Compte-rendu TP2

Aide à la décision – Job Shop



CHEVALIER Pierre – PIERREVAL Adrien

27/11/2015

Compte-rendu TP2

Aide à la décision – Job Shop

# Sommaire

[Sommaire 1](#_Toc436257807)

[1. Introduction 1](#_Toc436257808)

[2. La structure de données 1](#_Toc436257809)

[2.1. Diagramme UML 1](#_Toc436257810)

[2.2. Fichiers de données 1](#_Toc436257811)

[3. Algorithmes implémentés 1](#_Toc436257812)

[3.1. Algorithme Evaluer 1](#_Toc436257813)

[3.2. Recherche locale 1](#_Toc436257814)

[3.3. Algorithme génétique 1](#_Toc436257815)

[1. Structure de données utilisée 1](#_Toc436257816)

[2. Implémentation 1](#_Toc436257817)

[4. Résultats 1](#_Toc436257818)

[5. Conclusion 1](#_Toc436257819)

# Introduction

Le principe de ce TP d'Aide à la Décision consistait à mettre en œuvre un algorithme de recherche locale et de recherche génétique afin de trouver les meilleures manières de concevoir une série de pièces sur un nombre de machines données. Pour ce faire, nous nous sommes appuyés sur des fichiers de données fournis, qui nous ont permis d'effectuer différents tests.

Le Job Shop c'est …

Une structure de données nous avait été fournie au démarrage du TP, permettant de lire les fichiers de données et d'initialiser les structures de données adéquates à notre travail. De ce fait, nous avons commencé avec la procédure evaluer(), qui génère un chemin critique à l'aide d'un vecteur de Bierwirth. Puis, nous avons implémenté l'algorithme de recherche locale pour améliorer nos résultats et enfin l'algorithme génétique.

Dans l'intro dire qu'on va parler d'assemblage de pièces sur des machines, mais que ça pourrait aussi être l'ordonnancement de tâches sur un processeur ou que sais-je encore.

# La structure de données

## Diagramme UML

Comme le montre ce diagramme, nous avons utilisé pour ce TP 4 classes :

* La classe Jobs : chaque instanciation de cette classe correspond à une opération à effectuer sur une pièce sur une machine donnée. Sont donc des données membres de l'objet la durée, la machine sur laquelle effectuer l'opération ainsi que la pièce sur laquelle effectuer l'opération. On notera également la présence de pointeurs qui permettent de placer des instanciations de cette classe dans des listes chaînées.
* La classe Data : cette classe permet le stockage de toutes les opérations lues dans un fichier de données. Le membre principal de cette classe est celui contenant toutes les opérations pour la fabrication de toutes les pièces. Data contient également les membres makespan, last\_op, destinés à contenir les résultats des algorithmes implémentés.
* La classe Bierwirth : expliquée plus en détail dans la prochaine section, elle sert principalement à permettre l'exécution des algorithmes d'évaluation et de recherche locale.
* La classe Population : détaillée également dans la section suivante, son rôle est de permettre l'exécution de l'algorithme génétique.

## Fichiers de données

La structure de données utilisée nous a été fournie avec ses constructeurs et. Initialisée à partir des fichiers de données, elle contenait toutes les informations nécessaires pour traiter notre problème.

Pour l’algorithme génétique nous avons utilisé un vector pour stocker nos individus.

# Algorithmes implémentés

## Algorithme Evaluer

Pour pouvoir utiliser la procédure evaluer(), nous avons besoin de ce qui s'appelle un **vecteur de Bierwirth**. Ce vecteur de Bierwirth permet de mettre dans un seul et même vecteur toutes les opérations contenues dans la matrice d'opérations de la structure de données data. Pour cela, nous effectuons un simple balayage sur la matrice afin d'en mettre toutes les composantes dans le vecteur et une fois cela fait, nous appliquons au vecteur la fonction shuffle() de la librairie standard afin de mélanger aléatoirement le vecteur et obtenir un vecteur de Bierwirth.

Une fois ce vecteur de Bierwirth constitué, notre programme va effectuer chaque opération dans l'ordre dans laquelle il les trouve, reliant ainsi entre elles et dans l'ordre les opérations s'effectuant sur une même machine, ou encore les opérations s'effectuant sur une même pièce, ce qui nous permettra de retracer l'ordre des opérations qui ont eu lieu pour une pièce ou une machine donnée.

De même, pour chaque nouvelle opération lue dans le vecteur de Bierwirth, la procédure met à jour le temps total, et surtout évalue quelle opération effectuée précédemment ralenti le processus global. Cette vérification nous permet de constituer le **chemin critique**, résultat final de la procédure evaluer(), qui donne, une fois l'algorithme terminé, les opérations dont dépend réellement le temps final d'assemblages de toutes les pièces, appelé **makespan**.

## Recherche locale

Pour améliorer les résultats obtenus avec la fonction évaluer à partir d’un vecteur de Bierwirth aléatoire, nous procédons à une recherche locale, c’est-à-dire le parcours de notre chemin critique et l’échange, s’il est possible et s’il nous donne un meilleur résultat, de deux membres au sein du vecteur de Bierwirth autant de fois que nécessaire.

Dans un premier temps nous recherchons donc les arcs disjonctifs qui sont des arcs (i, j) caractérisés par le fait i et j se trouve sur la même machine. De ce fait, il est donc possible d’échanger ces deux points dans le vecteur de Bierwirth. On évalue donc un nouveau makespan avec ce nouveau vecteur de Bierwirth et on le compare à l’ancien makespan.

S’il est plus avantageux de conserver notre l’ancien makespan, on oublie notre échange de valeur, et on continu notre parcours du chemin critique avec notre ancien chemin critique. Si au contraire le nouveau Bierwirth donne un meilleur résultat, on le conserve, on remplace nos anciens résultats par les nouveaux, et on recommence l’algorithme depuis le début en reprenant au début de notre nouveau chemin critique.

Notre algorithme s’arrête quand le chemin critique a été parcouru sans aucunes améliorations possibles.

Ainsi nous avons, à partir d’un résultat dû au hasard (méthode evaluer()), amélioré le plus possible le chemin critique et le makespan calculé.

## Algorithme génétique

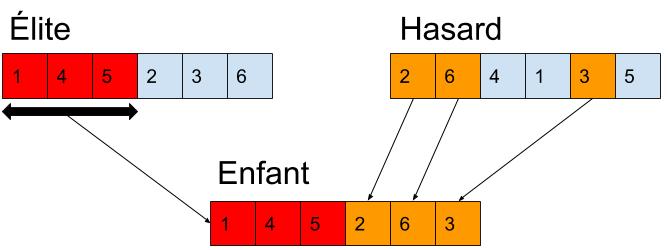
Afin d’obtenir un résultat toujours plus optimal nous avons implémenté un algorithme génétique sur nos résultats dans les fichiers Population.cpp et Population.h. La classe Population est composée d’une vecteur (une population de par défaut 100 individus) de vecteur de Bierwirth avec leurs résultats associés (chemin critique et makespan).

Elle se construit à l’aide d’un premier vecteur de Bierwirth (fait avec le constructeur classique) qui est mélangé avec la fonction shuffle() n fois pour crever les n individus. On appelle ensuite les méthodes evaluer() et recherche\_locale() sur ceux-ci et on crée ainsi notre population. S’en suit un tri des individus dans le vecteur afin de repérer lesquels possèdent les meilleurs makespan. Ainsi dans un premier temps nous avons déjà un « bon » résultat de makespan, celui du meilleur individu de la population.

Pour améliorer nos résultats, on va créer une nouvelle population de même taille que la première en s’inspirant des meilleurs individus de celle-ci. Notre algorithme pour créer la population fille est le suivant :

Les individus considérer comme d’élite sont les premiers 20% de la population.  
 Nous utilisons un « gène » de cette population pour créer la population enfant : les 50% premiers membres de leur vecteur de Bierwirth.  
 On tire au hasard un des individus d’élite (à l’aide d’un Mersenne Twister de seed « bierwirth », et on prend le gène de cet individu. Celui-ci sera la base de notre nouvel individu.

On tire au hasard dans l’ensemble de l’ancienne population un autre individu. Celui-ci va nous permettre de compléter notre nouvel individu à l’aide d’une Union. Ainsi les 50% membre du nouveau vecteur de Bierwirth sont un gène préexistant et le 50% autres sont dû au hasard.



Une fois a population fille terminer on fusionne les deux populations (car rien ne nous dis que la population fille n’aura pas de mauvais résultats), on trie et on garde les n meilleurs membres de la nouvelle population.

On peut exécuter cet algorithme plus de fois mais le temps d’exécution est assez lent.

### Structure de données utilisée

#### Membres :

* bierwirth\_vector\_ : Vecteur de pointeurs sur les opérations.
* d\_ : Copie de la structure de données.
* last\_cp\_ : Pointeur sur la dernière opération effectuée.
* makespan\_ : Temps de réalisation le plus important.
* tabItem\_ : Séquences et temps selon les pièces.
* tabOpe\_ : Séquence et temps selon les machines.

#### Méthodes :

* afficher\_chemin\_critique : Utilisé après évaluation, permet d'afficher les opérations du chemin critique
* afficher\_sequences : Affiche sur la console l'ordre de passage des différentes pièces sur chaque machine.
* Bierwirth : Constructeur, peut être initialisé en donnant en référence des données, un vecteur d'entiers correspondant aux IDs des opérations ou encore par copie d'un autre vecteur.
* display : Affiche dans l'ordre les opérations contenues dans le vecteur de Bierwirth.
* evaluer : Evalue le vecteur de Bierwirth en mettant à jour les différents pointeurs et temps pour pouvoir afficher le chemin critique ainsi que les séquences.

### Implémentation

# Résultats

# Conclusion

On s’est bien amusé !