RÉPUBLIQUE DU BÉNIN
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ D'ABOMEY-CALAVI
INSTITUT DE FORMATION ET DE RECHERCHE EN INFORMATIQUE
BP 526 Cotonou Tel : +229 21 14 19 88
http://www.ifri-uac.net Courriel : contact@ifri.uac.bj

Mémoire pour l'obtention du Master en Informatique

Résolution du *Pigment Sequencing Problem* avec les algorithmes génétiques

Tafsir GNA
gnatafsir@gmail.com

Sous la supervision de :
Dr Ing. Vinasétan Ratheil HOUNDJI
&
Professeur Mahouton Norbert HOUNKONNOU

Année Académique : **2016-2017**

Table des figures

Liste des tableaux

Sommaire

Table des matières

Remerciements

Je tiens à remercier tous ceux qui ont aidé et participé à la réalisation de ce travail à travers leurs différents apports et soutiens.

Je remercie particulièrement :

- Pr. Eugène EZIN, Directeur de l'Institut de Formation et de Recherche en Informatique (IFRI) ainsi que tous les membres du corps enseignant et administratif de l'IFRI ;
- Pr. Mahouton Norbert HOUNKONNOU, pour avoir accepté de superviser mes travaux ainsi que Dr Ing. Vinasétan Ratheil HOUNDJI pour l'encadrement et les conseils apportés ;
- Mon père, ma mère et par extension toute ma famille et mes proches pour leur soutien et leurs encouragements.

Liste des Algorithmes

Liste des sigles et abréviations

AG : Algorithmes Génétiques

ATSP : Asymmetric Traveling Salesman Problem

C-PGA : Coarse-grained Parallel Genetic Algorithm

CLSP : Capacited Lot Sizing Problem

CP : Constraint Programming

DLSP : Discrete Lot Sizing Problem

ELSP : Economic Lot Scheduling Problem

EOQ : Economic Order Quantity

F-PGA : Fine-grained Parallel Genetic Algorithm

GA : Genetic Algorithm

HCM-PGA : Hierarchical Coarse-grained and Master-slave Parallel Genetic Algorithm

HFC-PGA : Hierarchical Fine-grained and Coarse-grained Parallel Genetic Algorithm

M-PGA : Master-slave Parallel Genetic Algorithms

MIP : Mixed Integer Programming

PSP : Pigment Sequencing Problem

SA : Simulated Annealing

WW : Wagner-Within

Résumé.

Le dimensionnement de lots tient une place importante en planification de production en industrie. Il consiste à trouver un plan de production qui à la fois satisfait les demandes et prend en compte les objectifs financiers de l'entreprise. De récentes recherches ont testé une variante NP-Difficile du problème de dimensionnement de lots : Le *Pigment Sequencing Problem* (PSP). Différentes méthodes de résolution de problèmes ont ainsi été appliquées au PSP. Aucune des méthodes appliquées n'est basée sur les algorithmes génétiques qui ont pourtant montré leur efficacité sur nombre de problèmes d'optimisation. Dans ce travail, nous appliquons deux méthodes de résolution de problèmes basées sur les algorithmes génétiques au PSP. Il s'agit des algorithmes génétiques parallèles et hiérarchiques. Les tests effectués nous ont permis de comparer cette dernière méthode de résolution à celles déjà appliquées dans de précédentes recherches. Ces premiers résultats montrent que les algorithmes génétiques pourraient être adaptés à la résolution du PSP.

Mots clés : *Algorithme génétique, planification de production, pigment sequencing problem, dimensionnement de lots*.

Abstract.

Lot sizing takes an important place in production planning in industry. It consists in determining a production plan that at the same time meets the orders and takes into account the financial objectives of the enterprise. Recent researches have experimented an NP-Hard variant of lot sizing problem: the *Pigment Sequencing Problem* (PSP). Several methods have been tested on PSP. None of the tested methods is based on genetic algorithms whereas they showed their efficiency in solving optimization problems. In this document, we apply two solving methods based on genetic algorithms on PSP which are the hierarchical and parallel genetic algorithms. The experiments allow us to compare this last solving method to the ones applied in ealier researches. This very first results show that genetic algorithms could be fit in solving PSP.

Keys words: *Genetic algorithm, production planning, pigment sequencing problem, lot sizing*.

Introduction

Dans un processus de planification de production, le problème de dimensionnement de lots (lot sizing) consiste à identifier les articles à produire, quand il faut les produire et sur quelle machine de façon à satisfaire les demandes tout en considérant les objectifs financiers. Connu dans la littérature sous le nom de problème de *lot sizing*, il a été beaucoup étudié ces dernières décennies [38]. En effet, la résolution du problème de dimensionnement de lots se heurte à des difficultés. Ainsi, une ressource de production n'est le plus souvent pas seulement dédiée à un unique article, mais plutôt utilisée pour produire différents types d'articles. Aussi, un plan de production doit remplir plusieurs objectifs parfois contradictoires, notamment, garantir un excellent niveau du service-client et minimiser la production et les coûts de stockage.

Différentes versions du problème de dimensionnement de lots ont été proposées dans la littérature, chacune étant spécifique à leur domaine d'application. Récemment, Houndji et al. [24] et Ceschia et al. [7] ont expérimenté une variante NP-Difficile [1] du problème de dimensionnement de lots. Cette version est connue sous le nom de *Pigment Sequencing Problem* (PSP) (Pochet et Wolsey [34]) et a été récemment inclue à la bibliothèque CSPlib (Gent and Walsh, [15]) . Il s'agit de produire plusieurs articles avec une seule machine dont la capacité de production est limitée à un article par période. L'horizon de planification est discret et fini, et il y a des coûts de stockage et des coûts de transition d'une production à une autre. Par ailleurs, les demandes sont normalisées et donc binaires.

Le PSP comme tout problème de dimensionnement de lots ou d'optimisation peut être formalisé et resolu à l'aide des algorithmes génétiques. Les algorithmes génétiques sont des méthodes de recherche heuristiques inspirées de l'évolution naturelle des espèces vivantes. En se basant sur le principe du "mieux adapté", ils permettent sur différentes générations de déterminer la meilleure solution à un problème. Différentes recherches[19] [32] ont montré que l'utilisation des algorithmes génétiques pour des problèmes d'optimisation semble raisonnable dans le cas de grands problèmes où trouver la solution avec d'autres algorithmes reste encore

[1] En théorie de la complexité, un problème NP-Difficile est un problème au moins aussi difficile que tous les autres problèmes de la classe NP (problème de complexité polynomiale dont on peut vérifier "rapidement" si une solution candidate est bien solution)[25].

problématique en temps.

Contribution

Le *Pigment Sequencing Problem* (PSP) est un problème NP-Difficile d'optimisation combinatoire pour lequel les instances de taille moyenne (de l'ordre de 100 périodes) peuvent être efficacement résolues en utilisant une formulation appropriée de programmation en nombre mixte [34]. Cependant, à notre connaissance, aucun modèle basé sur les algorithmes génétiques n'a encore été proposé pour ce problème. Nous proposons deux méthodes de recherche basées sur les algorithmes génétiques pour ce problème. La première est une méthode appelée *Hierarchical Coarsed-grained and Master-slave Parallel Genetic Algorithm* (HCM-PGA) qui divise la population globale en sous-populations et qui confie l'évaluation des différents chromosomes à des processus fils. La seconde, appelée *Hierarchical Fine-grained and Coarse-grained Parallel Genetic Algorithm* (HFC-PGA) qui réduit le champ de croisement à l'environnement immédiat tout en échangeant des individus avec les processus voisins. Les tests que nous avons menés afin de valider nos deux méthodes montrent que les algorithmes génétiques pourraient être efficaces pour résoudre le PSP.

Organisation du travail

Le travail effectué est organisé dans ce document en 3 chapitres. Le premier chapitre présente une revue de littérature des problèmes de dimensionnement de lots en planification de production en nous concentrant sur les classes de problèmes qui nous concernent dans notre étude. Ensuite, nous présentons le PSP, le décrivons et présentons les modèles et méthodes utilisés dans sa résolution. Nous présentons également dans ce chapitre les algorithmes génétiques, leurs étapes ainsi que leur fonctionnement. Dans le deuxième chapitre, nous détaillons les deux méthodes de recherche utilisées ainsi que les algorithmes associés utilisés. Ensuite, dans le dernier et troisième chapitre, nous expérimentons nos deux méthodes, présentons les résultats obtenus et comparons ces résultats à l'état de l'art en la matière. Pour finir, nous tirons une conclusion du travail effectué et dressons les perspectives possibles en vue d'améliorer ce qui a été fait.

Chapitre 1

État de l'art

Résumé. Le dimensionnement de lots est un problème classique en planification de production. Différents critères [38] permettent de classifier les problèmes de dimensionnement de lots. Suivant les classifications proposées, différentes classes peuvent être identifiées au nombre desquelles on peut citer le *Discrete Lot Sizing Problem* (DLSP) dont le PSP fait partie. Différentes méthodes et modèles ont ainsi été appliqués au PSP [22] [8] avec différents résultats.

Introduction

Dans ce chapitre, nous présentons la revue de littérature des problèmes de dimensionnement de lots en planification de production dont fait partie le PSP. Nous exposons les méthodes de recherche déjà appliquées dans la résolution du PSP et soulignons les travaux effectués sur les algorithmes génétiques. Nous montrons également en quoi les algorithmes génétiques pourraient être particulièrement adaptés à ce genre de problèmes.

1.1 Le dimensionnement de lots en planification de production

La planification de production est un processus qui consiste à déterminer un plan qui indique quelle quantité d'articles produire durant un intervalle de temps appelé "horizon de planification". La planification de production occupe donc une place stratégique en industrie aussi bien en terme de performance que de coûts d'exploitation.

Différentes recherches [3] ont été menées dans le but de classifier les problèmes de dimensionnement de lots. Plusieurs critères ont pu être identifiés. Nous présentons dans cette section

quelques critères de classification [38] des problèmes de dimensionnement de lots.

1.1.1 Critères de classification

Différents critères interviennent dans la classification des problèmes de dimensionnement de lots, notamment :

***L'échelle de temps* :** L'horizon de planification peut être soit discret ou continu. Lorsqu'il est discret, on distingue alors les problèmes de petite taille ou *Small time buckets*, les problèmes de grande taille ou *Big time Buckets* et enfin les problèmes de très grande taille ou *Very big time buckets*.

***Le nombre de niveaux* :** Suivant qu'une relation ou non soit définie entre les articles, on distingue les problèmes à un niveau et les problèmes à plusieurs niveaux. Dans les problèmes à plusieurs niveaux, la production d'un article est conditionnée par celle d'un autre article.

***Le nombre de produits* :** Un problème peut être qualifié soit de *multi-item* (plusieurs types d'articles) ou de *single-item* (un unique type d'articles). Il est *multi-item* lorsque plusieurs types d'articles sont produits et *single-item* dans le cas d'un unique type d' articles produit.

***Les contraintes de capacité* :** Les contraintes de capacité sont diverses. Elles peuvent concerner aussi bien la capacité de stockage, la capacité des machines, etc. La prise en compte de ces contraintes de capacité rend le problème plus complexe à résoudre.

***Les demandes* :** Les demandes peuvent être aussi bien :

- constantes avec des valeurs qui ne changent pas sur l'horizon de temps ou dynamiques avec des valeurs qui sont fonction du temps.
- certaines avec des valeurs connues à l'avance ou stochastiques avec des valeurs basées sur des probabilités.
- indépendantes sans lien entre les articles à produire ou dépendantes en cas de relation entre articles.

***Les coûts et temps de lancement ou préparation (setup)* :** Ils sont induits en cas de changement de types d'articles produits au cours d'un horizon de planification. On parle alors de transition.

Nous avons choisi dans cette étude de décrire les problèmes de dimensionnement de lots suivant l'échelle de temps. Nous allons donc nous intéresser aux deux premiers c'est à dire les problèmes de petite et grande taille.

1.1.2 Classes de problèmes de dimensionnement de lots

1.1.2.1 Problèmes de petite taille ou à courtes périodes

Ces problèmes (*Small time bucket problems*) sont caractérisés par des périodes de l'ordre de quelques heures, et la séquence des lots lors de la production est prise en compte. Quatre types de problèmes sont étudiés dans la littérature. Ces différents types de problèmes sont donc :

Discrete lot-sizing and scheduling problem (DLSP) : Dans ce problème [12], au plus un article peut être produit par période. Dans le cas où un article est produit sur une période, alors toute la capacité disponible sur cette période sera utilisée. Les coûts de lancement sont en général pris en compte seulement lorsqu'un nouveau lot commence et non à chaque période.

Continuous setup lot-sizing problem (CSLP) : Le CSLP reprend les mêmes principes que le problème précédent à la seule différence que lorsqu'un article est produit, ce n'est plus toute la capacité qui est utilisée mais uniquement la capacité nécessaire à la production.

Proportional lot-sizing and scheduling problem (PLSP) : Dans ce problème, la capacité restante pour une période donnée est réutilisée pour produire un second article. Le cas à plusieurs niveaux et plusieurs machines représente par exemple, une extension de ce problème.

General lot-sizing and scheduling problem (GLSP) : Dans ce problème [13], Aucune restriction n'est appliquée sur le nombre d'articles à produire par période. En intégrant des temps de lancement de production dépendants de la séquence des lots à produire, on obtient des extensions du GLSP [14].

1.1.2.2 Problèmes de grande taille ou à longues périodes

Dans les problèmes de grande taille (*Big time bucket problems*), l'horizon va de quelques jours à quelques semaines. De même, un produit peut être fabriqué par période. Nous proposons dans cette section, une étude concise de ces problèmes.

Les problèmes à un niveau et un produit :

Manne [30] et Wagner et Whitin [37] ont été à l'initiative sur l'étude de cette classe de problèmes. Ils ont permis la mise au point d'une méthode de résolution utilisée afin de résoudre des problèmes plus complexes. En effet, la résolution des problèmes à plusieurs niveaux bénéficie grandement des méthodes mises en œuvre dans la résolution des problèmes à un niveau et un produit.

Les problèmes à un niveau et plusieurs produits :

Ces problèmes introduisent des contraintes de capacité qui s'accompagnent d'une relation entre les différents articles produits, rendant ainsi le problème plus difficile à résoudre.

Les problèmes à plusieurs niveaux :

Une des principales caractéristiques de ces problèmes est le fait que les produits intermédiaires entrent dans la production des produits finis. Ainsi, les demandes entre les produits ou demandes dépendantes et les demandes indépendantes que sont demandes arrivant de l'extérieur sont prises en compte dans ce type de problème.

Un exemple d'un problème de "*Big time bucket*" est le cas où différents articles peuvent être produits sur une même ressource en une seule période. Ce problème est connu sous le nom de "Capacited Lot Sizing Problem" (CLSP). Le CLSP consiste à déterminer le coût et le temps de production des articles dans l'horizon de planification. La figure 1.1 présente un exemple de classification des modèles de dimensionnement de lots.

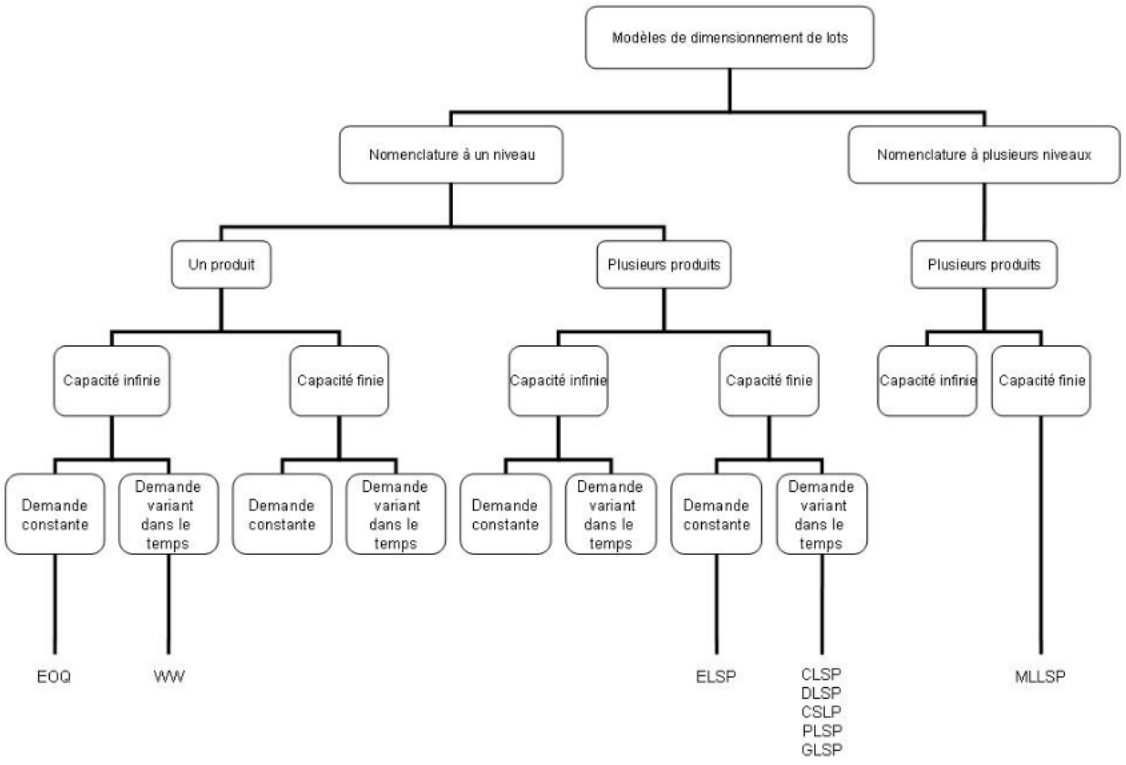

FIGURE 1.1 – Exemple de classification des modèles de dimensionnement de lots [38]

1.2 Le *Pigment Sequencing Problem* (PSP)

1.2.1 Revue de littérature

Le PSP appartient à la catégorie des DLSP. En effet, il s'agit d'un problème où toute la capacité disponible sur cette période sera utilisée si un article est fabriqué. Nous présentons dans cette section une brève revue de littérature des différents travaux effectués autour du PSP.

Miller et Wolsey [31] ont formulé le DLSP avec des coûts de préparation indépendants de séquence comme un problème de réseau à flot. Ils ont présenté des formulations MIP pour les différentes extensions (avec *backlogging*, avec stock de sûreté, avec stock initial). Plusieurs variantes et formulations MIP du DLSP ont été proposées et discutées par Pochet et Wolsey [34].

Gicquel et al. [16] présentent une formulation et dérivent des inégalités valides pour le DLSP avec plusieurs articles et des coûts et temps de préparation séquentiels ; laquelle est une extension du problème proposé par Wolsey [39]. Dans Gicquel et al. [17], les auteurs ont proposé une nouvelle manière de modéliser le DLSP avec plusieurs articles et des coûts et temps de préparation séquentiels ; qui exploite le fait que les attributs physiques pertinents des articles, tels que la couleur, la dimension, le niveau de qualité. Cela leur a permis de réduire significativement le nombre des variables et des contraintes dans les modèles MIP.

Houndji et al. [24] ont introduit une nouvelle contrainte globale, qu'ils ont appelée *StockingCost* afin d'efficacement résoudre le PSP en programmation par contrainte. Les auteurs l'ont alors testée sur de nouvelles instances et les ont publiées sur CSPlib (Gent et Walsh [15]). Les résultats expérimentaux ont montré que le *StockingCost* est plus efficace en filtrage par rapport aux autres contraintes généralement utilisées dans la communauté de la programmation par contrainte.

Plus récemment, Ceschia et al. [8] ont appliqué le recuit simulé sur le PSP. Ils ont introduit une procédure de recuit simulé afin de guider la recherche locale, qu'ils ont ensuite appliquée sur de nouvelles instances disponibles sur le bibliothèque Opthub [7].

1.2.2 Description du problème

Des différentes études [22] [8] déjà effectuées sur le PSP, nous pouvons le décrire comme un problème qui consiste à trouver un plan de production de plusieurs articles à partir d'une machine avec des coûts de transition. Les coûts de transition sont les coûts encourus lors du passage de la production de l'article i à celui de l'article j avec $i \neq j$. Le plan de production doit satisfaire les demandes des clients tout en :

- respectant la capacité de production de la machine ;
- minimisant les coûts de stockage et de transition.

On suppose que la période de production est suffisamment courte pour ne produire qu'au plus un article par période et que les demandes sont normalisées : la capacité de production de la machine est limitée à un article par période et d(i, t) $\in$ 0, 1 avec i l'article et t la période.

Il s'agit d'un problème de planification de production ayant les caractéristiques suivantes : un horizon de planification discret et fini ; des contraintes de capacité ; une demande statique et déterministe ; multi-item et small bucket, des coûts de transition ; un seul niveau ; sans shortage [1].

Illustration : Soit un problème avec les données ci-dessous :

- Nombre d'articles : $NI = 2$;
- Nombre de périodes : $NT = 5$;
- Demande par période. Soit d(i, t) la demande de l'article i à la période t : $d(1, t) = (0, 1, 0, 0, 1)$ et $d(2, t) = (1, 0, 0, 0, 1)$;
- Coût de stockage. Soit $h(i)$ le coût de stockage de l'article i : $h(1) = h(2) = 2$.

Soit *xT* le plan de production qui représente une solution potentielle du problème. Il s'agit d'un tableau de dimension *NT* , contenant à son indice t (avec $t \in 1...NT$) l'article i à produire. Une solution admissible du problème est : $xT = (2, 1, 2, 0, 1)$ avec un coût de $q(2, 1)+q(1, 2)+q(2, 1)+2*h(2) = 15$. La solution optimale est : $xT = (2, 1, 0, 1, 2)$ avec un coût de $q(2, 1)+q(1, 2)+h(1) = 10$.

1.2.3 Modèles et formulations

Différents modèles ont été proposés afin de représenter, modéliser et résoudre le PSP. Il s'agit des modèles de programmation mixte en nombres entiers ou MIP (au nombre de 3) proposés par Pochet et Wolsey [34], considérés comme l'état de l'art des méthodes de résolution exactes sur les problèmes de dimensionnement de lots en particulier celui du PSP, du modèle CP et du modèle SA. Nous présentons donc ici le premier modèle MIP dont les deux derniers sont une reformulation, le modèle CP ainsi que le modèle SA.

[1] On parle de shortage lorsque c'est possible de ne pas satisfaire toutes les demandes [22]

1.2.3.1 Modèle MIP 1

Le modèle MIP 1 [34] tel qu'exposé par Pochet et Wolsey se présente comme suit :

$$min \sum_{i,j,t} q^{i,j}\chi_t^{i,j} + \sum_{i,t} h^i s_t^i \tag{1.1}$$

$$s_0^i = 0, \forall i \tag{1.2}$$

$$x_t^i + s_{t-1}^i = d_t^i + s_t^i, \forall i, t \tag{1.3}$$

$$x_t^i \leq y_t^i, \forall i, t \tag{1.4}$$

$$\sum_i y_t^i = 1, \forall t \tag{1.5}$$

$$\chi_t^{i,j} = y_{t-1}^i + y_t^j - 1, \forall i, j, t \tag{1.6}$$

$$x, y, \chi \in \{0,1\}, s \in \mathbb{N}, i \in \{0..NI\}, t \in \{1..NT\} \tag{1.7}$$

avec les variables de décisions suivantes :

- x_t^i : variable binaire de production qui vaut 1 si l'article i est produit à la période t et 0 sinon ;
- y_t^i : variable binaire de setup qui vaut 1 si la machine est préparée pour la production de l'article i et 0 sinon ;
- s_t^i : variable entière de stockage qui contient le nombre d'articles i stockés à la période t ;
- $\chi_t^{i,j}$: variable binaire de transition qui vaut 1 si à la période t, on est passé de la production de l'article i à l'article j et 0 sinon.

L'objectif est de minimiser la somme des coûts de stockage et des coûts de transition. Il est exprimé par la contrainte (1.1). La contrainte (1.2) rappelle qu'il n'y a pas de stock initial. La contrainte (1.3) exprime la règle de la conservation de flot. La contrainte (1.4) vise à forcer la variable de setup y_t^i à prendre la valeur 1 s'il y a production de l'article i à la période t. La contrainte (1.5) s'assure qu'il y a toujours un article qui est préparé. En accord avec la fonction objectif, y_t^i va prendre la valeur qui minimise le coût de transition. En général s'il n'y a pas de production à la période t, $y_t^i = y_{t-1}^i$ ou $y_t^i = y_{t+1}^i$ mais parfois il peut être intéressant de préparer la machine pour un article intermédiaire sans le produire. La contrainte (1.6) assigne les valeurs aux variables de transition. En effet, si y_{t-1}^i et y_t^i valent 1 alors $\chi_t^{i,j}$ est obligé de prendre la valeur 1 et sinon $\chi_t^{i,j}$ serait égal à 0 grâce à la fonction objectif qui minimise le coût de transition.

1.2.3.2 Modèle CP

Dans ce modèle [22], l'objectif est d'attribuer à chacune des demandes, une période qui respecte la date limite de satisfaction de la demande. On note $date(p) \in [1, ..., T],\ \forall p \in [1, ..., n]$ la période dans laquelle la demande p a été satisfaite, $dueDate(p) \in [1, ..., T]$ la date limite de la demande p et $I(p)$ l'article correspondant à la demande p. Afin de s'assurer de la faisabilité de la solution, les principales contraintes sont les suivantes :

$$date(p) \leq duedate(p), \forall p \tag{1.8}$$

$$alldifferent(date) \tag{1.9}$$

dans lesquelles :

- Equation 1.8 : chaque demande doit être satisfaite avant sa date limite ;
- Equation 1.9 : Puisque la capacité d'une machine est limitée à 1, chaque demande doit être satisfaite à différente période.

Il est possible de modéliser la partie des coûts de transition comme un ATSP, chaque demande représente une ville qui doit être visitée et les coûts de transition sont les distances entre deux villes. Ainsi, on ajoute une demande artificielle $n + 1$ de sorte que $date(n + 1) = T + 1$ avec $q^{I(p),I(n+1)} = q^{I(n+1),I(p)} = 0, \forall p \in [1, ..., n]$. On note alors $successor(p), \forall p \in [1, ..., n]$, la demande satisfaite juste après la satisfaction de la demande p. Les contraintes suivantes additionnelles peuvent être alors ajoutées :

$$circuit(successor) \tag{1.10}$$

$$date(p) \leq date(successor(p)), \forall p \in [1, ..., n] \tag{1.11}$$

dans lesquelles :

- Equation 1.10 : il garantit l'existence d'un circuit hamiltonien ;
- Equation 1.11 : la demande p doit être satisfaite avant ses successeurs.

Enfin, l'objectif est de minimiser les coûts de stockage et les coûts de transition.

$$\sum_{p} (dueDate(p) - date(p)) * h_{I(p)} + \sum_{p} q^{I(p),I(successor(p))}$$

1.2.3.3 Modèle SA

Dans ce modèle [8], la procédure de recuit simulé conduit à chaque itération à une action aléatoire en utilisant une distribution uniforme. En recuit simulé, l'action est toujours acceptée

si elle est facteur d'amélioration. Dans le cas contraire, elle est acceptée, si elle est basée sur une distribution en temps croissant de façon exponentielle. Dans le détail, une mauvaise action est acceptée avec une probabilité de $e^{-\Delta/T}$ où Δ est la différence du coût total induit par l'action et T est la température. La température commence à une valeur T_0 et est multipliée par α (avec $0 < \alpha < 1$), après un nombre fixé d'échantillons n_s, selon la procédure de refroidissement géométrique standard du recuit simulé.

Dans le but d'accélérer les premières étapes de la procédure de recuit simulé, Ceschia et al.[8] ont utilisé un mécanisme de *cut-off* (Johnson et al. [28]). Ils ont ajouté un nouveau paramètre n_a représentant le nombre maximal d'actions acceptées à chaque niveau de température. Ainsi, la température diminue lorsque la première des deux conditions suivantes est remplie : (i) le nombre d'actions échantillonnées atteint n_s, (ii) le nombre d'actions acceptées atteint n_a.

Le critère de terminaison est basé sur le nombre total d'itérations I_{max}, plutôt que sur un seuil de température. De cette manière, le temps d'exécution est approximativement le même pour toutes les configurations des paramètres.

1.3 Les algorithmes génétiques

Les algorithmes génétiques sont des algorithmes qui tirent leur origine de l'analogie avec l'évolution naturelle des espèces vivantes et des mécanismes de reproduction. Ils ont été proposés pour la première fois par John Holland [21] en 1970. Un des principes fondamentaux des algorithmes génétiques est celui de la survie du "mieux adapté". Selon ce principe, un individu avec des caractéristiques en adéquation avec le milieu dans lequel il évolue à beaucoup plus de chances de survivre. Au même titre que l'évolution naturelle, les algorithmes génétiques simulent l'échange aléatoire de matériel génétique à travers les différents individus de la population. Dans ses travaux, Holland développa un formalisme permettant de prédire la qualité de la prochaine génération. Ce théorème est plus connu comme le théorème de schémas.

1.3.1 Concepts de base

Le processus de recherche au niveau des algorithmes génétiques se déroule suivant deux phases : l'exploration et l'exploitation. L'exploration consiste à parcourir l'espace de recherche à la quête de solutions optimales au problème à résoudre. Afin de guider l'exploration intelligente de l'espace de recherche, les algorithmes génétiques procèdent par choix aléatoires dans le parcours de cet espace. L'objectif global des algorithmes génétiques est d'optimiser une fonction coût connue comme fonction de "fitness". Différents termes empruntés de la génétique naturelle sont employés au niveau des algorithmes génétiques pour nommer des processus et objets. Ainsi, les algorithmes génétiques travaillent sur une population qui est un ensemble d'individus ou de chromosomes. Chaque chromosome possède une représentation codée ap-

pelée génotype. Le génotype est un ensemble de gènes disposés en une succession linéaire et un allèle est une des valeurs que peut prendre un gène. La fonction de fitness mesure l'adaptation d'un individu à l'environnement dans lequel il évolue. Plus un individu est adapté à son environnement, plus ses chances de survie sont élevées.

1.3.2 Fonctionnement

L'algorithme 1 présente le principe de fonctionnement de l'algorithme génétique simple.

Algorithme 1 : Algorithme génétique standard [19]

Générer la population initiale P_i
Évaluer la population P_i
tant que *le critère de terminaison n'est pas satisfait* **faire**
  Sélectionner les éléments de P_i à copier dans P_{i+1}
  Appliquer le croisement aux éléments de P_i et les mettre dans P_{i+1}
  Appliquer la mutation aux éléments de P_i et les mettre dans P_{i+1}
  Évaluer la nouvelle population P_{i+1}
  $P_i = P_{i+1}$
fin

Cet algorithme 1 est explicité plus en détails à l'aide de la figure 1.2.

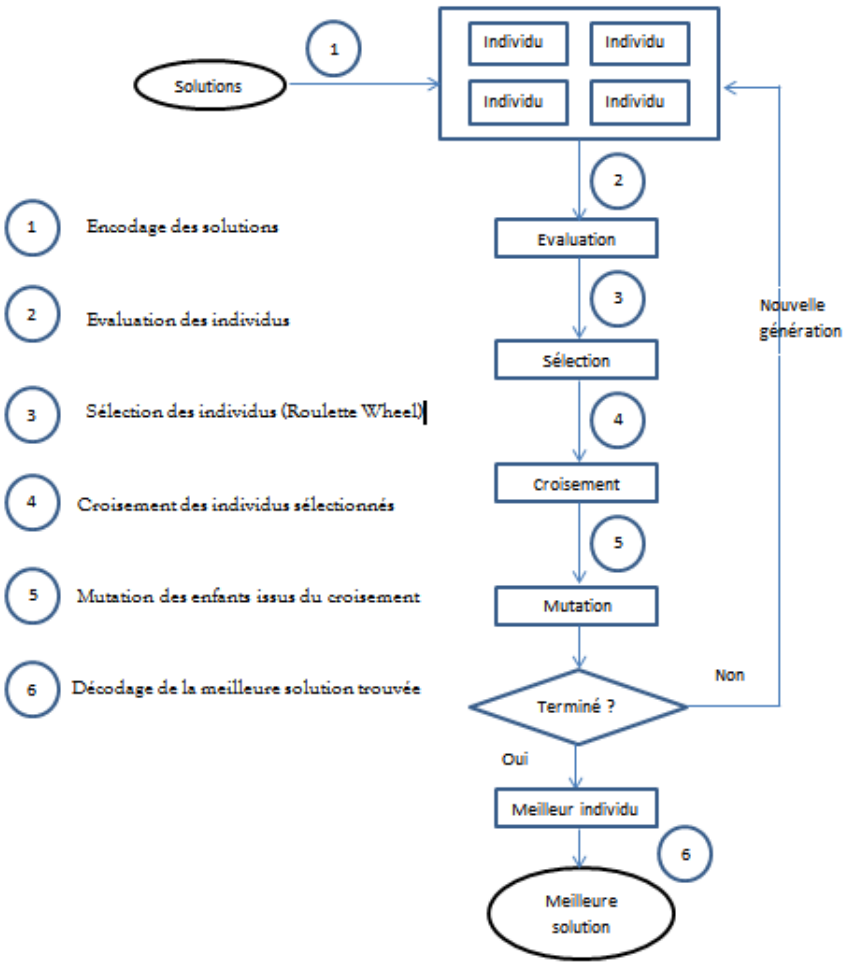

FIGURE 1.2 – Diagramme d'un algorithme génétique standard

1.3.3 Les opérateurs

Un algorithme génétique simple utilise les trois opérateurs suivants : la sélection, le croisement et la mutation.

1.3.3.1 L'opérateur de sélection

L'opérateur de sélection désigne le processus par lequel les individus d'une population sont choisis en fonction de leur fonction de *fitness* dans le but de former une nouvelle génération. Le choix d'un individu est proportionnel à sa fonction de fitness. Un individu avec une valeur de *fitness* plus élevée aura plus de chance d'être sélectionné qu'un autre avec une valeur de *fitness* inférieure. Un opérateur simple de sélection est la méthode du *Roulette Wheel* ou roulette pondérée où chaque individu d'une population dispose d'une probabilité d'être choisi proportionnelle à sa valeur de fitness. Le choix des individus devant se reproduire est alors effectué sur la base de ces probabilités et est aléatoire. En effet, l'aspect aléatoire tend à protéger la

population d'un manque de diversité en évitant de privilégier d'un individu ou d'un groupe d'individu sur le reste de la population.

1.3.3.2 L'opérateur de croisement

Le croisement s'applique sur deux individus. Sa principale composante est le transfert de gènes des deux individus sélectionnés dans le but de produire de nouveaux chromosomes dits chromosomes "*fils*". Chacun des enfants héritent donc d'une partie du patrimoine génétique des parents. Différentes formes de croisement existent au nombre desquelles nous pouvons citer :

Le croisement à un point : Il consiste à désigner au hasard un point du génotype. Ce point divise donc le génotype des parents en deux parties distinctes. Ainsi, un des enfants prend une partie "*avant*" d'un des parents et le recombine avec la partie "*arrière*" de l'autre parent. Le second fait de même en prenant la partie "*arrière*" du premier parent et le recombine avec la partie "*avant*" du second parent.

Le croisement à deux points : Le principe est le même que celui du croisement en un point sauf que deux points de croisement sont désignés ici au hasard. Ces deux points divisent le génotype des parents en trois sections qui seront transférées aux individus "*fils*"

Le croisement uniforme : Il a été proposé par Syswerda [36] et consiste à désigner pour un individu "*fils*" et avec la même probabilité attribuée aux gènes de l'individu parent quels sont les allèles qui lui seront transmis à leur même position dans l'individu parent.

1.3.3.3 L'opérateur de mutation

La mutation est une opération qui n'intervient que sur un unique individu. La mutation consiste à altérer la valeur d'un gène pour un individu. Il s'agit d'un remplacement de la valeur d'un gène par une autre valeur. La mutation est un élément important dans la conservation de la diversité au sein d'une population. En règle générale, la mutation n'intervient que très rarement et n'améliore pas le qualité d'un individu. La mutation vise à reproduire l'erreur qui survient dans le nature lorsqu'un chromosome est reproduit ou copié.

1.3.4 Les algorithmes génétiques parallèles

Le développement d'ordinateurs massivement parallèles et le besoin toujours plus pressant de performance dans les systèmes d'aide à la prise de décision en industrie ont amené les chercheurs à se pencher sur la problématique de l'amélioration des performances en temps et en qualité des algorithmes génétiques. Des recherches menées [4], trois modèles tous parallèles d'algorithmes génétiques ont émergé. L'idée derrière ces modèles est l'application du principe de "diviser pour règner". Il s'agit de diviser une tâche principale en sous-tâches à attribuer à

différents processeurs. Ces trois modèles appliquent à leurs manières ce principe. Le premier modèle est le modèle des algorithmes génétiques parallèles de type *master-slave*, le second est le modèle des algorithmes génétiques parallèles de type *coarse-grained* et le troisième est le modèle des algorithmes génétiques parallèles génétiques parallèles de type *fine-grained*.

1.3.4.1 Les algorithmes génétiques parallèles de type *master-slave*

Les *master-slave parallel genetic algorithms* (M-PGA) ou algorithmes génétiques parallèles de type *master-slave* sont le type le plus simple d'algorithmes génétiques parallèles. Elles consistent essentiellement à distribuer l'évaluation de la population globale entre plusieurs processeurs. Le processeur qui conserve la population et exécute l'algorithme génétique est le maître et les processeurs qui évaluent la population sont les esclaves. La figure 1.3 montre un schéma de l'algorithme génétique parallèle de type *master-slave*.

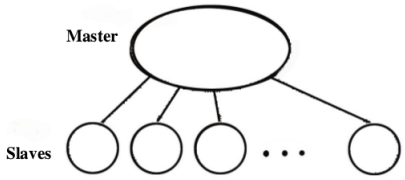

FIGURE 1.3 – Schéma d'un algorithme génétique de type "master-slave" [5]

1.3.4.2 Les algorithmes génétiques parallèles *coarse-grained*

Les algorithmes génétiques de type "*coarse-grained*" ou encore "*modèle de l'île*" ou encore *coarse-grained parallel genetic algorithms* (C-PGA) consistent en un ensemble de sous-populations qui échangent des individus de manière fréquente. La complexité de ces algorithmes réside dans le fait qu'ils nécessitent de contrôler beaucoup de paramètres. Cependant, plus les sous-populations sont utilisées, plus leur taille peut être réduite sans sacrifier la qualité de la recherche. Vu que chaque processus s'exécute en parallèle, il en résulte une réduction du temps dédié aux calculs. Cependant, utiliser plus de sous-populations et donc de processus augmente la communication dans le système. Un compromis doit donc être trouvé entre le temps de calculs et le temps de communication. L'implémentation des algorithmes génétiques parallèles de type *"coarse-grained"* soulève généralement des problématiques concernant principalement la migration qui est l'échange d'individus entre processus. Il s'agit de la fréquence de migration, du chois et du nombre d'individus à échanger, de la topologie des connexions entre processus et de la méthode d'intégration des migrants.

Fréquence de migration

La migration affecte la qualité de la recherche et l'efficacité de l'algorithme en plusieurs

points. Ainsi, de fréquentes migrations entraînent l'échange massif de potentiellement bons matériels génétiques, mais il affecte aussi négativement la performance dans la mesure où les communications sont coûteuses. La même chose se produit dans les topologies densément connectées où chaque processus communique avec les autres. Le but ultime des algorithmes génétiques parallèles est de trouver de bonnes solutions assez rapidement. Il est donc nécessaire de trouver l'équilibre entre le coût de la migration et l'augmentation des chances de trouver de bonnes solutions.

Choix et nombre de migrants

La migration envoie un nombre prédéterminé d'individus d'un processus à ces processus voisins logiques sur le graphe de communications. Ces individus ou "*migrants*" seront ainsi intégrés dans la population des processus auxquels ils ont été envoyés. Il est possible de choisir les "migrants" de façon aléatoire ou alors parmi les meilleurs individus de la population actuelle. La sélection aléatoire a l'avantage de disséminer plus de diversité et les chances d'explorer de nouvelles régions de l'espace de recherche peuvent être améliorées. La sélection des meilleurs individus peut aider à disséminer un matériel génétique qui a déjà été testé et qui serait donc intéressant.

Topologie de connexions

La topologie est également une importante partie des algorithmes génétiques de type *"coarse-grained"*. En théorie, toutes les topologies arbitraires peuvent être utilisées. Cependant, certains modèles sont fréquents. Il s'agit : des topologies linéaires, des anneaux, des hypercubes. des densément connectés, des isolés.

Méthode d'intégration des migrants

Différentes alternatives existent pour incorporer les "*migrants*". Deux d'entre elles sont récurrentes. Il s'agit : du remplacement aléatoire des individus de la population actuelle par les "*migrants*" et du remplacement compétitif ou élitiste. Dans le remplacement aléatoire, les individus devant être remplacés sont désignés de manière aléatoire. Dans le remplacement compétitif, seuls les individus avec les plus mauvais scores de "fitness" sont remplacés par les nouveaux arrivants.

1.3.4.3 Les algorithmes génétiques parallèles *fine-grained*

Les algorithmes génétiques parallèles de type "*fine-grained*" ou *fine-grained parallel genetic algorithms* (F-PGA) ont seulement une population, mais la structure spatiale limite les interactions entre les individus. Un individu ne peut compétir et se reproduire qu'avec ses individus voisins. Vu que les voisinages se chevauchent, les bonnes solutions peuvent ainsi se disséminer

à travers la population entière. Le choix le plus important dans l'implémentation de ce type d'algorithmes génétiques parallèles est : la topologie de connexions.

Topologie de connexions

Différentes topologies de connexions sont valables. Il s'agit entre autres des grilles 2-D, des hypercubes, le torus, le cube. Il est cependant répandu de placer les individus dans un algorithme génétique parallèle de type "*fine-grained*" dans une grille 2-Dimension. En effet, dans la plupart des ordinateurs massivement parallèles, les éléments de traitement et de calculs sont connectés en suivant cette topologie [4].

Principe de fonctionnement

Le principe de fonctionnement des algorithmes génétiques parallèles de type "*fine-grained*" est différent de celui de type *coarse-grained* et *master-slave* et est détaillé à l'algorithme 2.

Algorithme 2 : Principe des algorithmes génétiques parallèles de type "*fine-grained*" [33]

```
pour chaque nœud en parallèle faire
    generer un individu de façon aléatoire
fin

tant que le critère de terminaison n'est pas satisfait faire
    pour chaque nœud en parallèle faire
        evaluer le fitness de l'individu
        obtenir la valeur de fitness des individus voisins
        selectionner l'individu voisin dont la valeur de fitness est la plus grande
        appliquer un croisement avec cet individu
        muter l'individu qui en a résulté
    fin
    Tester le critère de terminaison
fin
```

1.3.4.4 Hiérarchisation entre algorithmes génétiques parallèles

***Algorithmes génétiques parallèles et hiérarchiques entre* coarse-grained *et* master-slave**

Une fois, ces différents modèles d'algorithmes génétiques parallèles abordés, la question de l'implémentation rapide des algorithmes génétiques parallèles de type *coarse-grained* revient. Une des réponses serait en effet de combiner ces algorithmes génétiques parallèles avec les algorithmes génétiques parallèles de type *master-slave* par exemple. On obtient alors une nouvelle classe d'algorithmes génétiques parallèles que sont les algorithmes génétiques parallèles

hiérarchiques[4] avec au niveau supérieur un algorithme génétique parallèle de type *coarse-grained* et au niveau inférieur un algorithme génétique parallèle de type *master-slave* comme le présente la figure 1.4.

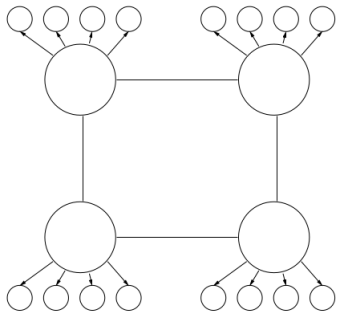

FIGURE 1.4 – Algorithmes génétiques parallèles et hierarchiques avec au niveau supérieur un algorithme génétique parallèle de type *coarse-grained* et au niveau inférieur un algorithme génétique parallèle de type *master-slave* [4]

Algorithmes génétiques parallèles et hiérarchiques entre coarse-grained *et* fine-grained

Des chercheurs [4] ont combiné deux des modèles d'algorithmes génétiques parallèles produisant des algorithmes génétiques parallèles hiérarchiques. Il s'agit des algorithmes génétiques parallèles de type *coarse-grained* et de type *fine-grained*. Ce nouvel algorithme ajoute un nouveau degré de complexité à ces algorithmes déjà compliqués. On obtient alors des algorithmes génétiques parallèles hiérarchiques avec au niveau supérieur un algorithme génétique parallèle de type *coarse-grained* et au niveau inférieur un algorithme génétique de type *fine-grained* comme détaillé à la figure 1.5.

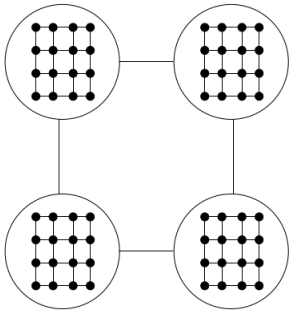

FIGURE 1.5 – Hiérarchie entre algorithmes génétiques parallèles de type *fine-grained* et de type *coarse-grained* [4]

1.3.4.5 Hybridation

L'hybridation [19] consiste à combiner deux méthodes de recherche afin d'engendrer une nouvelle méthode de recherche dit *hybride*. Les algorithmes génétiques facilitent l'hybridation avec les autres techniques de recherche locale afin d'obtenir la solution optimale. De façon basique, la recherche locale et les algorithmes génétiques sont complémentaires. Les algorithmes génétiques sont efficaces lorsqu'il s'agit de parcourir un espace de recherche global, dans la mesure où elles sont capables de rapidement trouver des régions prometteuses. Cependant, elles prennent relativement beaucoup de temps à trouver des optimums dans ces régions. La recherche locale est capable de trouver des optimums avec une grande précision.

1.3.5 Application des algorithmes génétiques aux problèmes d'optimisation

Les algorithmes génétiques ont fait l'objet de différentes études ces dernières années [27]. Ils ont montré leur efficacité sur des problèmes combinatoires tels que les problèmes d'ordonnancement [9] et les problèmes de collecte et de distribution. Les algorithmes génétiques ont été testés et ont permis notamment de résoudre les problèmes d'ordonnancement d'un atelier classique de type Job-Shop (JSP). [1]. Des algorithmes hybrides ont aussi été appliqués avec succès sur des problèmes d'optimisation. [6]. La représentation du génotype reste une problématique importance dans la résolution de ces problèmes. Différentes approches de représentation et différents types d'opérateurs d'AGs ont été proposés, pour résoudre ces problèmes [32].

Conclusion

Le dimensionnement de lots en planification de production est un important défi pour les entreprises industrielles. Il consiste à trouver un plan de production qui satisfait aux contraintes spécifiques relatives au système de production. Le PSP constitue en effet une variante NP-Difficile de ces types de problèmes. Plusieurs méthodes peuvent servir à résoudre ce problème. Au nombre de ces méthodes, figurent les algorithmes génétiques. Les AGs, à travers l'exploration et l'exploitation de l'espace de recherche ont permis de résoudre bon nombre de problèmes d'optimisation par le passé. Nous avons également présenté le PSP ainsi que les modèles et leur formulation qui ont été utilisés dans sa résolution. Dans la partie suivante, nous décrivons les outils de test, les méthodes de résolution proposées ainsi que les algorithmes associés.

Chapitre 2

Matériel et Méthodes

Résumé. Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnaires. Ils permettent d'obtenir une solution approchée ou exacte à un problème d'optimisation. Au nombre de ces algorithmes génétiques figurent une catégorie connue sous le nom d'algorithmes génétiques parallèles. En combinant ces différents algorithmes génétiques parallèles [4], l'on obtient alors des algorithmes génétiques parallèles et hiérarchiques encore plus efficaces.

Introduction

Dans ce chapitre, nous présentons dans un premier temps les outils de test utilisés, puis le modèle utilisé dans notre étude afin de représenter les instances de PSP et ensuite les aspects généraux ou communs à nos deux approches heuristiques basées sur les algorithmes génétiques. Dans un second temps, nous décrivons ces deux méthodes de recherche en nous appuyant sur les algorithmes implémentés.

2.1 Outils de test

2.1.1 Matériel

Pour l'implémentation de nos tests, nous avons travaillé sur un ordinateur présentant les caractéristiques suivantes :

- Système d'exploitation : Linux Ubuntu 16.04 LTS ;

- Processeur : Intel® Core ™ i7 CPU L 640 @ 2.13GHz x 4 ;
- Mémoire : 3,7 Gio ;
- Type du système d'exploitation : 64 bits.

2.1.2 Langage de programmation

Le langage de programmation utilisé afin d'implémenter nos deux approches est le langage *Python* dans sa version Python 3.5. *Python* est un langage de programmation interprété et orienté objet. Il est placé sous une licence libre et fonctionne sur la plupart des plate-formes informatiques [26].

2.2 Modèle et formulation utilisés

Dans le but de modéliser et de formuler les instances de PSP, nous nous sommes servis de la première formulation en programmation en nombres entiers (MIP1). Nous rappelons ici ce modèle.

$$min \sum_{i,j,t} q^{i,j}\chi_t^{i,j} + \sum_{i,t} h^i s_t^i \tag{2.1}$$

$$s_0^i = 0, \forall i \tag{2.2}$$

$$x_t^i + s_{t-1}^i = d_t^i + s_t^i, \forall i, t \tag{2.3}$$

$$x_t^i \leq y_t^i, \forall i, t \tag{2.4}$$

$$\sum_i y_t^i = 1, \forall t \tag{2.5}$$

$$\chi_t^{i,j} = y_{t-1}^i + y_t^j - 1, \forall i, j, t \tag{2.6}$$

$$x, y, \chi \in \{0,1\}, s \in \mathbb{N}, i \in \{0..NI\}, t \in \{1..NT\} \tag{2.7}$$

avec les variables de décisions suivantes :

- x_t^i : variable binaire de production qui vaut 1 si l'article i est produit à la période t et 0 sinon ;
- y_t^i : variable binaire de setup qui vaut 1 si la machine est préparée pour la production de l'article i et 0 sinon ;
- s_t^i : variable entière de stockage qui contient le nombre d'articles i stockés à la période t ;

- $\chi_t^{i,j}$: variable binaire de transition qui vaut 1 si à la période t, on est passé de la production de l'article i à l'article j et 0 sinon.

2.3 Aspects généraux aux deux méthodes de recherche proposées

2.3.1 Représentation génétique

Différentes représentations peuvent être utilisées avec les techniques évolutionnaires telles que les algorithmes génétiques. La représentation la plus simple pour les algorithmes génétiques est celle utilisée par John Holland : une chaîne de bits. Une chaîne de bits est connue comme un chromosome, et chaque bit est un gène. En début d'étude, nous avons donc commencé par représenter un chromosome en chaîne de bits.

Exemple :

En suivant l'exemple d'une instance de PSP en page 9, nous pouvons représenter un chromosome conformément à la figure 2.1.

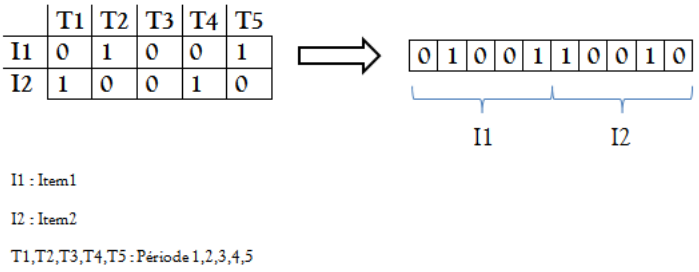

FIGURE 2.1 – Représentation génétique initiale

Autrement dit :

$$ch_T = \{(x_{1,1}), ..., (x_{1,t+1}), ..., (x_{1,T}), (x_{i+1,1}), ..., (x_{i+1,t+1}), ..., (x_{i+1,T}), ..., (x_{I,T})\}$$

où ch_T est un chromosome dont l'horizon de planification est de T périodes et x_{it} est la variable booléenne qui indique la production ou non d'un article *i* en période *t*.

Dans cette représentation, un chromosome est une chaîne de bits (0 et 1) qui indique la production ou non d'un article *i* et de longueur *nItems* * *nTimes* (où *nItems* est le nombre d'articles et *nTimes* est le nombre de périodes). Ainsi, l'article 1 est produit dans les périodes 2 et 5 ; et l'article 2 est produit dans les périodes 1 et 4. Le chromosome représenté ci-dessus est ainsi un plan de production qui satisfait aux contraintes du système de production spécifiques à cette instance du problème. Toutefois, au cours de notre étude, une seconde représentation nous est

apparue plus facilement manipulable. Cette représentation génétique est présentée à la figure 2.2 et étudiée par Mirshekarian et al. [32].

T1	T2	T3	T4	T5
2	1	0	2	1

T1, T2, T3,T4,T5 : Période 1,2,3,4,5

FIGURE 2.2 – Représentation génétique adoptée

Dans cette représentation, un chromosome est une suite d'entiers correspondant aux articles produits et de longueur *nTimes*. La longueur réduite de ce chromosome réduit dans le même temps la durée du parcours du chromosome lors des implémentations.

2.3.2 Initialisation

Le processus d'initialisation consiste à construire la population initiale ; c'est à dire celle à partir de laquelle se feront les opérations de sélection, croisement ou encore mutation afin de la faire évoluer sur des générations. L'initialisation au niveau des algorithmes génétiques se fait de manière aléatoire dans l'optique de trouver des individus suffisamment différents capables de constituer une population diverse dans le but de largement couvrir l'espace de recherche.

Une autre approche consiste à créer les individus devant composer la population initiale en se servant de stratégies de recherche informées munies de fonctions d'évaluation menant à des bonnes solutions. Notre approche a été donc d'utiliser une stratégie de recherche en l'occurrence le *Hill Climbing* [1] dotée d'une fonction d'évaluation afin de déterminer lequel des fils à considérer.

Principe :

Le principe d'initialisation de la population est le suivant : remplir un chromosome en commençant par le dernier gène et en finissant par le premier. Cela permet de s'assurer de la faisabilité des solutions que nous trouvons au bout du processus. Ainsi, pour chaque gène visité, nous disposons de la liste des allèles qui peuvent prétendre occuper ce dernier, nous pouvons alors choisir l'allèle qui minimise le "fitness" du chromosome entrain d'être constitué.

Algorithme :

L'algorithme 3 détaille le processus de génération de la population initiale :

[1] technique d'optimisation mathématique qui appartient à la famille des méthodes de recherche locale

Algorithme 3 : Processus de génération de la population initiale

Données : instance de PSP à traiter, taille de la population

Résultat : Population initiale constituée

$queue \leftarrow []$

$noeud \leftarrow nouveauNoeud()$

tant que *taille(populationInitiale) est inférieure à taillePopulation* **faire**

  si *noeud.chromosome est prêt* **alors**

    populationInitiale.ajouter(noeud.chromosome)

  sinon

    $noeudFils \leftarrow noeud.obtenirSuccesseurs()$

    noeudFils.trier(decroissant)

    queue.ajouter(noeudFils)

    si *queue est vide* **alors**

      retourner populationInitiale

    fin

    $noeud \leftarrow queue.dernier()$

  fin

fin

retourner populationInitiale

L'algorithme 4 décrit la génération des successeurs d'un noeud donné.

Algorithme 4 : Processus de génération des successeurs d'un noeud

Données : noeud

Résultat : liste des successeurs constituée

$successeurs \leftarrow []$

pour *article* $\leftarrow$ *nombreArticles* **à** *1* **faire**

  si *article.dernierDeadline* $>=$ *noeud.positionActuelle* **alors**

    noeudFils = copie(noeud)

    noeudFils.positionActuelle -= 1

    successeurs.ajouter(noeudFils)

  fin

fin

retourner successeurs

Description de l'algorithme :

L'algorithme d'initialisation prend en entrée l'instance de PSP à résoudre et une taille limite pour la population et retourne en sortie la population initiale. Un nœud est une structure de données qui contient un chromosome et des informations relatives à la recherche telles que la profondeur du nœud visité, etc. La population initiale est une liste de chromosomes issus de la recherche de *Hill climbing*. La primitive *taille()* nous permet de récupérer la taille de la population. Ainsi pour un nœud donné, tant que la taille de la population est inférieure à la taille limite définie en entrée, nous étendons ce nœud (primitive *obtenirSuccesseurs()*). Les successeurs sont triés (*noeudFils.trier()*) puis ajoutés (*queue.ajouter*) à la queue qui est une pile. Diverses autres fonctions nous permettent par exemple de retirer un élément de la queue (*queue.dernier()*) ou de copier un nœud (*copie*). Une fois un chromosome entièrement constitué, il est ajouté à la population initiale (*populationInitiale.ajouter()*).

La figure 2.3 présente un schéma illustratif de l'application de l'algorithme du Hill climbing à l'instance de PSP détaillée en page 9. A travers cette figure et en appliquant notre algorithme basé sur le *hill climbing*, le nœud "*fils*" de gauche sera choisi sur le nœud "*fils*" de droite car il présente une valeur de coût inférieure.

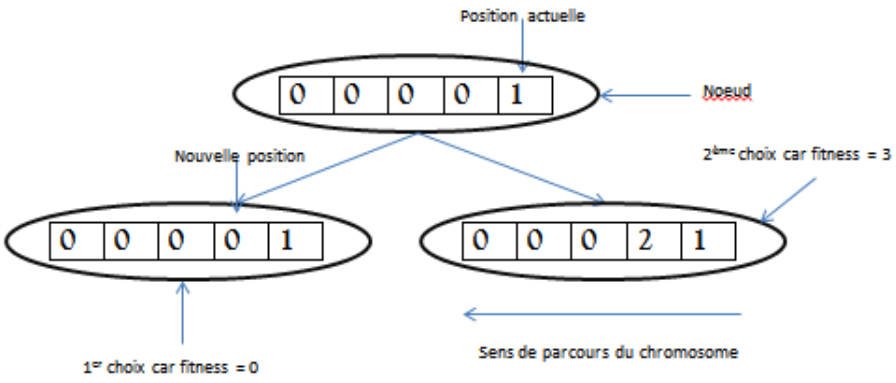

FIGURE 2.3 – Schéma illustratif de l'application de l'algorithme du Hill climbing à une instance de PSP

2.3.3 Opérateurs génétiques

2.3.3.1 Sélection

Dans notre étude, nous choisissons de nous intéresser à la plus connue et commune d'entre les méthodes de sélection et qui offre en général des résultats intéressants sur ce genre de problèmes : le *"Roulette Wheel"*. Rappelons que dans la sélection par "*roulette wheel*", les individus se voient attribués une probabilité d'être sélectionnés, proportionnelle à leur fonction

d'évaluation dans le but de produire des "*fils*". Ces "*fils*" sont ainsi de nouvelles solutions au problème et forment une nouvelle population.

2.3.3.2 Croisement

Une fois les individus sélectionnés, intervient le croisement. Le croisement en un point a été choisi afin de reproduire ces individus. La figure 2.4 présente une illustration du croisement appliqué à l'instance de PSP introduite à la page 9.

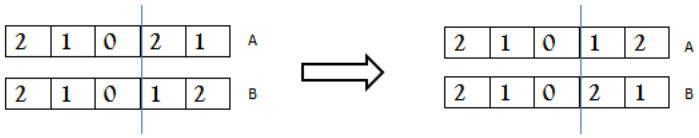

FIGURE 2.4 – Illustration du croisement utilisé

Le croisement se fait ici après la troisième période. Le croisement peut engendrer des individus qui, contrairement à la figure 2.4, ne respectent pas les contraintes de système en terme de *shortage* ou de *backlogging* [2]. Il faut alors rendre ces individus à nouveau faisables avant de procéder à une quelconque mutation.

Algorithme

L'algorithme 5 présente le croisement implémenté dans le cadre de cette étude. Notons que notre algorithme a une complexité linéaire en $O(n)$ avec n, le nombre de périodes du problème à résoudre.

[2]On parle de backlogging lorsqu'on peut satisfaire une demande après son deadline

Algorithme 5 : Algorithme de croisement utilisé

Données : parent1, parent2, seuil_probabilite
Résultat : child1 et child2

child1 $\leftarrow nouveau\ chromosome()$
child2 $\leftarrow nouveau\ chromosome()$
$probabilite \leftarrow random(1, 99)$

si *probabilite* $\leq$ *seuil_probabilite* **alors**
  $nbGenes \leftarrow NombreDeGenes(parent1)$
  $indice\_gene \leftarrow random(1, nbGenes)$
  $//Le\ nombre\ de\ genes\ du\ parent1\ est\ le\ meme\ que\ celui\ du\ parent2$
  pour $i \leftarrow 0$ **à** *nbGenes* **faire**
    $gene1 \leftarrow obtenirGene(parent1, i)$
    $gene2 \leftarrow obtenirGene(parent2, i)$
    si $i \leq$ *indice_gene* **alors**
      $child1.ajouterGene(gene1)$
      $child2.ajouterGene(gene2)$
    sinon
      $child2.ajouterGene(gene1)$
      $child1.ajouterGene(gene2)$
    fin
  fin
fin
retourner *child1, child2*

Description de l'algorithme

L'algorithme utilisé afin de croiser deux chromosomes parents prend entrée les deux parents en plus du seuil de probabilité de croisement. Nous initialisons les deux chromosomes "*fils*" avec la primitive *nouveauChromosome()* et récupérons la probabilité de croisement (primitive *random()*). Si cette probabilité est inférieure ou égale au seuil de probabilité défini en entrée, alors la croisement a lieu. Nous déterminons le point de croisement à l'aide des deux primitives *NombreDeGenes()* et *random()*. Tous les gènes du premier parent se situant avant le point de croisement sont copiés dans le premier "*fils*" (primitives *obtenirGene* et *ajouterGene*) et tous les gènes du deuxième parent se situant avant le point de croisement sont copiés dans le deuxième "*fils*". Le même processus se fait pour le reste des chromosomes parents en intervertissant les chromosomes "*fils*".

2.3.3.3 Mutation

Après la sélection et le croisement, une nouvelle population d'individus est prête. Certains ont été copiés directement et d'autres se sont reproduits par croisement. Dans le but de s'assurer que les individus ne sont pas exactement les mêmes, une mutation est appliquée à chacun des individus "*fils*". Un visuel de la mutation appliquée à l'instance de PSP explicitée en section 1.2.2 est présenté à la figure 2.5.

Principe

L'objectif de notre algorithme de mutation est de parvenir à altérer la valeur d'un gène. Pour ce faire et dans le souci de conserver le chromosome comme étant une solution du problème, nous avons décidé de permuter les valeurs de gènes choisis de façon aléatoire. Ainsi, dans un premier temps, nous choisissons de façon aléatoire un gène à muter. Une fois que cela est fait, nous déterminons toujours aléatoirement une autre valeur d'article pour ce gène ou période. Cette nouvelle valeur nous permet, en partant de la période du premier gène (aléatoirement choisi) pour le premier gène du chromosome, de déterminer le premier gène ayant cette valeur et dont la date limite de demande est supérieure à notre premier gène aléatoirement choisi.

Algorithme

L'algorithme 6 détaille le processus utilisé afin de faire muter un chromosome.

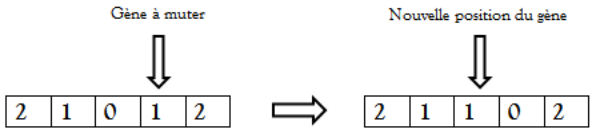

FIGURE 2.5 – Illustration de la méthode de mutation

Algorithme 6 : Algorithme de mutation utilisé

Données : chromosome, seuil_probabilite
Résultat : chromosome

$probabilite \leftarrow random(1, 99)$

si *probabilite* $\leq$ *seuil_probabilite* **alors**
  $nbGenes \leftarrow NombreDeGenes(chromosome)$
  $indice\_gene \leftarrow random(1, nbGenes)$
  $gene1 \leftarrow obtenirGene(chromosome, indice\_gene)$
  $//Le\ nombre\ de\ genes\ du\ parent1\ est\ le\ meme\ que\ celui\ du\ parent2$
  pour $i \leftarrow$ *indice_gene* **à** *0* **faire**
    $gene2 \leftarrow obtenirGene(chromosome, i)$
    si *deadline(gene2)* $>=$ *indice_gene* **alors**
      $deplacer(gene1, gene2, chromosome)$
      $arret$
    fin
  fin
fin
retourner *chromosome*

Description de l'algorithme

L'algorithme de mutation utilisé dans le cadre de notre travail prend en entrée un objet de type chromosome avec un seuil de probabilité de mutation et retourne ce même objet *chromosome* modifié. Au cours du processus, la primitive *random()* retourne une probabilité de mutation. Dans le cas où cette probabilité est supérieure au seuil de probabilité défini en entrée, la mutation survient. Nous récupérons alors le nombre de gènes du chromosome pour le problème à résoudre à l'aide de la primitive *NombreDeGenes()*. Notons que ce nombre de gènes est égal au nombre de périodes défini dans le problème. La primitive *random()* nous permet de déterminer de tous les gènes, lequel muter. Ce gène est ainsi récupérer à l'aide de *obtenirGene()*. En fin de processus, une fois les deux gènes à permuter connus, la primitive *déplacer()* nous permet d'effectuer la permutation.

2.3.4 Évaluation

L'évaluation dans notre étude se réfère à la fonction objectif.

Principe :

Il s'agit de minimiser les coûts d'exploitation et de production. Deux types de coût sont à

prendre en compte :

- Les coûts de preparation ou *setup* sont des coûts induits au moment d'un changement dans la configuration d'une ressource d'un type d'article à un autre. Il s'agit de perte potentielle de production durant la période de préparation, de force de travail additionnelle ou encore de ressources additionnelles brutes consommées durant la préparation.
- Les coûts de stockage qui sont induits lors du conditionnement et d'entreposage.

Algorithme :

L'algorithme 7 explicite le processus suivi afin d'évaluer un chromosome. Notre algorithme possède une complexité linéaire en O(n), avec n, le nombre de périodes du problème à résoudre. En effet, la fonction d'évaluation est une fonction qui détermine le fitness de nouveaux individus ou chromosomes. Ainsi, disposer une complexité linéaire sur cet algorithme est important afin de garder nos méthodes de recherche compétitives sur de grandes instances.

Algorithme 7 : Algorithme utilisé dans le processus d'évaluation d'un chromosome

Données : chromosome, cout_stockage, cout_transition
Résultat : eval

$eval \leftarrow 0$
$//On\ calcule\ le\ cout\ de\ stockage\ de\ chaque\ production$
pour *gene in chromosome* **faire**
  si *gene.value != 0* **alors**
    $date\_limite \leftarrow getDateLimite(gene)$
    $temp \leftarrow (date\_limite\ -\ gene.periode) * cout\_stockage(gene.value)$
    $evaluation \leftarrow evaluation\ +\ temp$
  fin
fin
$//On\ calcule\ le\ cout\ de\ transition\ de\ entre\ deux\ productions$
pour *gene in chromosome* **faire**
  si *gene.valeur != 0* **alors**
    $next\_gene \leftarrow obtenirProchainGene(chromosome)$
    si *transition(gene, prochain_gene) est vrai* **alors**
      $temp \leftarrow cout\_transition(gene, prochain\_gene)$
      $evaluation \leftarrow evaluation\ +\ temp$
    fin
  fin
fin
retourner *evaluation*

Description de l'algorithme :

L'algorithme d'évaluation implémenté lors de notre étude prend en entrée un chromosome

ainsi que le liste des coûts de transition et de stockage liés aux différents articles. Dans un premier temps, nous calculons les coûts de stockage de chaque article produit. Pour ce faire, nous parcourons le chromosome et pour chaque article produit, nous obtenons la période limite de la demande à l'aide la primitive *obtenirDateLimite()*. L'écart entre la période limite de la demande et la période de production multiplié par le coût de stockage de l'article (primitive *cout_stockage()*) nous donne le total des coûts de stockage. Dans un second temps, nous calculons les coûts de transition. Et pour chaque article produit, nous déterminons l'article produit à la période suivante (primitive *obtenirProchainGene()*). Le coût de transition entre ces deux articles est obtenu avec la primitive *cout_transition()*. Les coûts de transition sont ainsi ajoutés aux coûts de stockage pour fournir l'évaluation du chromosome.

2.3.5 Terminaison

Deux moyens sont utilisés par lesquels les algorithmes génétiques se terminent. Habituellement, une limite est mise sur le nombre de générations après lesquelles le processus se termine. Avec certains problèmes, le processus de recherche se termine quand une solution particulière a été trouvée ou encore lorsque la plus haute valeur de "fitness" dans la population a atteint une valeur particulière.

Le critère de terminaison utilisé dans notre étude afin de terminer une recherche est le suivant : la recherche se termine lorsque l'algorithme converge sur un individu considéré comme une solution optimale. A cet optimal local, est appliqué une fonction de recherche locale afin de déterminer dans l'entourage immédiat de cet individu, un autre individu de meilleure qualité. Dans le cas, où un meilleur individu ne serait pas trouvé, la recherche s'arrête donc sur cet optimal local.

2.3.6 Fonction de la faisabilité

Le croisement et la mutation sont tous des opérateurs génétiques qui produisent en sortie des chromosomes. En fonction du gène muté dans le cas de la mutation ou du point de croisement dans le cas du croisement, ces chromosomes peuvent ne pas être faisables ; c'est-à-dire qu'ils ne représentent pas des solutions au problème à résoudre. Il importe donc de les rendre à nouveau faisable. La fonction de faisabilité permet de rendre un chromosome ou individu à nouveau faisable.

Principe :

Dans le but de rendre un chromosome faisable, nous réduisons le surplus de production dans le cas d'un surplus et augmentons la quantité d'articles produits dans le cas d'un manque de production. Concrètement, il s'agit dans un premier temps de parcourir le chromosome à la recherche, pour chaque date limite de demandes, de surplus de production que nous avons sup-

primés. Dans un second temps, nous avons, pour chaque demande, vérifié si elle était satisfaite. Dans le cas où la demande n'était pas satisfaite alors, nous produisons un article correspondant à la demande.

Algorithme :

L'algorithme 8 est celui utilisé dans l'objectif de rendre un chromosome faisable.

Algorithme 8 : Algorithme utilisé comme fonction de faisabilité

```
Données : chromosome, deadlines
// On commence par reduire le surplus de production ;
pour i ← 1 à Nombre_Periodes faire
    article ← chromosome.obtenirArticle(i) ;
    si estEnSurplus(article, deadline(i)) alors
        supprimerProduction(article) ;
    fin
fin
// On compense le manque de production ;
pour article in liste_articles faire
    article_deadlines ← deadlines(article) ;
    pour deadline in article_deadlines faire
        si nonProduit(dealine) alors
            produire(article);
        fin
    fin
fin
```

Description de l'algorithme :

L'algorithme utilisé afin de rendre un chromosome faisable prend en entrée un objet de type chromosome et la liste des demandes correspondant à chaque article. Pour chaque gène du chromosome, nous vérifions si l'article produit, ne représente pas un surplus de production à l'aide de la primitive *estEnSurplus()*. Dans le cas d'un surplus de production, la primitive *supprimerSurplus()* nous permet de supprimer ce surplus. Une fois les surplus identifiés et supprimés, pour chaque demande d'article (primitive *deadline(article)*), la primitive *nonProduit()* permet de savoir si la demande est satisfaite. Pour une demande non satisfaite, nous la satisfaisons avec la primitve *produire()*.

2.4 Méthodes de recherche proposées

Nous nous intéressons dans notre étude à appliquer les algorithmes génétiques parallèles et hiérarchiques "*coarse-grained*" et "*fine-grained*" et les algorithmes génétiques parallèles et hiérarchiques "*coarse-grained*" et "*master-slave*" présentés en section 1.3.4.4. Nous exposons les choix effectués dans le processus d'implémentation de ces algorithmes génétiques.

2.4.1 Algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave*

En implémentant les algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave*, nous avons effectué des choix en relation avec différents points importants. Il s'agit de la fréquence de migration, du choix et du nombre de migrants, de la topologie de connexion et de la méthode d'intégration des migrants.

2.4.1.1 Fréquence de migration

Dans l'implémentation des algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave*, nous avons donné à l'utilisateur la possibilité d'entrer l'intervalle de générations entre les migrations. Cependant, le comportement par défaut de l'algorithme est de migrer uniquement lorsque la population a convergé complètement.

2.4.1.2 Choix et nombre de migrants

Dans la suite de notre étude, nous avons choisi d'échanger les meilleurs individus. L'utilisateur a ainsi la possibilité d'entrer le nombre d'individus à faire migrer. Nous rappelons la sélection des meilleurs individus peut aider à disséminer un matériel génétique qui a déjà été testé et qui serait donc intéressant au contraire de la sélection aléatoire des "migrants".

2.4.1.3 Topologie de connexions

La topologie utilisée a été celle densément connectée présentée à la figure 2.6. Dans cette topologie, tout nœud ou processus pris individuellement est le voisin logique de tous les autres nœuds de la topologie.

2.4.1.4 Méthode d'intégration des migrants

Nous avons appliqué un remplacement compétitif à notre programme. Il s'est agi d'identifier les plus mauvais chromosomes en terme de fitness et de les remplacer par les nouveaux arrivants.

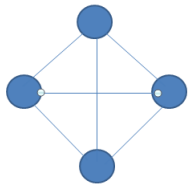

FIGURE 2.6 – Topologie de connexions utilisée

2.4.2 Algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *fine-grained*

L'implémentation des algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *fine-grained* a suivi le même processus que celui des Algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave*. Les mêmes questionnements quant à la fréquence de migration, à la topologie de connexion, à la méthode d'intégration et au nombre de migrants sont revenus.

2.4.2.1 Fréquence de migration

Dans la même idée que l'implémentation des algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave*, nous avons donné à l'utilisateur la possibilité d'entrer l'intervalle de générations entre les migrations. Cependant, le comportement par défaut de l'algorithme est de migrer uniquement lorsque la population a convergé complètement.

2.4.2.2 Choix et nombre de migrants

Afin de sélectionner les migrants nous avons choisi d'échanger les meilleurs individus. Nous assurons ainsi la dissémination de matériel génétique à travers les différents nœuds de la topologie de connexion adoptée. L'utilisateur a toute latitude d'entrer le nombre d'individus à faire migrer.

2.4.2.3 Méthode d'intégration des migrants

Nous avons choisi d'écarter les plus mauvais chromosomes (c'est à dire avec de mauvais scores de fitness) et de les remplacer par les nouveaux arrivants. Avec un remplacement aléatoire, nous avons estimé courir le risque de remplacer de potentiellement bons chromosomes par les nouveaux arrivants.

2.4.2.4 Topologie de connexions

La figure 2.7 présente la topologie (grille 2-Dimension) utilisée afin d'implémenter l'algorithme génétique parallèle de type *"fine-grained"*.

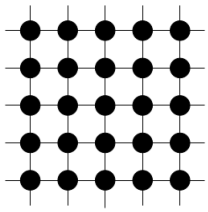

FIGURE 2.7 – Topologie utilisée en algorithme génétique parallèle de type "*fine-grained*" [4]

2.4.3 Autres algorithmes implémentés

2.4.3.1 Hybridation

Dans notre étude, Nous avons mis au point un algorithme de recherche locale qui est utilisé à chaque fois que l'algorithme génétique converge sur une solution optimale afin de parcourir l'espace de recherche immédiat à cette solution et ainsi améliorer cette solution optimale. L'algorithme 9 décrit le processus suivi afin de chercher de meilleures solutions à partir d'un chromosome fourni en paramètre. Cet algorithme a une complexité polynomiale en $O(n^2)$ avec n, le nombre de périodes du problème à résoudre ou la taille d'un chromosome.

Algorithme 9 : Algorithme de recherche locale d'une meilleure solution

Données : individu à améliorer

pour *chaque gène du chromosome* **faire**

récupérer les coûts relatifs à la valeur de ce gène

déterminer toutes les gènes à zéro respectant les contraintes liées à la valeur de ce gène.

calculer le fitness du chromosome pour chaque gène à zéro

choisir le gène à zéro qui maximise le fitness du chromosome

insérer la valeur de l'item dans ce dernier gène

fin

Si on prend l'exemple du PSP en section 1.2.2 et qu'on applique cet algorithme au problème, on obtient la figure 2.8.

2.4.3.2 Table de Hash

Dans le processus de résolution des problèmes à l'aide des algorithmes génétiques, au fur et à mesure que les générations passent, la population évolue et la diversité au sein de cette dernière diminue amenant les mêmes chromosomes à être régulièrement réévalués. Dans les faits, l'effort de calculs dépensé à évaluer le "*fitness*" dépasse celui dépensé sur les opérateurs

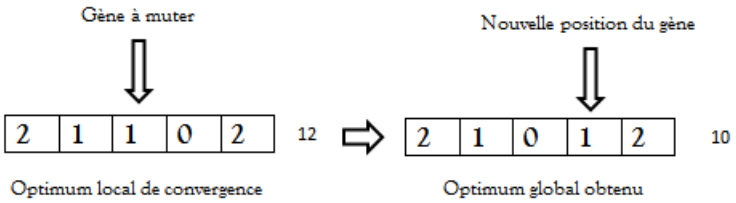

FIGURE 2.8 – Application de l'algorithme de recherche de meilleure solution à un chromosome

génétiques.

Principe :

L'utilisation d'une table de hash permet de stocker les chromosomes récemment évalués, une amélioration significative des performances peut être constatée. La table de hash est représenté dans notre programme par un dictionnaire ou tableau associatif afin de stocker ces derniers.

Algorithme :

L'algorithme 10 implémente l'utilisation de la table de hash dans nos deux méthodes de résolution proposées.

Algorithme 10 : Algorithme implémentant l'utilisation de la table de hash dans nos méthodes proposées

```
Données : chromosome, table_hash, max_len
solution ← chromosome.solution;
si solution in table_hash alors
    indice ← table_hash[solution] ;
    chromosome.valeurFitness ← table_hash[solution][valeurFitness] ;
sinon
    chromosome.valeurFitness ← evaluation(solution) ;
    table_hash.ajouter([solution, chromosome.valeurFitness]);
    si longueur(table_hash) > max_len alors
        table_hash.enleverDernier() ;
    fin
fin
```

Description de l'algorithme :

L'utilisation de la table de hash intervient lors de la constitution d'un chromosome. Notre algorithme prend en entrée un chromosome. Ainsi, au moment de la formation du chromosome, différentes propriétés doivent être remplies, on vérifie alors si ce chromosome n'avait pas déjà été constitué. Si oui, alors nous récupérons la valeur de fitness associée à la solution décrite par

le chromosome (*table_hash[solution][valeurFitness]*) que l'on affecte au chromosome. Dans le cas où il s'agit de la première constitution d'un pareil chromosome, la primitive *evaluation()* renvoie la valeur de fitness affectée au chromosome. Enfin, nous ajoutons ce chromosome à la table de hash (*table_hash.ajouter()*)

Conclusion

Cette deuxième partie a présenté le modèle utilisé dans la résolution du PSP ainsi que les aspects communs aux deux solutions proposées. Dans un second temps, nous avons présenté et détaillé deux approches heuristiques basées sur les algorithmes génétiques parallèles utilisées afin de résoudre le problème. Dans la suite, nous présentons nos deux approches et analysons les résultats afin de les comparer à l'état de l'art en la matière.

Chapitre 3

Résultats et Discussion

Résumé. Les deux approches basées sur les algorithmes génétiques présentées en chapitre 2 sont testées. Il en ressort que sur les instances proposées par Houndji [23], les deux approches parviennent à trouver des solutions très proches des solutions optimales assez rapidement. Elles ne parviennent cependant pas à en faire autant sur les grandes instances proposées par Ceschia [8].

Introduction

Dans ce chapitre, nous testons les théories émises et analysons les résultats obtenus de tests afin de vérifier nos approches de solution du problème énoncées. A cette fin, nous présentons d'abord les données (instances) et paramètres de test. Ensuite, nous expérimentons nos solutions et à partir des résultats obtenus, nous comparons nos approches heuristiques basées sur les algorithmes génétiques à celles déjà appliquées à ce problème.

3.1 Données et paramètres de test

Données de test

Afin de tester nos deux solutions, nous commençons dans un premier temps par utiliser un ensemble de 20 instances des 100 proposées par Houndji et al. [23] auxquelles ils ont appliqué la CP. Ces instances sont caractérisées par un nombre de périodes $NT = 20$, un nombre d'articles $NI = 5$ et un nombre de demandes $ND = 20$. Le choix de ces 20 instances s'est fait de la manière suivante : nous avons choisi les 5 premières instances des 100 proposées par Houndji, puis nous avons ensuite choisi 15 autres en veillant à combiner les instances sur lesquelles l'approche CP

prenait le plus de temps et celles où elle en prenait le moins. Les résultats seront comparés à ceux obtenus par Houndji et al. dans leur application de l'approche CP à ces instances.

Dans un second temps, nous appliquons nos deux solutions à 6 autres instances des 27 proposées par Ceschia et al [8] dans leur bibliothèque Opthub. En effet, dans le but de tester leur approche de solution basée sur le recuit simulé aux instances de PSP, Ceschia et al (au même titre que Houndji dans son expérimentation du CP) ont développé un générateur paramétré produisant de nouvelles instances plus grandes. Le générateur travaille de sorte que l'instance produite est faisable, c'est à dire qu'elle satisfait aux contraintes de *NoBacklog*. Nous testons notre approche de solution sur ces nouvelles grandes instances et comparons nos résultats à ceux obtenus par Ceschia et al dans leur application du recuit simulé à ces instances.

Dans les deux cas, nos critères d'analyse sont de deux ordres. Premièrement, nous avons analysé les résultats sous l'angle de la performance en temps. Il s'agit de voir comment ces algorithmes arrivent à répondre aussi rapidement que possible nos instances de PSP. Le second angle d'analyse s'est porté sur la performance en terme de la qualité de la solution trouvée. Nous avons calculé l'écart entre la solution trouvée et la solution optimale connue dans la littérature.

Paramètres de test

Les paramètres de test utilisés afin d'effectuer les tests sont présentés comme suit selon que le programme implémente *Hierarchical Coarse-grained and Master-slave Parallel Genetic Algorithm* (HCM-PGA) ou *Hierarchical Fine-grained and Coarse-grained Parallel Genetic Algorithm*(HFC-PGA) :

1. HCM-PGA

   - Taille de la population : 25 individus par processus ;
   - Probabilité de mutation : 5% ;
   - Probabilité de croisement : 80% ;
   - Nombre de migrants : 1 individu ;
   - Nombre de processus esclaves : 2 processus ;
   - Nombre de processus principaux : 2 processus ;
   - Nombre de générations avant migration : 0 génération (la migration intervient après une convergence).

2. HFC-PGA

- Taille de la population : 25 individus par processus ;
- Probabilité de mutation : 5% ;
- Probabilité de croisement : 80% ;
- Nombre de migrants : 1 individu ;
- Nombre de processus principaux : 2 processus ;
- Nombre de générations avant migration : 0 génération (la migration intervient après une convergence).

Ces paramètres de test, retenus expérimentalement, sont les valeurs qui nous sont parues maximisant la qualité des solutions trouvées par chacun des algorithmes.

3.2 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques *fine-grained* et *coarse-grained*

Nous avons testé les algorithmes génétiques parallèles hiérarchiques "fine-grained" et "coarse-grained" dans un premier temps sur 20 instances de PSP proposées par Houndji et dans un second temps sur 6 autres instances de PSP plus grandes proposées par Ceschia. Les résultats de ces tests sont consignés dans les tableaux 3.1 et 3.2.Ces tables présentent les résultats respectivement de l'approche CP et de l'approche de recuit simulé. Ces tableaux présentent les performances en temps et qualité des solutions des algorithmes génétiques parallèles hiérarchiques "fine-grained" et "coarse-grained". Nous avons effectué nos tests à travers 10 essais sur chacune des instances et obtenu des résultats compilés, une moyenne des performances de notre méthode de résolution proposée.

3.3 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques *coarse-grained* et *master-slave*

Après les tests de l'algorithme génétique parallèle hiérarchique *fine-grained* et *coarse-grained*, nous avons procédé aux tests sur l'algorithme génétique parallèle hiérarchique *master-slave* et *coarse-grained* sur les mêmes instances que celles testées ci-dessus. Au bout de 10 essais, les résultats et la moyenne obtenue des essais effectués sont consignés dans les tableaux 3.3 et 3.4 ;

[1]Coût de la solution optimale
[2]Temps en secondes de l'approche CP
[3]Temps moyen en secondes de notre approche génétique
[4]Différence entre la solution optimale et la solution trouvée par notre approche génétique
[5]Meilleure solution trouvée par notre approche génétique
[6]Nombre de générations parcourues par notre approche génétique

Instance	Opt cost [1]	CP time [2]	GAs time [3]	GAs result gap [4]	GAs Best Sol [5]	Nb Gen [6]
instance 1	1377	9.14	3.9	0.2%	1378	15
instance 2	1447	7.292	1.7	1.67%	1447	12.8
instance 3	1107	2.946	3.2	0.07%	1108	20.2
instance 4	1182	1.784	2.4	1.4%	1199	17
instance 5	1471	0.235	4.5	0%	1471	17.8
instance 8	3117	25.352	10.3	0.25%	3117	15.4
instance 21	2774	11.177	2.2	0%	2774	16.3
instance 23	1473	15.039	6.1	0%	1473	13
instance 35	2655	12.846	2.5	0.5%	2670	16.4
instance 36	1493	121.909	6.5	0%	1493	18.3
instance 53	1108	0.935	2.6	1.8%	1128	16.7
instance 58	1384	2.347	2.11	1.9%	1411	14.2
instance 61	977	0.711	2.1	0%	977	14.4
instance 69	1619	1.223	2.5	0%	1619	13.1
instance 73	1104	12.508	3.8	3.7%	1145	65.6
instance 78	1297	16.187	3.04	1.3%	1297	17.9
instance 85	2113	9.404	2.47	2.3%	2162	14
instance 87	1152	1.589	3.5	2.1%	1152	15.9
instance 90	2449	23.811	3.7	3.3%	2531	14.1
instance 94	1403	11.726	5.9	1.3%	1403	16

TABLEAU 3.1 – Performances du HFC-PGA et CP sur 20 instances du PSP

et sont comparés respectivement à ceux obtenus en programmation par contrainte et en recuit simulé.

3.4 Discussion

Nous effectuons dans cette section une analyse comparative des résultats afin de dégager des conclusions. Ainsi, nous procédons à une analyse des performances entre algorithmes génétiques parallèles hiérarchiques *fine-grained* et *coarse-grained* et programmation par contraintes ainsi que recuit simulé d'une part et d'autre part de pouvoir faire de même entre programmation par contrainte et algorithmes génétiques hiérarchiques *coarse-grained* et *master-slave* ainsi que recuit simulé. Enfin, nous répondons à la question de savoir lequel des algorithmes génétiques parallèles hiérarchiques *coarse-grained* et *fine-grained* et algorithmes génétiques parallèles hiérarchiques *master-slave* et *coarse-grained* est le plus performant. Nous tenons en effet à rappeler que l'approche CP ne saurait être comparée dans l'absolu à l'approche basée sur les algorithmes génétiques, dans la mesure où l'approche CP est une approche exacte qui prouve l'optimalité des solutions tandis que celle basée sur les algorithmes génétiques est stochastique. De même, toute comparaison et conclusion dans les performances entre approche SA et approche basée sur les algorithmes génétiques, ne pourra être lue qu'à la lumière des résultats obtenus

Instance	Opt cost	SA time	GAs time	GAs result gap	GAs Best Sol	Nb Gen
ps-200-10-80	18089	72.2	9.9	8.7%	19484	18.2
ps-200-20-80	19190	73.8	18.8	11.6%	21302	18.9
ps-300-10-80	26343	78.4	56.8	16.5%	30452	17.3
ps-300-20-80	30206	68.3	96.8	7.4%	32289	20
ps-400-10-80	34206	85.5	117.9	19.5%	40775	25.9
ps-400-20-80	41329	69.5	120.3	10.1%	45521	22.9

TABLEAU 3.2 – Performances du HFC-PGA et SA sur 6 instances du PSP

dans le cadre de notre étude.

3.4.1 HCM-PGA et Approche CP

L'approche CP dans son application au PSP s'est avérée efficace sur des instances dont l'horizon de planification est inférieur ou égal à 20 périodes. Nous analysons à présent les résultats obtenus en appliquant le HCM-PGA au PSP. La figure 3.1 nous présente sous forme de graphique les résultats des tests de performance en matière de qualité de la meilleure solution trouvée. L'analyse de ce graphique nous permet de remarquer que la courbe de performance de HCM-PGA est sensiblement la même que celle de CP. Ces deux courbes présentent la même allure et aussi sensiblement les mêmes valeurs de performance. Notre méthode HCM-PGA présente donc les mêmes performances à quelques petites différences près que l'approche de référence (approche CP). Sur les figures en rapport à l'approche CP, les numéros 1,2,3, ..., 20 correspondent respectivement aux instances 1, 2, 3, ..., 90.

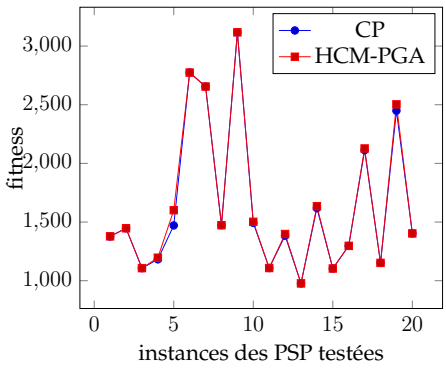

FIGURE 3.1 – Performances comparées de CP et HCM-PGA en fitness de la meilleure solution trouvée

Nous analysons ensuite la performance de HCM-PGA en matière de temps de la meilleure solution trouvée sur nos 10 essais ; comme le montre la figure 3.2, avec pour critère de comparaison le temps de recherche. Nous constatons que le HCM-PGA parcourt bien plus vite l'espace de recherche. Il nous est possible de constater que sur certaines instances le gain de

Instance	Opt cost	CP time	GAs time	GAs result gap	GAs Best Sol	Nb Gen
instance 1	1377	9.14	1.73	0.07%	1378	19.2
instance2	1447	7.292	1.4	0.2%	1447	21.3
instance 3	1107	2.946	1.6	0%	1107	17.1
instance 4	1182	1.784	1.3	1.1%	1196	21.1
instance 5	1471	0.235	2.4	8%	1601	16.5
instance 8	3117	25.352	4.08	0%	3117	18.9
instance 21	2774	11.177	2.6	0%	2774	18.3
instance 23	1473	15.039	1.9	0%	1473	20
instance 35	2655	12.846	2.6	0.2%	2655	26
instance 36	1493	121.909	3.4	0%	1502	20.5
instance 53	1108	0.935	1.5	0%	1108	21.4
instance 58	1384	2.347	1.8	1.01%	1398	20.3
instance 61	977	0.711	1.6	0%	977	24.1
instance 69	1619	1.223	1.7	0.9%	1635	21.5
instance 73	1104	12.508	1.8	0%	1104	21.9
instance 78	1297	16.187	1.8	0.4%	1297	20.7
instance 85	2113	9.404	2.3	0.9%	2127	21.5
instance 87	1152	1.589	1.8	1.6%	1152	25
instance 90	2449	23.811	2.32	2.6%	2503	24.5
instance 94	1403	11.726	2.5	0%	1403	19.7

TABLEAU 3.3 – Performances du HCM-PGA et CP sur 20 instances du PSP

temps est dans le meilleur des cas (instance 10) de 97,21% . En poursuivant notre analyse, nous pouvons également faire noter que le temps de recherche est relativement constant. Sur des instances où l'approche CP prend beaucoup plus de temps (instances 6,7,8,9 et 10), notre méthode HCM-PGA conserve un temps de recherche stable. En conséquence, l'usage de notre méthode HCM-PGA s'avère pertinent car elle permet d'avoir sur ces instances, des solutions de bonnes qualités assez rapidement.

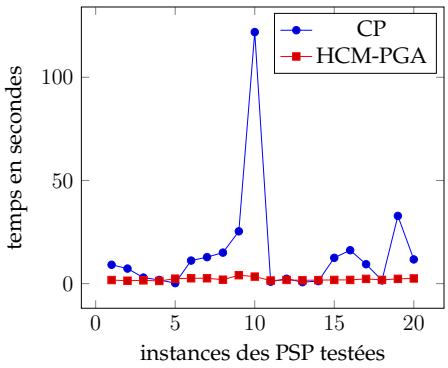

FIGURE 3.2 – Performances comparées de CP et HCM-PGA en temps

En combinant cette analyse ainsi que la précédente, nous pouvons déduire que notre

Instance	Opt cost	SA time	GAs time	GAs result gap	GAs Best Sol	Nb Gen
ps-200-10-80	18089	72.2	11.2	10%	19799	24.4
ps-200-20-80	19190	73.8	11.5	11.2%	21345	18.2
ps-300-10-80	26343	78.4	32.2	33%	34985	20.7
ps-300-20-80	30206	68.3	30.8	7%	32207	22.2
ps-400-10-80	34206	85.5	37.8	18.4%	40433	23.9
ps-400-20-80	41329	69.5	60.9	10.3%	45647	22.9

TABLEAU 3.4 – Performances du HCM-PGA et SA sur 6 instances du PSP

solution HCM-PGA réussit à parcourir l'espace de recherche plus vite que notre approche de référence tout en réussissant sur nos 10 essais tests à trouver une solution optimale ou une solution approchant cette dernière de très près ; excepté d'autres instances (instance 5). Nous allons pour la suite, analyser la moyenne des solutions trouvées en matière de performance de fitness.

Trouver une solution proche ou exacte sur un essai est une chose. Arriver à prouver que notre approche trouve des solutions voisines ou proches à la solution exacte sur de nombreux essais est encore une autre chose. La figure 3.3 nous permet de dire que notre méthode HCM-PGA peut être efficace dans la résolution des 20 instances présentées et plus largement des instances proposées par Houndji.

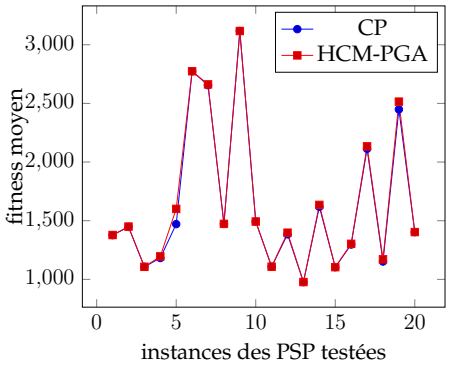

FIGURE 3.3 – Performances comparées de CP et HCM-PGA en fitness moyen des solutions trouvées

3.4.2 HFC-PGA et Approche CP

Une fois, notre première approche HCM-PGA vérifiée pour les instances proposées par Houndji ; nous passons à la seconde méthode de HFC-PGA. La procédure utilisée est similaire à celle utilisée afin de vérifier notre approche HCM-PGA. Nous avons donc analysé les performances en terme de qualité des solutions fournies par notre méthode HFC-PGA. La figure 3.4 présente les courbes des deux méthodes CP et HFC-PGA. Il en ressort que notre approche

HFC-PGA est aussi efficace que l'approche CP. En effet, la déviation maximale sur les 20 instances testées est de 3,2% (instance 90). Nous estimons cette déviation comme faible. Ainsi, nous pouvons dire que les performances en qualité des meilleures solutions obtenues de l'approche HFC-PGA sont sensiblement égales à celles obtenues de l'approche CP.

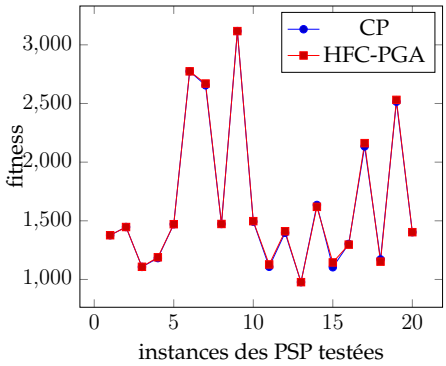

FIGURE 3.4 – Performances comparées de CP et HFC-PGA en fitness de la meilleure solution trouvée

La figure 3.5 nous présente les résultats en temps des essais de l'approche HFC-PGA comparés à ceux de l'approche CP. Ainsi, l'approche HFC-PGA parcourt également bien plus vite l'espace de recherche en comparaison à l'approche CP. Un gain de 94,67% est notamment constaté sur l'instance 10 justifiant la performance en temps de notre solution.

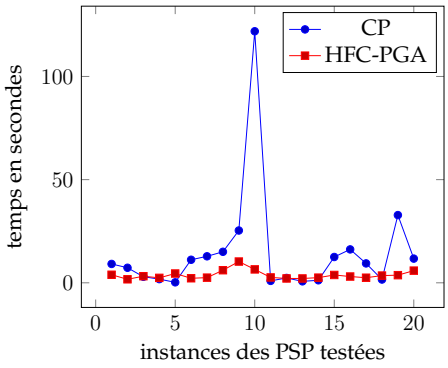

FIGURE 3.5 – Performances comparées de CP et HFC-PGA en temps

Afin de vérifier que notre méthode arrive à trouver en général des solutions proches de la solution optimale, une moyenne de résultats obtenus a été effectuée sur les 10 essais de test. Comme le montre la figure 3.6 qui présente les courbes des résultats moyens en terme de qualité du HFC-PGA ; nous pouvons remarquer que notre méthode arrive à trouver sur les différents essais effectués des solutions très proches de la solution optimale. En effet, la déviation maximale observée est de 3,3%. Des observations précédentes et de celle-ci, nous pouvons dire que

notre méthode HFC-PGA comme sa seconde HCM-PGA parcourt bien plus vite l'espace de recherche tout en trouvant de solutions optimales sinon très proches de ces dernières.

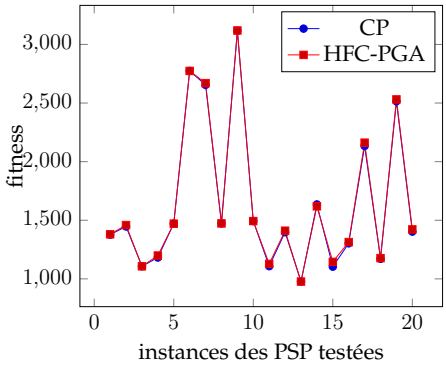

FIGURE 3.6 – Performances comparées de CP et HFC-PGA en fitness moyen des solutions trouvées

3.4.3 HCM-PGA et Approche SA

Les analyses précédentes nous ont permis de dire que nos deux approches basées sur les algorithmes génétiques performent aussi bien en qualité que l'approche CP, tout en parcourant bien plus vite l'espace de recherche. Nous allons dans la suite tenter d'analyser les résultats issus des tests effectués sur des instances bien plus grandes de la bibliothèque Opthub. Ces instances sont en effet plus grandes avec leur horizon de planification supérieur à 200 périodes. La figure 3.8 montre les courbes décrivant l'allure des résultats en temps des deux approches HCM-PGA et SA ; et nous permet ainsi de comparer les résultats obtenus.

Sur les figures en rapport à l'approche SA, les numéros 1,2,3,4,5 et 6 correspondent respectivement aux instances ps-200-10-80, ps-200-20-80, ps-300-10-80, ps-300-20-80, ps-400-10-80 et ps-400-20-80.

En observant la figure 3.7, on constate que les résultats de notre méthode HCM-PGA diffèrent nettement de celles optimales de la méthode SA. On remarque en effet, que la méthode de résolution HCM-PGA trouve des solutions dont le coût est différent de 32% dans le pire des cas (instance ps-300-10-80). Nous pouvons cependant retenir comme point positif que notre méthode HCM-PGA réussit à approcher de 6.6% dans le meilleur des cas (instance ps-300-20-80) la solution optimale.

Notons que notre méthode HCM-PGA ne performe pas aussi bien qu'attendu sur les instances plus grandes lorsqu'on parle de qualité de la meilleure solution trouvée. Nous analysons à présent ses performances en temps. La figure 3.8 nous indique que notre méthode HCM-PGA parcourt toujours aussi vite l'espace de recherche (jusqu'à 84% de gain en temps dans le meilleur des cas). Nous résumons donc de nos deux analyses en terme de temps et de qualité,

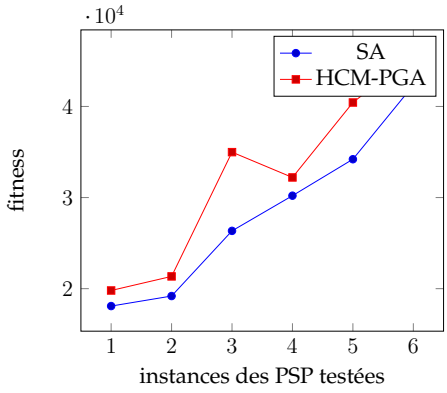

FIGURE 3.7 – Performances comparées de SA et HCM-PGA en fitness de la meilleure solution trouvée

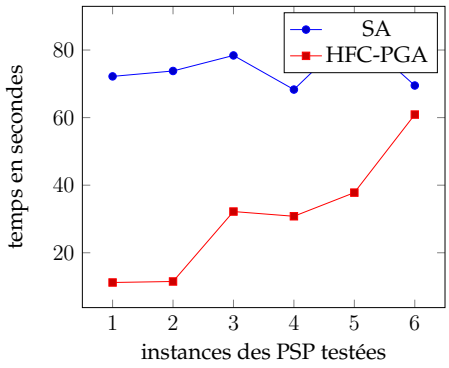

FIGURE 3.8 – Performances comparées de SA et HCM-PGA en temps

que notre méthode HCM-PGA explore toujours aussi vite l'espace de recherche mais ne réussit pas à mieux approcher la solution au delà de 6.6%.

3.4.4 HFC-PGA et Approche SA

La figure 3.9 présente les résultats des tests effectués en terme de qualité de la solution trouvée. Une analyse similaire à celle menée entre SA et HCM-PGA peut être menée dans le sens où les performances en terme de qualité des solutions obtenues des essais de la méthode HFC-PGA sur les 6 instances plus grandes, ne sont pas aussi intéressantes qu'espérées. En effet, notre méthode de HFC-PGA ne fait pas mieux que 6.4% dans le meilleur des cas (instance ps-300-20-80). Il est à noter cependant comme point positif que notre méthode HFC-PGA ne fait pas moins bien que 19.2% de déviation maximale.

Du point de vue de la performance en temps de notre méthode comparée à la méthode SA, plus l'horizon de planification s'étend plus, l'efficacité en temps se dégrade. En effet, elle est de 9.9 secondes sur un horizon de 200 périodes et de 120 secondes sur un horizon de 400 périodes ; environ deux fois plus que le temps pris par la méthode SA pour cette dernière instance comme

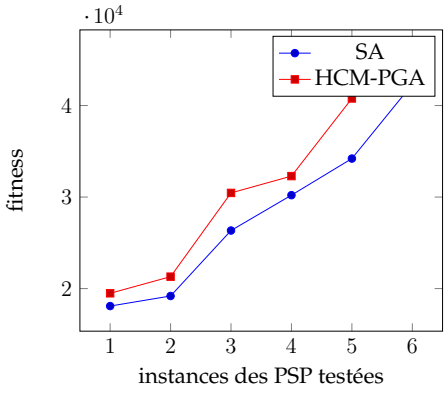

FIGURE 3.9 – Performances comparées de SA et HFC-PGA en fitness de la meilleure solution trouvée

le montre la figure 3.10.

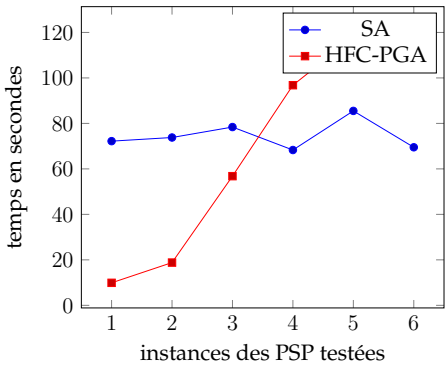

FIGURE 3.10 – Performances comparées de SA et HFC-PGA en temps

3.4.5 HCM-PGA et HFC-PGA

Dans cette section, nous tentons de répondre à la question de savoir laquelle des méthodes HCM-PGA et HFC-PGA est la plus pertinente dans la résolution des problèmes du PSP en nous basant sur les résultats des tests effectués sur les instances proposées par Houndji. Les figures 3.12 et 3.11 présentent respectivement les résultats en terme de temps et de qualité. On note ainsi globalement que la méthode de HCM-PGA est plus rapide que celle du HFC-PGA et également plus précise que cette dernière lorsqu'on parle de qualité des solutions trouvées. Nous nous expliquons cela sans doute par le fait que les algorithmes génétiques parallèles et hiérarchiques HFC-PGA doivent faire démarrer un grand nombre de nœuds (ou threads) correspondants chacune à un chromosome afin de pouvoir explorer l'espace de recherche ainsi qu'à la topologie utilisée dans le HFC-PGA.

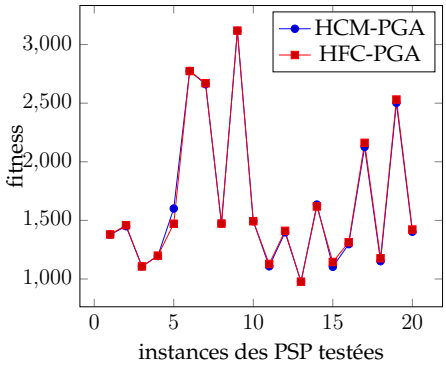

FIGURE 3.11 – Performances comparées de HCM-PGA et HFC-PGA en fitness moyen des solutions trouvées

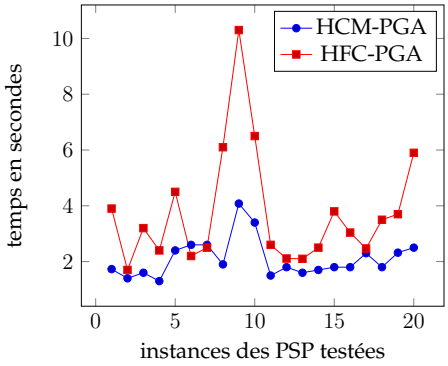

FIGURE 3.12 – Performances comparées de HCM-PGA et HFC-PGA en temps

Conclusion

Les tests effectués dans cette partie ont été l'occasion de vérifier nos deux méthodes de résolution proposées basées sur les algorithmes génétiques. Nous avons ainsi pu comparer chacune d'elles aux approches CP et SA et avons pu tirer des conclusions. Il ne nous reste plus qu'à conclure notre étude et à présenter les perspectives envisagées.

Conclusion et perspectives

Nous avons présenté les problèmes de dimensionnement de lots. Nous avons montré que la résolution de ces types de problèmes soulève certains défis intéressants. Nous nous sommes en particulier intéressés au *Pigment Sequencing Problem* (PSP). Plusieurs méthodes de résolution du PSP ont été testées au cours de récentes recherches. Il s'agit des approches MIP, CP et SA. Cependant, aucune approche basée sur les algorithmes génétiques n'avait été proposée alors que ces derniers ont montré leur efficacité sur d'autres problèmes d'optimisation.

Nous avons proposé deux méthodes de résolution du PSP en utilisant les algorithmes génétiques parallèles hiérarchiques : *Hierarchical Coarse-grained and Master-slave Parallel Genetic Algorithm* (HCM-PGA) et *Hierarchical Fine-grained and Coarse-grained Parallel Genetic Algorithm*(HFC-PGA). L'approche HCM-PGA divise la population globale en sous-populations ou îlots auxquels sont appliquées les opérations génétiques telles le croisement, la mutation ainsi que la sélection. Également, l'approche HCM-PGA confie à différents nœuds ou threads la tâche d'appliquer les opérations génétiques les plus gourmandes en ressource afin d'accélérer la recherche. Quant à l'approche HFC-PGA, elle divise dans la même logique la population globale en sous-populations ; ces dernières étant disposées suivant une topologie de connexion particulière favorisant le chevauchement des bons chromosomes et la dissémination du matériel génétique. Nous avons vérifié nos deux propositions de méthodes de résolution à travers des tests que nous avons effectués sur deux groupes d'instances. Le premier est proposé dans la bibliothèque CSPlib et le second dans la bibliothèque Opthub. Ces tests ont permis d'analyser le comportement de nos deux propositions et de dire qu'elles parviennent en un temps raisonnable à trouver de solutions approchant la solution optimale en particulier sur les instances dont l'horizon de planification est inférieur ou égal à 20 périodes.

Le travail présenté dans ce document est un bon point de départ pour de futurs développements et extensions utilisant les algorithmes génétiques. Il serait par exemple intéressant de penser à des algorithmes de recherche locale qui améliorent significativement la qualité d'un chromosome tout en étant rapides sur les chromosomes des grandes instances testées dans notre étude. D'un autre côté, le croisement est un aspect très important dans la résolution des

problèmes. Une perspective intéressante est de penser à des moyens de croisement plus intelligents qui améliorent la population. Enfin, l'application des algorithmes génétiques parallèles et hiérarchiques à des problèmes encore plus complexes à plusieurs machines ou à plusieurs niveaux est un sujet intéressant pour de futurs travaux.

Bibliographie

[1] H. Boukef, M. Benrejeb, and P. Borne. A proposed genetic algorithm coding for flow-shop scheduling problems. *International journal of computers*, pages 229–240, 2007.

[2] N. Brahimi, S. Dauzère-Pérès, N. Najid, and A. Nordili. Single item lot sizing problems. *European journal of Operational Research 168*, pages 1–16, 2006.

[3] Nadjib Brahimi, Stéphane Dauzère-Pérès, Najib.M. Najid, and Atle Nordli. Etat de l'art sur les problèmes de dimensionnement des lots avec contraintes de capacité. 04 2003.

[4] Erick Cant-paz. A survey of parallel genetic algorithms. 06 1999.

[5] E. Cantu-Paz. Implementing fast and flexible parallel genetic algorithms. 1999.

[6] S. Cavalieri and P. Caiardelli. Hybrid genetic algorithms for a multiple-objective scheduling problem. *Journal of Intelligent manufacturing*, pages 361–367, 1998.

[7] S Ceschia. opthub.uniud.it, 2016.

[8] S. Ceschia, L. Di Gaspero, and A. Schaerf. Solving discrete lot-sizing and scheduling by simulated annealing. 2016.

[9] L. Davis. Job-shop scheduling problems using genetic algorithms. *Computers industrial Engeneering*, pages 983–997, 1996.

[10] A. Drexl and Kimms A. Single item lot sizing problems. *Lot sizing and scheduling - survey and extensions*, pages 221–235, 1997.

[11] A. Drexl and K. Haase. Proportional lot-sizing and scheduling. *International Journal of Production Economics 40*, pages 73–87, 1995.

[12] B. Fleischmann. The discrete lot-sizing and scheduling problem. *European Journal of Operational Research 44*, pages 337–348, 1990.

[13] B. Fleischmann and H. Meyr. The general lot-sizing and scheduling problem. *OR Spektrum 19*, pages 11–21, 1997.

[14] B. Fleischmann and H. Meyr. Simultaneous lotsizing and scheduling by combining local search with dual reoptimization. *European Journal of Operational Research 120*, pages 311–326, 2000.

[15] I. P. Gent and T. Walsh. Csplib : a benchmark library fr constraints. *Springer*, pages 480–481, 1999.

[16] C. Gicquel, M. Minoux, and Y. Dallery. On the discrete lot-sizing and scheduling problem with sequence-dependent changeover times. *Operations Research Letters 37*, pages 32–36, 2009.

[17] C. Gicquel, N. Miègeville, M. Minoux, and Y. Dallery. Discrete lot sizing and scheduling using product decomposition into attributes. *Computers and Operations Research 36*, pages 2690–2698, 2009.

[18] D.E. Goldberg and J.H. Holland. Classifier systems and genetic algorithms. *The University of Michigan Press*, pages 235–282, 1989.

[19] J. F. Gonçalves and J. J. de Magalhaes Mendes. A hybrid genetic algorithm for the job shop scheduling problem. *European journal of Operational research*, pages 10–11, 2005.

[20] D. Hermawanto. Genetic algorithm for solving simple mathematical equality problem.

[21] J.H. Holland. Adaptation in natural and artificial system. *The University of Michigan Press*, 1975.

[22] V. R. Houndji. *Deux Problèmes de Planification de Production : Formulations et Résolution par Programmation en Nombres Entiers et par Programmation par Contraintes*. PhD thesis, Ecole Polytechnique de Louvain, 2013.

[23] V. R. Houndji. *Cost-based filtering algorithms for a capacitated lot sizing problem and the constrained arborescence problem*. PhD thesis, Université Catholique de Louvain & Université d'Abomey-Calavi, 2017.

[24] V. R. Houndji, P. Wolsey, R. Schaus, and Y. Deville. The stockingcost constraint. *Principles and Practice of Constraint Programming - CP*, pages 382–397, 2014.

[25] Wikimedia Foundation .Inc. Problème np-complet, 2017. fr.wikipedia.org.

[26] Wikimedia Foundation .Inc. Python (langage), 2017. fr.wikipedia.org.

[27] N. Jawahar, S. Aravindan, and G. Ponnambalam. A genetic algorithm for scheduling flexible manufacturing systems. *internation journal of advanced manufacturing technology*, pages 588–607, 1998.

[28] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing : an experimental evaluation ; part i, graph partitioning. *Operations Research 37*, pages 865–892, 2014.

[29] B. Karimi, S. Fatemi Ghomi, and J. Wilson. Single item lot sizing problems. *European journal of Operational Research 168*, 1 :1–16, 2006.

[30] A. Manne. Programming of economic lot sizes. *Management Science 4*, pages 115–135, 1958.

[31] L. A. Miller, A. J. Wolsey. Tight mip formulation for multi-item discrete lot-sizing problems. *Operations Research 51*, pages 557–565, 2003.

[32] S. Mirshekarian and Gürsel A. S. Experimental study of seeding in genetic algorithms with non-binary genetic representation. Seeding GA, 2014.

[33] R.K. Nayak and B.S.P. Mishra. Implementation of gpu using fine-grained parallel genetic algorithm. *International Journal of Computer Applications*, 2015.

[34] Y. Pochet and L. A. Wolsey. Production planning by mixed integer programming. *Springer Science and Business Media*, 2006.

[35] J. R. Povinelli and X. Feng. Improving genetic algorithms performance by hashing fitness values. 1998.

[36] G. Syswerda. Schedule optimization using genetic algorithm. *Handbook of Genetic algorithm*, pages 332–348, 1991.

[37] H. Wagner and T. Whitin. Dynamic version of the economic lot size model. *Management Science 5*, pages 89–96, 1958.

[38] C. Wolosewicz. *Approche intégrée en planification et ordonnancement de la production*. PhD thesis, Ecole Nationale Supérieure des Mines de Saint-Etienne, 2008.

[39] L. Wolsey. Solving multi-item lot-sizing problems with an mip solver using classification and reformulation. *Management Science 48*, pages 1587–1602, 2002.

Annexe A

Annexe 1

A.1 Classes et implémentation

Nous présentons ici la structure et quelques classes du programme (HFC-PGA) et nous détaillons les codes de quelques fonctions utilisées.

A.1.1 Classes

```
1   #Classe représentant un chromosome
2
3  class Chromosome(object):
4
5    mutationRate = 0
6    problem = 0
7    hashTable = {}
8
9    # Builder
10   def __init__(self):
11     # Fonction d'initialisation de chromosome
12
13   def init2(self, solution, itemsRank):
14     # Autre fonction d'initialisation de chromosome
15
16   def __lt__(self, chromosome):
17     # Fonction utilisée dans le comparaison entre deux chromosomes
18
19   def _get_fitnessValue(self):
20     # Fonction de calcul du fitness d'un chromosome
21
22   def _get_solution(self):
23     # Fonction de récupération du genotype d'un chromosome
24
25   def _get_itemsRanks(self):
26     # Fonction de récupération de la période de satisfaction d'une production
27
28   def _get_hashSolution(self):
29     # Fonction de récupération du hash du génotype
```

```
30
31    def _set_hashSolution(self, new_value):
32      # Fonction d'attribution du hash d'un genotype
33
34    def _set_itemsRanks(self, new_value):
35      # Fonction d'attribution de la liste des rangs des productions
36
37    def _set_solution(self, new_solution):
38      # Fonction d'attribution du genotype d'un chromosome
39
40    def _set_fitnessValue(self, new_value):
41      # Fonction d'attribution de la valeur de fitness a un chromosome
42
43    def __repr__(self):
44      # Fonction d'affichage d'un chromosome
45
46    def __eq__(self, chromosome):
47      # Fonction utilisée dans le comparaison entre deux chromosomes
48
49    def __ne__(self, chromosome):
50      # Fonction utilisée dans le comparaison entre deux chromosomes
51
52    def isFeasible(self):
53      # Fonction de vérification de la faisabilité d'un chromosome
54
55    def mutate(self):
56      # Fonction utilisée afin de faire muter un chromosome
57
58    def advmutate(self):
59      # Fonction utilisée comme fonction de recherche locale
60
61    def getFeasible(self):
62      # Fonction utilisée afin de rendre faisable un chromosome
63
64    def getCostof(cls, indice, item, rank,solution, secondIndice = -1):
65      # Fonction de détermination du cout d'un gene dans le génotype d'un chromosome
66
67    # Class' methods
68    getCostof = classmethod(getCostof)
69
70    # Properties
71    solution = property(_get_solution,_set_solution)
72    fitnessValue = property(_get_fitnessValue,_set_fitnessValue)
73    itemsRank = property(_get_itemsRanks, _set_itemsRanks)
74    hashSolution = property(_get_hashSolution, _set_hashSolution)
```

```
1     // Classe représentant un thread principal en HFC-PGA
2
3  class ClspThread(Thread):
4
5    listMainThreads = 0
6    NumberOfMigrants = 0
7    NbGenToStop = 0
8    crossOverRate = 0
9    MigrationRate = 0
10   NbMaxPopulation = 0
11   FITNESS_PADDING = 0
12
```

```
13    def __init__(self, threadId):
14      # Constructeur de la classe
15
16    def run(self):
17      # Fonction d'exécution du thread
18
19    def getPopImproved(self):
20      # Fonction utilisée dans l'amélioration de la qualité de la population initiale
21
22    def exploit(self, chromosome):
23      # Fonction utilisée afin d'exploiter un chromosome apres exploration
24
25    def initSearch(self, queue, parameter = "main"):
26      # Fonction d'execution de la formation de la population initiale
27
28    def sendMigrants(self):
29      # Fonction utilisee afin d'envoyer des chromosomes aux threads voisins
30
31    def receiveMigrants(self, chromosomes):
32      # Fonction utilisee dans la reception des chromosomes de threads voisins
33
34    def replace(self, chromosome):
35      #Code de la fonction ici
36
37    def getFitnessData(self):
38      # Fonction utilisee dans la procedure de Roulette Wheel
39
40    def mate(self, chromosome1, chromosome2):
41      # Fonction utilisee afin de reproduire deux chromosomes
42
43    def crossover(self, randValue1, randValue2):
44      # Fonction utilisee dans le croisement de deux chromosomes
45
46    def crossPopulation(self):
47      # Fonction utilisee afin de lancer le croisement a travers toute la population
48
49    def insert(self, chromosome):
50      # Fonction utilisee afin d'inserer un chromosome dans la population initiale
```

```
1  class GeneticAlgorithm:
2
3    # Class' variables
4    NbMaxPopulation = 25
5    mutationRate = 0.05
6    crossOverRate = 0.80
7    FITNESS_PADDING = 1
8    NumberOfMigrants = 1
9    MigrationRate = 0
10   nbMainThreads = 3
11   nbSlavesThread = 3
12   NbGenToStop = 7
13
14   # Builder
15   def __init__(self, inst):
16     # Constructeur de la classe
17
18   def start(self):
19     # Fonction utilisée afin de lancer l'exécution de l'algorithme génétique
```

```
20
21   def printResults(self):
22     # Fonction utilisée afin d'afficher les résultats
```

A.1.2 Fonctions

```
1    # Fonction d'évaluation d'un chromosome
2
3  def evaluate(cls, sol):
4
5      solution = list(sol)
6
7      fitnessValue = 0
8      # Calculation of all the change-over costs
9      itemsRank = [1] * Chromosome.problem.nbItems
10     i = 0
11     for gene in solution:
12       #print("gene : ", gene, " cost : ", Chromosome.getCostof(i, gene, itemsRank[gene-1], solution))
13       fitnessValue += Chromosome.getCostof(i, gene, itemsRank[gene-1], solution)
14       if gene != 0:
15         itemsRank[gene-1] += 1
16       i += 1
17
18 return fitnessValue
```

```
1      # Fonction de faisabilité d'un chromosome
2
3  def getFeasible(self):
4
5      #print(" In Chromosome 1 : ", self._solution)
6      #print(self._solution)
7
8      #if self.isFeasible() is False:
9
10     #print(" grid : ", grid)
11     copy_solution = list(self._solution)
12
13     # i make sure that the number of goods producted isn't superior to the number expected
14     i = 0
15     while i < Chromosome.problem.nbTimes:
16
17       if self._solution[i] != 0:
18
19         item = self._solution[i]
20         #print(" ok : ", self._solution, self._itemsRank)
21         #print(" item picked : ", item)
22         rank = self._itemsRank[i]
23         #print(i, item-1, rank-1, self.manufactItemsPeriods)
24         value = self.manufactItemsPeriods[item-1][rank-1]
25
26         if value == -1:
27           itemDemandPeriods = self.manufactItemsPeriods[item-1]
28           itemDemandPeriods[rank-1] = i
29           #print(" It isn't yet in the tab")
30           #print(" == -1 ", item, i, rank)
31
32         else:
```

```
33
34          #print(" != -1 ", item, i, rank)
35          #print(" It is already in the tab")
36          cost1 = Chromosome.getCostof(value, item, rank, copy_solution, i)
37          cost2 = Chromosome.getCostof(i, item, rank, copy_solution, value)
38
39          #print(" cost 1 : ", cost1, " cost2 : ", cost2)
40          if cost2 < cost1 :
41            itemDemandPeriods = self.manufactItemsPeriods[item-1]
42            itemDemandPeriods[rank-1] = i
43
44            #print(" cost2 < cost1 : ", value, item)
45            self._solution[value] = 0
46
47          else:
48            self._solution[i] = 0
49        i+=1
50
51      #print(" in middle getFeasible : ", self._solution, ", ", self._itemsRank)
52      #print()
53      # i make sure that the number of items producted isn't inferior to the number expected
54      i = 0
55      while i < Chromosome.problem.nbItems:
56
57        j = 0
58        nbmanufactItemsPeriods = len(self.manufactItemsPeriods[i])
59        while j < nbmanufactItemsPeriods:
60
61          if self.manufactItemsPeriods[i][j] == -1:
62            if j == 0:
63              lbound = 0
64            else:
65              lbound = self.manufactItemsPeriods[i][j-1]
66
67            zeroperiods = []
68            k = lbound+1
69            while k <= Chromosome.problem.deadlineDemandPeriods[i][j]:
70              if self._solution[k] == 0:
71                zeroperiods.append(k)
72              k+=1
73
74            #print("zeroperiods : ", zeroperiods)
75            nbZeroPeriods = len(zeroperiods)
76
77            if nbZeroPeriods > 0:
78
79              cost1 = Chromosome.getCostof(zeroperiods[0], i+1, j+1, copy_solution)
80              #print(" cost1 : ", cost1 )
81
82              k = 1
83              indice = zeroperiods[0]
84              while k < nbZeroPeriods:
85                cost2 = Chromosome.getCostof(zeroperiods[k], i+1, j+1, copy_solution)
86                #print(" cost2 : ", cost2 , zeroperiods[k])
87                if cost2 < cost1:
88                  #print(" cost2 < cost1 : ", cost1 , cost2 )
89                  indice = zeroperiods[k]
```

```
90                k+=1
91
92              self._solution[indice] = i+1
93
94              itemDemandPeriods = self.manufactItemsPeriods[i]
95              itemDemandPeriods[j] = indice
96
97            else:
98
99              # experimental code
100
101             # if there's no place to put this item, then i check all the other times in order to put this item there
102
103             lbound = 0
104             p = 1
105             for deadline in Chromosome.problem.deadlineDemandPeriods[i]:
106
107               zeroperiods = []
108               k = lbound
109               while k <= deadline:
110                 if self._solution[k] == 0:
111                   zeroperiods.append(k)
112                 k += 1
113               lbound = deadline + 1
114
115               nbZeroPeriods = len(zeroperiods)
116               if nbZeroPeriods > 0:
117
118                 cost1 = Chromosome.getCostof(zeroperiods[0], i+1, p, copy_solution)
119                 #print(" cost1 : ", cost1 )
120
121                 k = 1
122                 indice = zeroperiods[0]
123                 while k < nbZeroPeriods:
124                   cost2 = Chromosome.getCostof(zeroperiods[k], i+1, p, copy_solution)
125                   #print(" cost2 : ", cost2 , zeroperiods[k])
126                   if cost2 < cost1:
127                     #print(" cost2 < cost1 : ", cost1 , cost2 )
128                     indice = zeroperiods[k]
129                   k+=1
130
131                 self._solution[indice] = i+1
132
133                 itemDemandPeriods = self.manufactItemsPeriods[i]
134                 itemDemandPeriods[j] = indice
135
136 bre
137 ak
138             p += 1
139
140
141         j+=1
142       i+=1
```

Table des matières

Table des matières