiale

À partir de la solution « mère » initiale trouvée, certaines solutions « filles » sont sélectionnées **aléatoirement** dans le voisinage. Ce processus est appelée « mutation ». Si l'une des solutions filles est meilleure que la solution mère, elle devient la solution mère utilisée pour la prochaine itération. L'algorithme continue jusqu'à ce que la solution mère soit meilleure ou équivalente à toutes ses filles (critère : somme des retards).

Le nombre de solutions filles considérées à chaque itération ($nombre\_mutations$) est un paramètre optionnel de l'algorithme. Si aucune valeur n'est spécifiée lors de l'invocation du programme, la valeur par défaut utilisée est égale au nombre de jobs dans l'instance.

Attention Le voisinage d'une solution est un ensemble de taille finie. Or, à chaque itération, l'algorithme fait en sorte de générer des solutions distinctes. Par conséquent, si la valeur du paramètre $nombre\_mutations$ est trop élevée, le programme tournera indéfiniment, car il restera bloqué au niveau de la boucle de génération des solutions filles.

Implémentation

Comme spécifié dans le cahier des charges, notre programme a été implémenté en utilisant le langage C. Pour fonctionner, le programme a besoin d'un fichier contenant différentes données qui sont présentées d'une manière bien précise. À la fin de l'exécution, le programme crée un fichier de sortie contenant des données également présentées d'une manière bien précise. Dans un premier temps, nous allons détailler le format des fichiers d'entrée et de sortie, puis, dans un deuxième temps, la documentation qui présente l'architecture du projet.

Les fichiers d'en-têtes sont disponibles dans le dossier `program/headers/` et les fichiers sources sont disponibles dans le dossier `program/sources/`. La création du programme compilé peut se faire à l'aide de la commande *make* grâce au *Makefile* qui est présent dans le dossier `program/`. L'executable alors généré est présent dans le dossier `program/bin/` et se nomme *program.exe*.

5.1 Format des fichiers

5.1.1 Fichiers d'entrée

Comme indiqué précédemment, les fichiers d'entrée sont structurés d'une manière bien précise. Concernant le nom des fichiers, ils sont tous nommés de la même manière :

`I_(nombre de machines)_(nombre de jobs)_(numero d'instance).txt`.

Voici comment sont structurés ces fichiers.

(nombre de machines)(tabulation)(nombre de jobs)
(durées sur la première machine, séparées par des tabulations)
...
(durées sur la dernière machine, séparées par des tabulations)
(dates dues, séparées par des tabulations)
(distances de l'usine aux autres sites, séparées par des tabulations)
(distances du premier site aux autres sites, séparées par des tabulations)
...
(distances du dernier site aux autres sites, séparées par des tabulations)

5.1.2 Fichiers de sortie

Les fichiers de sorties sont tous nommés de la même manière :

`S_(nombre de machines)_(nombre de jobs)_(numero d'instance).txt`.

Voici comment sont structurés ces fichiers.

```
(somme des retards)
(temps CPU en millisecondes)
(numéros des jobs selon la séquence d'ordonnancement, séparés par des tabulations)
(nombre de tournées)
(nombres de sites par tournée, séparés par des tabulations)
(numéros des sites de la première tournée selon l'ordre de parcours, séparés par des tabulations)
...
(numéros des sites de la première tournée selon l'ordre de parcours, séparés par des tabulations)
```

5.2 Documentation

Le projet a été documenté grâce à l'outils doxygen. Pour visualiser la documentation, il faut se rendre dans le dossier du projet et ouvrir le fichier index.html qui se trouve dans le dossier `documentation/html/`.

5.3 Utilisation du programme

Pour fonctionner, le programme que nous avons réalisé a besoin de plusieurs paramètres :

1. Le fichier contenant l'instance à traiter.
2. Le fichier contenant les résultats après l'exécution du programme.
3. Le nombre de mutations par itération. Ce paramètre est optionnel, c'est-à-dire que s'il n'est pas indiqué, le nombre de mutations effectuées est égal au nombre de jobs de l'instance.

Résultats

6.1 Automatisation du processus de test : scripts Bash

Pour tester notre programme, nous avons sollicité le doctorant Quang Chieu Ta, dont le sujet de thèse porte sur le problème mathématique dont il est question dans ce projet de « Sciences de la Décision ». Celui-ci nous a fourni 160 instances, réparties en 16 classes contenant 10 instances chacune :

- Instance à 2 machines et ...
  - ... 10 jobs
  - ... 20 jobs
  - ... 30 jobs
  - ... 50 jobs
  - ... 70 jobs
  - ... 100 jobs
  - ... 150 jobs
  - ... 200 jobs
  - ... 250 jobs
  - ... 300 jobs
- Instance à 4 machines et ...
  - ... 10 jobs
  - ... 30 jobs
  - ... 50 jobs
  - ... 100 jobs
  - ... 150 jobs
  - ... 200 jobs

Afin de faciliter la phase de réalisation des tests, nous avons utilisé deux scripts Bash :

1. `call.sh` : script utile à l'invocation du programme de manière automatisée
2. `compare.sh` : script utile à la comparaison des solutions retournées après exécution du programme

Ces deux scripts sont disponibles dans le dossier `scripts/` fourni avec ce rapport.

6.1.1 Automatisation de l'invocation de l'exécutable sur plusieurs instances

Le script `call.sh` prend en entrée les paramètres suivants :

1. le chemin vers le fichier exécutable correspondant au programme ;
2. le chemin vers un dossier `instances/` contenant des instances ;
3. le chemin vers un dossier `solutions/` (existant ou à créer) qui contiendra les solutions ;
4. le nombre de mutations à effectuer à chaque itération (optionnel).

Le script invoque automatiquement le programme sur toutes les instances du dossier `instances/` et les écrit dans le dossier `solutions/`.

6.1.2 Automatisation de la comparaison des résultats obtenus

Le script `compare.sh` prend en entrée les paramètres suivants :

1. le chemin vers un dossier `solutions_1/` contenant des solutions ;
2. le chemin vers un dossier `solutions_2/` contenant des solutions ;

Le script recherche tous les couples de fichiers de solutions associés à une même instance :
— l'un des deux fichiers est situé dans le dossier `solutions_1/` ;
— l'autre fichier est situé dans le dossier `solutions_2/` ;
— les deux fichiers portent le même nom `name.txt`.
Si les deux fichiers sont différents, `name.txt` est affiché à l'écran.

À la fin, le programme affiche le nombre total de couples trouvés, le nombre de couples constitués de deux solutions différentes, avec leur proportion (en pourcentage).

6.2 Résultats de l'heuristique

Nous avons utilisé le script `call.sh` sur toutes les instances dont nous disposons, regroupées dans le dossier `files/instances/`. Dans le dossier `files/solutions/` fourni avec ce rapport, les solutions obtenues avec différentes valeurs du paramètre « nombre de mutations » pour toutes les instances sont fournies. Ces solutions sont regroupées en sous-dossiers de la manière suivante :

`S_(valeur du paramètre : nombre de mutations)_(lettre de a à d)`

— Si la valeur du paramètre « nombre de mutations » est vide, alors le script a été appelé sans préciser de valeur : pour chaque instance, le nombre de mutations par itération est égal au nombre de jobs dans l'instance.
— Les lettres servent à différencier plusieurs appels au script avec la même valeur pour le paramètre « nombre de mutations ».

Des fichiers textes de comparaison des solutions obtenues sont également disponibles dans le dossier `files/solutions/compare/`. Leur contenu est une copie de ce qu'affiche le script `compare.sh` (redirection de la sortie standard). Leur nom est de la forme suivante :

`compare_(nom dossier solutions 1)_(nom dossier solutions 2).txt`

6.2.1 Variabilité des solutions générées

L'heuristique implémentée est extrêmement instable : en appelant deux fois de suite le programme sur la même instance, les probabilités d'obtenir deux solutions différentes sont élevées. Ceci est dû au caractère aléatoire de l'exploration du voisinage de la solution mère dans l'algorithme de descente locale.

Pour plus de détails, veuillez consulter les données des fichiers de comparaison, pour une même valeur du paramètre « nombre de mutations » et des lettres différentes.

6.2.2 Influence du paramètre : nombre de mutations

Si on augmente la valeur du paramètre « nombre de mutations » :

1. Les chances d'obtenir des solutions proches d'une solution optimale sont de plus en plus grandes. (La valeur de la somme des retards diminue.)
2. Le temps d'exécution augmente :
   — raisonnablement pour des instances avec peu de jobs (jusqu'à 100 jobs, moins de 5 secondes environ) ;
   — déraisonnablement pour des instances avec beaucoup de jobs (au-delà de 150 jobs, 1 à 10 minutes environ).
3. Le phénomène d'instabilité est accru.

Pour plus de détails, veuillez consulter les données des fichiers de comparaison, pour des valeurs différentes du paramètre « nombre de mutations » (quelles que soient les lettres).

Conclusion

L'implémentation de l'heuristique a été réussie. Selon la valeur attribuée au nombre de mutations par itération, le temps d'exécution est plus ou moins acceptable. Du fait de l'exploration pseudo-aléatoire du voisinage d'une solution donnée, l'algorithme est extrêmement instable. Une piste envisageable pour limiter cette instabilité serait la mise en place d'un tabou : en maintenant à jour un historique (même partiel) de l'exploration effectuée, il deviendrait alors possible de s'extraire des minima locaux, qui ne sont généralement pas des solutions optimales (minima globaux).

7.1 Conclusion de Pierre-Antoine

Ce projet était la suite logique du mini-projet de Gestion des Flux et de la Production. J'ai apprécié le fait de pouvoir travailler sur une thématique très intéressante, de manière prolongée dans le temps, pour mieux appréhender le processus de réflexion associé à la recherche d'une solution heuristique (première approche simple, puis optimisation). Par ailleurs, les compétences d'Armand et les miennes se sont révélées être complémentaires, une fois encore.

7.2 Conclusion de Armand

Lors de ce projet, j'ai pu apprendre à utiliser de nouvelles méthodes de résolution en particulier l'utilisation de mutations pour une recherche de minimum local et voir le résultat sur de nombreuses instances. De plus travailler de nouveau avec Pierre-Antoine fut à nouveau une expérience enrichissante.

Heuristique pour un problème d'ordonnancement et de tournées

Département Informatique
4e année
2013 - 2014

Rapport - Projet de Sciences de la Décision

Résumé : Implémentation en langage C d'une heuristique pour un problème d'ordonnancement et de tournées. Le principe est de regrouper les jobs en groupes, pour lesquels l'algorithme de NEH fournit l'ordonnancement et la recherche du plus proche voisin fournit la tournée. Une descente locale est réalisée à partir d'une solution initiale.

Mots clefs : ordonnancement, tournée, heuristique, descente locale

Abstract: C language implementation of a heuristic for a scheduling and routing problem. The idea is to gather jobs in groups, for which the NEH algorithm provides the scheduling order and the research of the nearest neighbor provides the routing order. A local descent is made from an initial solution.

Keywords: scheduling, routing, heuristic, local descent

Encadrant
M. Jean-Charles BILLAUT
jean-charles.billaut@univ-tours.fr

Université François-Rabelais, Tours

Étudiants
Pierre-Antoine MORIN
pierre-antoine.morin@etu.univ-tours.fr
Armand RENAUDEAU
armand.renaudeau@etu.univ-tours.fr

DI4 2013 - 2014