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

1.1 Exemple de classification des modèles de dimensionnement de lots [37] 9
1.2 Diagramme d'un algorithme génétique standard [20] 15

2.1 Représentation génétique initiale 20
2.2 Représentation génétique adoptée 21
2.3 Illustration du croisement utilisé 23
2.4 Illustration de la méthode de mutation 25
2.5 Schéma d'un algorithme génétique de type "master-slave" [5] 28
2.6 Topologie de connexions utilisée 30
2.7 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] 30
2.8 Topologie utilisée en algorithme génétique parallèle de type "*fine-grained*" [4] . . . 32
2.9 Hiérarchie entre algorithmes génétiques parallèles de type *fine-grained* et de type *coarse-grained* [4] 32
2.10 Application de l'algorithme de recherche de meilleure solution à un chromosome 33

3.1 Performances comparées de CP et HCM-PGA en fitness de la meilleure solution trouvée 39
3.2 Performances comparées de CP et HCM-PGA en temps 40
3.3 Performances comparées de CP et HCM-PGA en fitness moyen des solutions trouvées 41
3.4 Performances comparées de CP et HFC-PGA en fitness de la meilleure solution trouvée 42
3.5 Performances comparées de CP et HFC-PGA en temps 42
3.6 Performances comparées de CP et HFC-PGA en fitness moyen des solutions trouvées 43
3.7 Performances comparées de SA et HCM-PGA en fitness de la meilleure solution trouvée 43
3.8 Performances comparées de SA et HCM-PGA en temps 44
3.9 Performances comparées de SA et HFC-PGA en fitness de la meilleure solution trouvée 45

3.10 Performances comparées de SA et HFC-PGA en temps 45
3.11 Performances comparées de HCM-PGA et HFC-PGA en temps 46
3.12 Performances comparées de HCM-PGA et HFC-PGA en fitness moyen des solutions trouvées 46

Liste des tableaux

3.1 Performances du HFC-PGA et CP sur 20 instances du PSP 38
3.2 Performances du HFC-PGA et SA sur 6 instances du PSP 39
3.3 Performances du HCM-PGA et CP sur 20 instances du PSP 40
3.4 Performances du HCM-PGA et SA sur 6 instances du PSP 41

Sommaire

Sommaire vi

Remerciements vii

Liste des algorithmes viii

Liste des sigles et abréviations ix

Résumé 1

Abstract 2

Introduction 3

1 État de l'art 5

Introduction 5

1.1 Le dimensionnement de lots en planification de production 5

1.2 Le *Pigment sequencing problem* (PSP) 9

1.3 Les algorithmes génétiques 13

Conclusion 17

2 Matériel et solutions 18

Introduction 18

2.1 Outils de test 18

2.2 Modèle et formulation utilisés 19

2.3 Aspects généraux aux deux méthodes de recherche proposées 20

2.4 Méthodes de recherche proposées 28

Conclusion 34

3 Résultats et Discussion **35**
Introduction 35
3.1 Données et paramètres de test 35
3.2 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques "fine-grained" et "coarse-grained" 37
3.3 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques "coarse-grained" et "master-slave" 37
3.4 Discussion 38
Conclusion 46

Conclusion et perspectives **47**

A Annexe 1 **52**
A.1 Classes et implémentation 52

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. 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

1 Algorithme génétique standard [19] 15

2 Processus de génération de la population initiale 22
3 Processus de génération des successeurs d'un noeud 22
4 Algorithme de croisement utilisé 24
5 Algorithme de mutation utilisé 25
6 Algorithme utilisé dans le processus d'évaluation d'un chromosome 26
7 Algorithme utilisé comme fonction de faisabilité 27
8 Principe des algorithmes génétiques parallèles de type "*fine-grained*" [32] 31
9 Algorithme de recherche locale d'une meilleure solution 33

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

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 travail présenté dans ce document vise à appliquer deux approches de résolution de problèmes basées sur les algorithmes génétiques à un problème typique de dimensionnement de lots : le *Pigment Sequencing Problem* (PSP). A cette fin, nous commençons par donner un aperçu général des problèmes de dimensionnement de lots en planification de production. Nous continuons en introduisant le PSP. Nous présentons les différents modèles et méthodes de résolution ayant été appliqués à ce problème au nombre desquels figurent l'approche basée sur la programmation par contraintes et celle basée sur le recuit simulé. Le document se poursuit en faisant un bref état de l'art de la méthode de résolution de problèmes que sont les algorithmes génétiques. Nous avons ainsi pu appliquer une catégorie d'algorithmes génétiques dits 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.

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

Abstract.

The work that this document exposes is about applying two approaches based on genetic algorithms on a typical lot sizing problem: the *Pigment Sequencing Problem* (PSP). In order to do so, we start giving a short view of the lot sizing problems in production planning. We go on doing the litterature review of the PSP. We give details on the different methods and models applied yet to this particular problem among which can be the approach based on the constraint programming and the one based on the simulated annealing. This document also does a short literature review of the solving method that are genetic algorithms. Furthermore, we applied a category of genetic algorithms which is the hierarchical and parallel genetic algorithms. The trials performed allow us to compare this last solving method to the ones applied in ealier researches.

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 [37].

Différentes versions 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 du problème de dimensionnement de lots. Cette version est connue sous le nom de *Pigment Sequencing Problem* (PSP) (Pochet et Wolsey [33]) 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.

Toutefois, la résolution d'un PSP se heurte comme tout problème de dimensionnement de lots 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. Dans ce contexte, la quête de méthodes de recherche toujours plus efficaces a guidé les différentes recherches effectuées[22] [8] dans le domaine.

Un problème d'optimisation tel que le problème de dimensionnement de lots se divise en deux phases : recherche des solutions admissibles puis recherche de la solution à coût optimal parmi ces dernières. L'usage d'un algorithme génétique est adapté à une exploration rapide et globale d'un espace de recherche de taille importante et est capable de fournir plusieurs solutions. Ainsi, l'utilisation des algorithmes génétiques pour des problèmes de dimensionnement de lots semble raisonnable dans le cas de grands problèmes où trouver la solution avec d'autres

algorithmes reste encore problématique en temps.

Contribution

Le *Pigment Sequencing Problem* (PSP) est un problème NP-Difficile d'optimisation combinatoire pour lequel les instances de taille moyenne peuvent être efficacement résolues en utilisant une formulation appropriée de programmation en nombre mixte [33]. Cependant, dans notre revue de littérature, aucun modèle basé sur les algorithmes génétiques n'a encore été proposé pour ce problème. Nous proposons donc deux méthodes de recherche basées sur les algorithmes génétiques pour le 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 [37] 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. Aussi, les algorithmes génétiques ont montré leur efficacité sur d'autres problèmes d'optimisation [19].

Introduction

Dans ce chapitre, nous présentons la revue de littérature des problèmes de dimensionnement de lots en planification de production dont fait parti le PSP. Nous exposons les méthodes de recherche déjà appliquées dans la résolution du PSP et soulignons les travaux effectués sur les algorithmes génétiques de même, nous montrons en quoi les algorithmes génétiques sont particulièrement adaptés à ces genres 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". Il est un important défi pour les entreprises industrielles car il a un fort impact sur leur performance en terme de qualité du service-client et des coûts d'exploitation.

Différentes recherches [3] ont été menées dans le but de classifier les problèmes de dimen-

sionnement de lots. Plusieurs critères ont pu être identifiés. Nous présentons dans cette section quelques critères de classification [37] 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* :** La planification peut être effectuée soit sur des périodes discrètes soit sur un horizon de temps continu. Dans le premier cas, la longueur des périodes peut être soit de petite taille (Small time buckets) correspondant à des heures ou jours, soit de grande taille (Big time buckets) correspondant à des jours ou semaines, soit de très grande taille (Very big time buckets) correspondant à des mois ou trimestres.

***Le nombre de niveaux* :** On distingue les problèmes à un niveau. Un état de l'art sur ce type de problème est proposé par Karimi et al. [28] . Lorsqu'il existe une relation entre les produits, on considère des problèmes à plusieurs niveaux.

***Le nombre de produits* :** Dans le cas où il n'y a pas de dépendance entre les produits, en particulier, s'il n'y a pas d'utilisation commune de la capacité, nous considérons des problèmes à un seul produit. Dans le cas inverse, on parle de problèmes à plusieurs produits. Un état de l'art sur les problèmes à un produit est proposé par Brahimi et al. [2].

***Les contraintes de capacité* :** Les contraintes de capacité incluent le nombre d'employés, la capacité des machines, la capacité de stockage, etc. Lorsque les contraintes de capacité sont introduites dans le modèle, elles rendent le problème plus difficile à résoudre, puisqu'elles lient les produits entre eux.

***Les demandes* :** Il existe plusieurs types de demandes qui peuvent être réparties selon trois groupes :

- les demandes constantes, les valeurs des demandes ne changent pas sur l'horizon de temps, ou demandes dynamiques , les valeurs varient au cours du temps.
- les demandes certaines, les valeurs sont connues à l'avance, ou demandes stochastiques , les valeurs sont basées sur des probabilités.
- les demandes indépendantes, lorsqu'un produit n'a pas besoin d'autres produits comme composants, ou demandes dépendantes, lorsqu'il existe une relation entre les produits.

***Les coûts et temps de lancement ou préparation (setup)* :** Une ressource peut exécuter des produits de types différents. Ainsi, il est parfois nécessaire de reconfigurer celle-ci à chaque

changement de produits. Les coûts et temps induits par le lancement de la ressource sont souvent importants car élevés et longs.

La complexité des problèmes de dimensionnement de lots varie fortement sous l'influence des différents facteurs comme le nombre de produits, le nombre de niveaux, les contraintes de capacité, etc. Nous avons choisi dans cette étude de décrire les problèmes de dimensionnement de lots selon la longueur de la période de l'horizon de temps. Nous allons donc nous intéresser aux problèmes à courtes et longues périodes.

1.1.2 Classes de problèmes de dimensionnement de lots

1.1.2.1 Problèmes 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. Pour une étude détaillée de tous ces problèmes, nous pourrons nous référer à Drexl and Kimms [10]. Ces différents types de problèmes sont donc :

Discrete lot-sizing and scheduling problem : Ce problème noté DLSP, est initialement formulé par Fleischmann [12]. La principale hypothèse de ce problème est qu'au plus un article peut être produit par période. Si un article est fabriqué sur une période, alors toute la capacité disponible sur cette période sera utilisée. Généralement, les coûts de lancement sont pris en compte seulement lorsqu'un nouveau lot commence et non à chaque période.

Continuous setup lot-sizing problem : Ce problème noté CSLP est similaire au précédent problème. La différence réside dans le fait que lorsqu'un article est produit sur une période, on utilise la capacité nécessaire pour produire cet article et non la capacité entière comme pour le problème précèdent.

Proportional lot-sizing and scheduling problem : Dans ce problème, que l'on notera PLSP, la capacité restante pour une période donnée est réutilisée pour produire un second article. Ce problème a été étudié par Drexl et Haase [11] et Kimms et Drexl [10]. Il existe des extensions pour ce problème, par exemple, le cas à plusieurs niveaux et plusieurs machines.

General lot-sizing and scheduling problem : Ce problème noté GLSP, est plus général que les problèmes précédents, puisque le nombre d'articles à produire par période n'est pas restrictif. Il est étudié par Fleischmann et Meyr [13]. Il existe des extensions de ce problème, par exemple en intégrant des temps de lancement de production dépendant de la séquence des lots à produire ([14]).

1.1.2.2 Problèmes à longues périodes

Les problèmes à longues périodes (*Big time bucket problems*) sont basés sur des horizons de l'ordre de quelques jours à quelques semaines et sont caractérisés par le fait que plus d'un produit peut être fabriqué par période. Nous proposons dans cette section, une étude restreinte de ces problèmes.

Les problèmes à un niveau et un produit :

Ces problèmes ne correspondent pas vraiment à la réalité mais reçoivent beaucoup d'attention et d'intérêt puisque les méthodes mises en œuvre pour les résoudre sont généralisées dans le cas des problématiques à plusieurs niveaux. Les premiers travaux sur cette problématique ont été conduits par Manne [29] et Wagner et Whitin [36]. Leur méthode de résolution est utilisée dans plusieurs concepts pour résoudre des problèmes plus complexes. Brahimi et al. [2] ont proposé un état de l'art sur ces problématiques.

Les problèmes à un niveau et plusieurs produits :

Le problème de dimensionnement de lots avec contraintes de capacité est considéré comme un problème complexe puisque la contrainte de capacité engendre un lien entre les différents produits. Dans le cas où la capacité est considérée comme infinie, le problème de dimensionnement de lots à N produits est réductible à N problèmes à un produit et sans capacité, chacun solvable en temps polynomial.

Les problèmes à plusieurs niveaux :

Ces problèmes sont caractérisés par le fait que les produits finis sont fabriqués à partir de produits intermédiaires. Ainsi, les demandes dépendantes (i.e. les demandes entre les produits) et les demandes indépendantes (i.e. les demandes arrivant de l'extérieur) sont prises en compte dans ce type de problème.

Un exemple typique d'un problème de "*big bucket*" est le "Capacited Lot Sizing Problem" (CLSP) , où différents articles peuvent être produit sur une même ressource en une seule période. Le CLSP classique consiste à déterminer le coût et le temps de production des articles dans l'horizon de planification : le résultat est un plan de production qui donne pour chaque période de planification, la quantité (*lot size*) de chaque article qui doit être produit. Le CLSP requiert que la ressource soit préparée pour un article donné dans la période où il est produit. La figure 1.1 présente un exemple de classification des modèles de dimensionnement de lots.

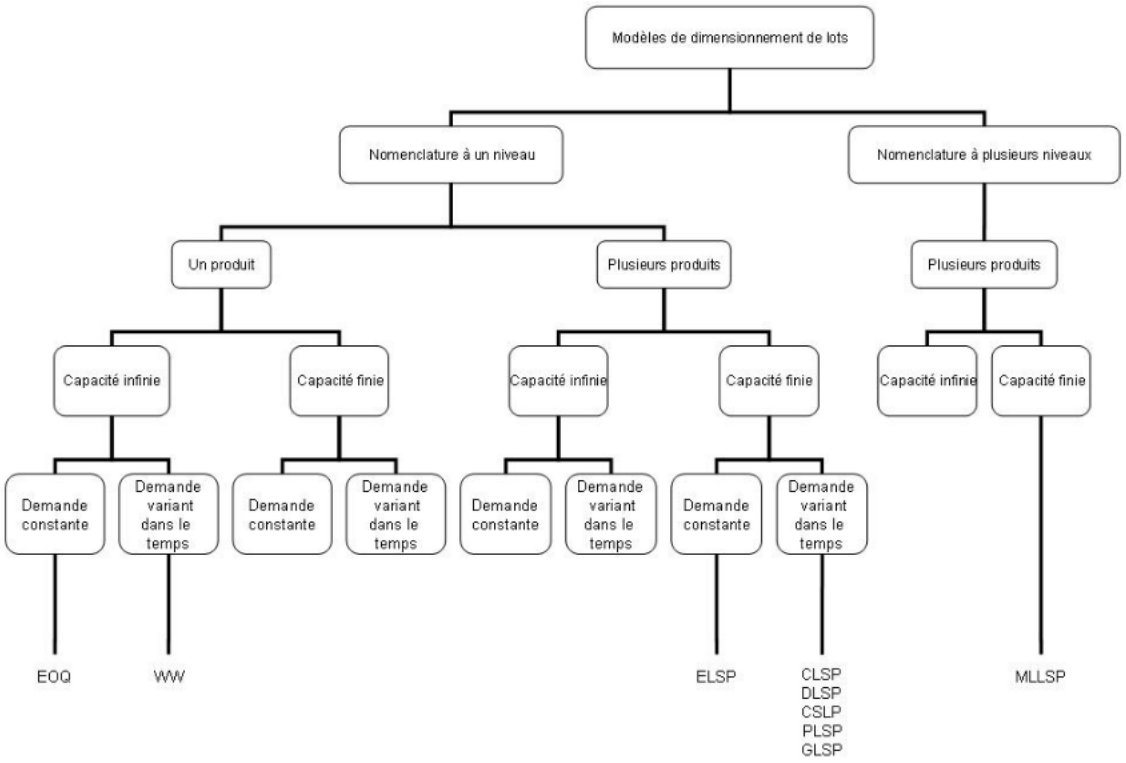

FIGURE 1.1 – Exemple de classification des modèles de dimensionnement de lots [37]

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 sous-section une brève revue de littérature des différents travaux effectués autour du PSP.

Miller et Wolsey [30] 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 [33].

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 [38]. 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 *Stocking-*

Cost 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 tout en prenant en compte les autres techniques de décompositions 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 implémenté une approche qui prend en charge à la fois la faisabilité de la solution et l'optimisation du coût. Dans le but d'illustrer leur méthode, 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 une 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.

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 [33], 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.

1.2.3.1 Modèle MIP 1

Le modèle MIP 1 [33] tel qu'exposé par Pochet et Wolsey se présente comme suit :

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

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

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

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

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

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

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

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 et est exprimé par la contrainte (1). La contrainte (2) rappelle qu'il n'y a pas de stock initial. La contrainte (3) exprime la règle de la conservation de flot. La contrainte (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 (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 (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 $ch_t^{i,j}$ serait égale 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 8 : chaque demande doit être satisfaite avant sa date limite ;
- Equation 9 : Puisque la capacité d'une machine est limitée à 1, chaque demande doit être satisfaite à différente période.

Il est possible de modeler 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 10 : il garantit l'existence d'un circuit hamiltonien ;
- Equation 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. Comme toujours 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. ont utilisé un mécanisme de *cut-off* (Johnson et al. [27]). 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 (GAs) sont des algorithmes d'exploration fondés sur les mécanismes de la sélection naturelle et de la génétique [21] [18]. Ils utilisent à la fois les principes de la survie des structures les mieux adaptées, et les échanges d'informations aléatoires, parfois guidés, pour former un algorithme d'exploration qui possède certaines des caractéristiques de l'exploration humaine. Ils ont été proposés pour la première fois et sont devenus populaires à travers les travaux de John Holland au début des années 1970, et particulièrement son livre *Adapation in Natural an Artificial Systems* (1975). Holland présenta les GAs comme une abstraction de l'évolution biologique et des concepts théoriques à l'adaptation pour les GAs dans son

livre. Il introduisit un formalisme afin de prédire la qualité de la prochaine génération, plus connue comme le théorème des schémas de Holland.

1.3.1 Concepts de base

Les AGs constituent une classe de stratégies de recherche réalisant un compromis entre l'exploration et l'exploitation. Ils représentent des méthodes qui utilisent un choix aléatoire comme outil pour guider une exploration intelligente dans l'espace des paramètres codés. Ce sont des algorithmes itératifs de recherche globale dont l'objectif est d'optimiser une fonction prédéfinie appelée fonction coût ou fonction « fitness ». Les algorithmes génétiques emploient un vocabulaire emprunté à la génétique naturelle. Ils travaillent sur un ensemble d'individus appelé population. Un individu a deux représentations appelées phénotype et génotype. Le phénotype représente une solution potentielle du problème à optimiser en utilisant la formulation originale du problème. Le génotype donne une représentation codée d'une solution potentielle sous la forme d'un chromosome. Un chromosome est formé de gènes disposés en une succession linéaire et chaque gène peut prendre plusieurs valeurs appelées allèles. Par exemple, un chromosome se compose d'une succession de 0 et de 1 (c.-à-d. une chaîne binaire), et la valeur pour une certaine position correspond à "on" (la valeur = 1) ou à "off" (la valeur = 0) d'un certain dispositif. Des formes plus compliquées, telles qu'un ordre des symboles et une permutation des alphabets, sont choisies pour décrire les chromosomes du problème à optimiser. Chaque individu a une fonction objectif f (fonction « fitness ») qui mesure l'adaptation de l'individu à son environnement local. La théorie darwinienne indique que, parmi des individus d'une population, celui qui est le mieux adapté à l'environnement local a le plus de chance de survivre et d'avoir un plus grand nombre de descendants : c'est la règle de la « survie du plus fort ». Ainsi, la fonction objectif f du problème d'optimisation joue le rôle d'un critère d'adaptation. Un des points les plus importants des algorithmes génétiques est la flexibilité dans la fonction objectif.

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.

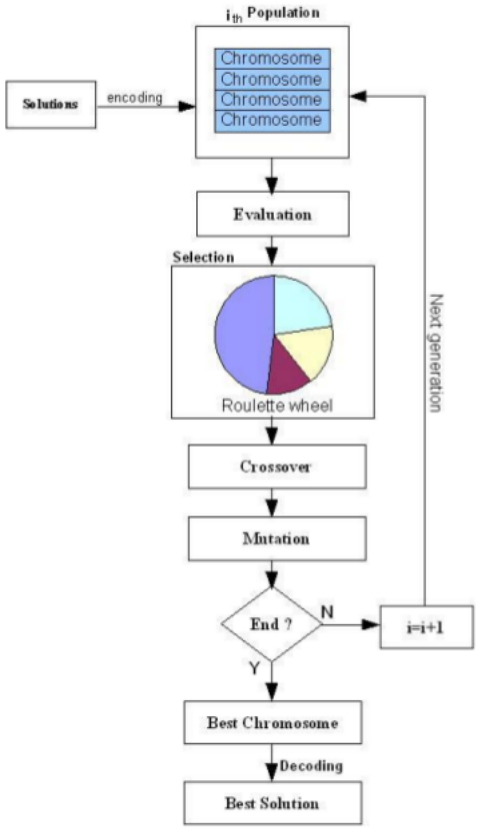

FIGURE 1.2 – Diagramme d'un algorithme génétique standard [20]

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

La sélection est un processus dans lequel des individus d'une population sont choisis selon les valeurs de leur fonction coût ou « fitness » pour former une nouvelle population. Les individus évoluent par des itérations successives de la sélection, appelées générations. Chaque individu est sélectionné proportionnellement à sa fonction « fitness », donc, un individu avec une fonction « fitness » plus élevée aura plus de chance d'être sélectionné qu'un autre avec une valeur de « fitness » inférieure. Cette fonction peut être envisagée comme une mesure de profit ou de qualité qu'on souhaite maximiser. Un opérateur simple de sélection est la technique de la roulette pondérée où chaque individu d'une population occupe une surface de la roulette proportionnelle à la valeur de sa fonction « fitness ». Pour la reproduction, les candidats sont sélectionnés avec une probabilité proportionnelle à leur « fitness ». Pour chaque sélection d'un individu, une simple rotation de la roue donne le candidat sélectionné. Cependant cette sélection n'est pas parfaite. En effet, le risque de favoriser un individu ou un petit ensemble d'individus constitue un inconvénient qui risque d'appauvrir la diversité de la population.

1.3.3.2 L'opérateur de croisement

Le croisement est un opérateur de recombinaison. Les individus d'une population sont couplés au hasard par paires représentant les parents. Chaque paire d'individus subit le croisement décrit comme suit : le croisement opère sur les génotypes (c.-à-d. les chromosomes) de deux individus appelés parents. Il produit de nouveaux individus (généralement deux) appelés enfants dont les gènes sont hérités de l'un ou/et de l'autre parent. Ceci peut être fait en dédoublant chacun des deux chromosomes dans des fragments et en les recombinant pour former de nouveaux chromosomes.

Le croisement à un point : Si le génotype est une chaîne binaire de longueur n. Le croisement à un point place un point de croisement au hasard. Un enfant prend une section avant le point de croisement d'un parent et prend l'autre section après le point de croisement de l'autre parent puis recombine les deux sections pour former une nouvelle chaîne binaire. L'autre enfant se construit inversement.

Le croisement à deux points : Le croisement à deux points place deux points de croisement au hasard, et prend une section entre les points d'un parent et les autres sections en dehors des points de l'autre parent puis les recombine.

Le croisement uniforme : Ce type de croisement a été proposé par Syswerda [35]. Il consiste à choisir avec la même probabilité un allèle de l'un ou de l'autre parent, pour transmettre sa valeur à la même position, aux enfants.

1.3.3.3 L'opérateur de mutation

La mutation opère sur le génotype d'un seul individu. Elle correspond, dans la nature, à une « erreur » qui se produit quand le chromosome est copié et reproduit. Dans une approche numérique, pour une chaîne binaire, elle consiste par exemple à faire pour un allèle un échange entre le « 0 » et le « 1 ». Si des copies exactes sont toujours garanties, alors le taux de mutation est égal à zéro. Cependant, dans la vie réelle, l'erreur de copie peut se produire dans diverses circonstances comme sous l'influence d'un bruit. La mutation change les valeurs de certains gènes avec une faible probabilité. Elle n'améliore pas, en général, les solutions, mais elle évite une perte irréparable de la diversité.

1.3.4 Application des algorithmes génétiques aux problèmes d'optimisation

Les algorithmes génétiques ont été largement utilisés ces dernières années [26]. L'utilisation des AGs dans de nombreux domaines a fait ses preuves, notamment dans des problèmes combinatoires tels que les problèmes d'ordonnancement [9] et les problèmes de collecte et de distribution. Les problèmes d'ordonnancement d'un atelier classique de type Job-Shop (JSP) ont notamment été largement étudiés et résolus par les AGs [1]. D'autres algorithmes hybrides ont été aussi proposés [6]. La difficulté principale dans la résolution de ces types de problèmes résulte dans la façon avec laquelle ils sont représentés sous forme algorithmique. Dans cette phase, la définition du chromosome représente le point le plus important dans la recherche génétique. Plusieurs approches de représentation et différents types d'opérateurs d'AGs ont été proposés, pour résoudre ces problèmes.

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 le partie suivante, nous décrivons les outils de test, les méthodes de résolution proposées ainsi que les algorithmes associés.

Chapitre 2

Matériel et solutions

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. De plus, 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 nos outils de test, 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 [25].

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.

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

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

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

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

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

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

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

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 10, nous pouvons représenter un chromosome conformément à la figure 2.1.

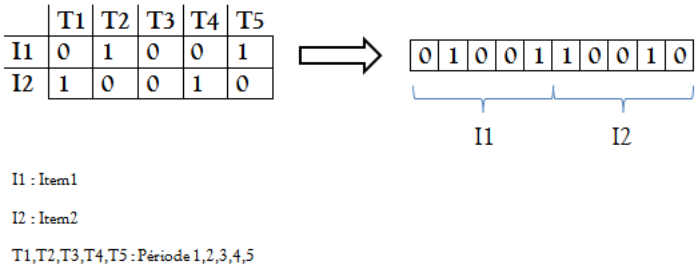

FIGURE 2.1 – Représentation génétique initiale

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 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 cohérente et facilement manipulable. Cette représentation génétique est présentée à la figure 2.2 et étudiée par Mirshekarian et al. [31].

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 déterministes et informés munis de fonctions d'évaluation suffisamment précises menant à des bonnes solutions. Notre approche a été donc d'utiliser une stratégie de recherche en l'occurrence le "Hill Climbing" 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 2 détaille le processus de génération de la population initiale :

Algorithme 2 : 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

    $noeudFils \leftarrow noeud.getSuccessors()$

    noeudFils.trier(decroissant)

    queue.ajouter(noeudFils)

    si *queue est vide* **alors**

      retourner populationInitiale

    fin

    $noeud \leftarrow queue.dernier()$

  fin

fin

retourner populationInitiale

L'algorithme 3 décrit la génération des successeurs d'un noeud donné.

Algorithme 3 : Processus de génération des successeurs d'un noeud

Données : noeud

Résultat : liste des successeurs constituée

$successeurs \leftarrow []$

pour *item* $\leftarrow$ *nbItems* **à** *1* **faire**

  si *item.dernierDeadline* $>=$ *noeud.currentPosition* **alors**

    noeudFils = copie(noeud)

    noeudFils.currentPosition -= 1

    successeurs.ajouter(noeudFils)

  fin

fin

retourner successeurs

2.3.3 Opérateurs génétiques

2.3.3.1 Sélection

La sélection est le processus qui consiste à choisir dans la génération actuelle, les chromosomes ou individus qui seront reproduits afin de former la prochaine génération. Différentes méthodes de sélection existent [4]. On distingue entre autres, la sélection par tournoi et la sélection par "*roulette wheel*".

Dans notre étude, nous choisissons de nous intéresser à la plus connue et commune d'entre les méthodes de sélection : le *"Roulette Wheel"*. Dans la sélection par "*roulette wheel*", les individus se voient attribués une probabilité d'être sélectionnés. Cette probabilité est directement proportionnelle à leur fonction d'évaluation. Les individus sont donc choisis aléatoirement en se basant sur leur probabilité et se reproduisent en générant des individus "*fils*". Ces "*fils*" sont ainsi de nouvelles solutions au problème et forme 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. Cette forme de croisement reste une des plus simples et répandues. La figure 2.3 présente une illustration du croisement appliqué à l'instance de PSP introduite à la page 10.

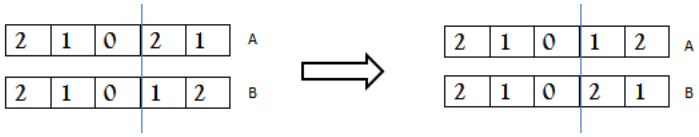

FIGURE 2.3 – Illustration du croisement utilisé

Le croisement se fait ainsi après la troisième période. Le croisement peut engendrer des individus qui, contrairement à la figure 2.3, ne respectent pas les contraintes de système en terme de *shortage* ou de *backlogging*. Il faut alors rendre ces individus à nouveau faisables avant de procéder à une quelconque mutation. L'algorithme 4 présente le croisement implémenté dans le cadre de cette étude. Notons que notre algorithme a une complexité linéaire en $O(n)$

Algorithme 4 : 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**
    $gene1 \leftarrow getGene(parent1, i)$
    $gene2 \leftarrow getGene(parent2, i)$
    si $i \leq$ *indice_gene* **alors**
      $child1.ajouterGene(gene)$
      $child2.ajouterGene(gene)$
    sinon
      $child2.ajouterGene(gene)$
      $child1.ajouterGene(gene)$
    fin
  fin
fin
retourner *child1, child2*

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*". A chaque gène, on attribue une chance de muter. Dans le cas, où la mutation se produit, la valeur du gène muté change de période. Un visuel de la mutation est présenté à la figure 2.4. La mutation est un élément vital de la garantie d'une diversité au sein de la population. L'algorithme 5 détaille le processus utilisé afin de faire muter un chromosome.

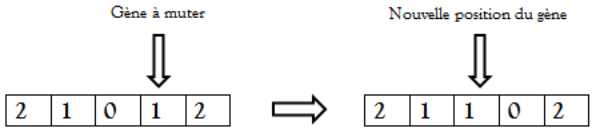

FIGURE 2.4 – Illustration de la méthode de mutation

Algorithme 5 : Algorithme de mutation utilisé

Données : chromosome, seuil_probabilite
Résultat : chromosome

$probabilite \leftarrow random(1, 99)$
si *probabilite* $\leq$ *seuil_probabilite* **alors**
  $nbGenes \leftarrow NombreDeGenes(chromosome)$
  $indice\_gene \leftarrow random(1, nbGenes)$
  $gene1 \leftarrow getGene(chromosome, indice\_gene)$
  $//Le\ nombre\ de\ genes\ du\ parent1\ est\ le\ meme\ que\ celui\ du\ parent2$
  pour $i \leftarrow$ *indice_gene* **à** *0* **faire**
    $gene2 \leftarrow getGene(chromosome, i)$
    si *deadline(gene2)* $>=$ *indice_gene* **alors**
      $deplacer(gene1, gene2, chromosome)$
      $arret$
    fin
  fin
fin
retourner *chromosome*

2.3.4 Évaluation

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

L'algorithme 6 explicite le processus suivi afin d'évaluer un chromosome. Notre algorithme possède une complexité linéaire en O(n). 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 6 : Algorithme utilisé dans le processus d'évaluation d'un chromosome

Données : chromosome, cout_stockage, cout_transition
Résultat : eval

```
eval ← 0
//On calcule le cout de stockage de chaque production
pour gene in chromosome faire
    si gene.value != 0 alors
        date_limite ← getDateLimite(gene)
        temp ← (date_limite − gene.periode) ∗ cout_stockage(gene.value)
        evaluation ← evaluation + temp
    fin
fin
//On calcule le cout de transition de entre deux productions
pour gene in chromosome faire
    si gene.value != 0 alors
        next_gene ← getNextGene(chromosome)
        si transition(gene, next_gene) est vrai alors
            temp ← cout_transition(gene, next_gene)
            evaluation ← evaluation + temp
        fin
    fin
fin
retourner evaluation
```

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 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. Afin d'y parvenir, nous avons réduit le surplus de production dans le cas d'un surplus et augmenter la quantité d'articles produits dans le cas d'un manque de production. L'algorithme 7 est celui utilisé dans cet objectif.

Algorithme 7 : Algorithme utilisé comme fonction de faisabilité

```
Données : chromosome, deadlines
Résultat : eval

// On commence par reduire le surplus de production ;
pour i ← 1 à Nombre_Periodes faire
    item ← chromosome.getItem(i) ;
    si estEnSurplus(item, deadline(i)) alors
        supprimerProduction(item) ;
    fin
fin
// On compense le manque de production ;
pour item in liste_items faire
    item_deadlines ← deadlines(item) ;
    pour deadline in item_deadlines faire
        si nonProduit(dealine) alors
            produire(item);
        fin
    fin
fin
```

2.4 Méthodes de recherche proposées

Les algorithmes génétiques sont des techniques de recherche qui ont été utilisées avec succès dans la résolution de problèmes dans différentes disciplines. Cependant, la recherche permanente de performance dans l'exécution des algorithmes et le développement constant d'ordinateurs toujours plus performants et parallèles ont conduit à l'émergence d'un type d'algorithmes génétiques plus performants que les algorithmes génétiques standards. A propos des algorithmes génétiques parallèles, on distingue généralement trois classes : les *master-slave parallel genetic algorithms* (M-PGA), les *coarsed-grained parallel genetic algorithms* (C-PGA) et les *fine-grained parallel genetic algorithms* (F-PGA). Les *master-slave parallel genetic algorithms* (M-PGA) 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 processus qui évaluent la population sont les esclaves. La figure 2.5 montre un schéma de l'algorithme génétique parallèle *master-slave*. Nous nous intéressons dans notre étude aux deux autres types, plus compliqués mais également plus intéressants dans leur fonctionnement et performances.

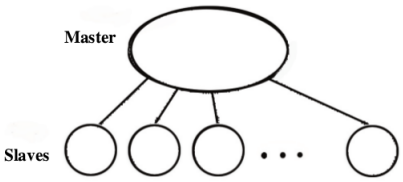

FIGURE 2.5 – Schéma d'un algorithme génétique de type "master-slave" [5]

2.4.1 Algorithmes génétiques parallèles de type "*coarse-grained*"

Les algorithmes génétiques de type "*coarse-grained*" ou encore "*modèle de l'île*" consiste en un ensemble de sous-populations qui échangent des individus de manière fréquente. Il s'agit probablement du type d'algorithmes génétiques le plus populaire bien qu'il nécessite de contrôler beaucoup de paramètres. La complète compréhension des effets de ces paramètres sur la qualité et la vitesse de recherche nous échappe encore. 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 effectué entre le temps de calculs et le temps de communication. L'implémentation des algorithmes génétiques parallèles soulève généralement trois problématiques principalement sur la migration qui est l'échange d'individus entre processus. Il s'agit : de la fréquence de migration, du nombre d'individus à échanger, de la

topologie des connexions entre processus et de la méthode d'intégration des migrants.

2.4.1.1 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.

Afin d'implémenter l'algorithme, 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 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. Dans la suite de notre étude, nous avons choisi d'échanger les meilleurs individus.

2.4.1.3 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. La topologie utilisée a été celle densément connectée présentée à la figure 2.6

2.4.1.4 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

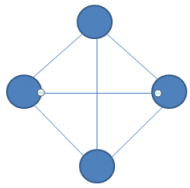

FIGURE 2.6 – Topologie de connexions utilisée

"*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. Un remplacement compétitif a été appliqué 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.

2.4.1.5 Algorithme hiérarchique entre *coarse-grained* et *master-slave*

Une fois, ces problématiques abordées, la question de l'implémentation rapide des algorithmes génétiques parallèles de type "*coarse-grained*" revient. Une des réponses serait en effet de combiner ce type d'algorithmes génétiques parallèles avec un autre type, le "*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 2.7. Il s'agit ainsi de cette combinaison que nous avons utilisée pour notre implémentation du programme et nos tests.

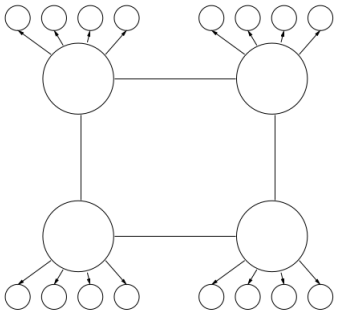

FIGURE 2.7 – Algorithmes génétiques parallèles et hierarchiques avec au niveau supérieur un algorithme génétique parallèle de type *coarse-grained* et au niveau inférieur un algorithme génétique parallèle de type *master-slave* [4]

2.4.2 Algorithmes génétiques parallèles de type "*fine-grained*"

les algorithmes génétiques parallèles de type "*fine-grained*" 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.

2.4.2.1 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]. La figure 2.8 présente la topologie utilisée afin d'implémenter l'algorithme génétique parallèle de type *"fine-grained"*.

2.4.2.2 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 8.

Algorithme 8 : Principe des algorithmes génétiques parallèles de type "*fine-grained*" [32]

```
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
```

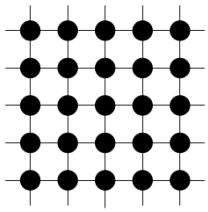

FIGURE 2.8 – Topologie utilisée en algorithme génétique parallèle de type "*fine-grained*" [4]

2.4.2.3 Algorithme hiérarchique entre *fine-grained* et *coarse-grained*

Des chercheurs [4] ont combiné deux des méthodes de parallélisation produisant des algorithmes génétiques parallèles hiérarchiques. Certains de ces nouveaux algorithmes ajoutent un nouveau degré de complexité à des 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 2.9. Il s'agit en effet de notre deuxième proposition de méthode de recherche des instances de PSP que nous avons implémentée.

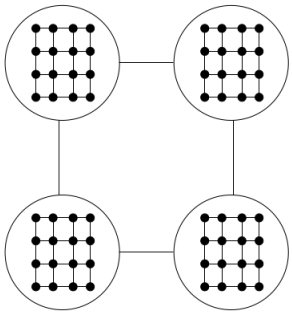

FIGURE 2.9 – Hiérarchie entre algorithmes génétiques parallèles de type *fine-grained* et de type *coarse-grained* [4]

2.4.3 Autres algorithmes implémentés

Au cours de notre étude de l'art et en implémentant nos deux méthodes de résolution proposées que sont les algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *master-slave* et les algorithmes génétiques parallèles et hiérarchiques *coarse-grained* et *fine-grained*, nous avons pu découvrir différentes fonctions annexes aux algorithmes génétiques qui permettent d'en améliorer la qualité. Il s'agit des méthodes d'hybridation [19] et de table de hashage [34].

2.4.3.1 Hybridation

L'hybridation 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.

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 une complexité polynomiale en $O(n^2)$

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 page 10 et qu'on applique cet algorithme au problème, on obtient la figure 2.10.

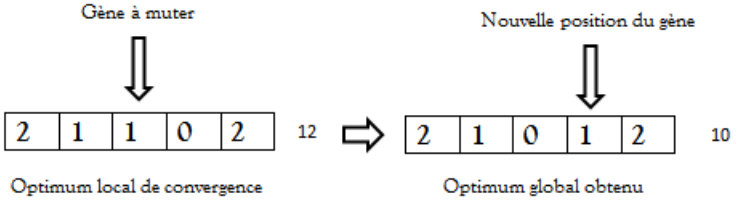

FIGURE 2.10 – Application de l'algorithme de recherche de meilleure solution à un chromosome

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 génétiques. En utilisant une table de hash afin de stocker les chromosomes récemment évalués, une amélioration significative des performances peut être constatée. Nous avons donc utilisé un dictionnaire ou tableau associatif afin de stocker ces derniers.

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 optimales sinon très proches de ces dernières en un temps plus court. Elles ne parviennent cependant pas à en faire autant sur celles proposées par Ceschia [8].

Introduction

Dans ce chapitre, nous testons les théories émises et tentons d'analyser 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 le 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 donc notre approche de solution à 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 nos 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 à côté de ceux obtenus 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.

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 sont consignés dans les tableaux 3.3 et 3.4 ; et sont comparés respectivement à ceux obtenus en programmation par contrainte et en recuit simulé.

[1]Coût de la solution optimale
[2]Temps en seconds de l'approche CP
[3]Temps moyen en seconds 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

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 dans le cadre de notre étude.

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

3.4.1 HCM-PGA et Approche CP

L'approche CP dans son application au PSP s'est avérée efficace. 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)

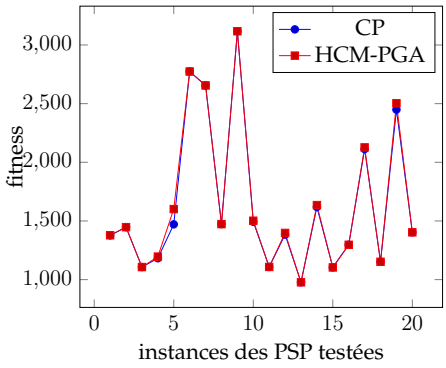

FIGURE 3.1 – Performances comparées de CP et HCM-PGA en fitness de la meilleure solution trouvée

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 avons pu constater 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 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

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

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.

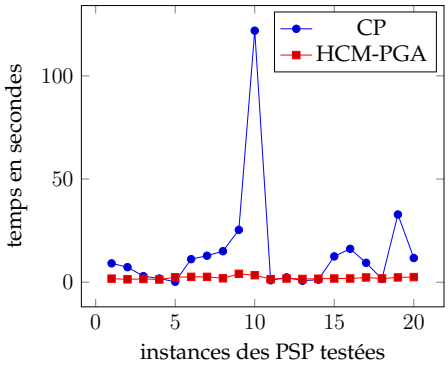

FIGURE 3.2 – Performances comparées de CP et HCM-PGA en temps

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

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

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.

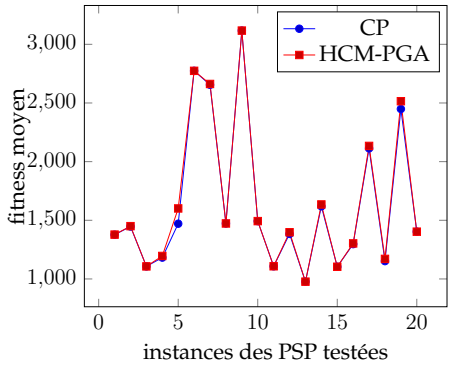

FIGURE 3.3 – Performances comparées de CP et HCM-PGA en fitness moyen des solutions trouvées

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 performe aussi bien que l'approche de référence c'est à dire 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.

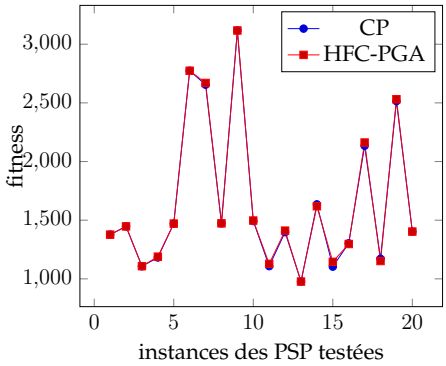

FIGURE 3.4 – Performances comparées de CP et HFC-PGA en fitness de la meilleure solution trouvée

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.

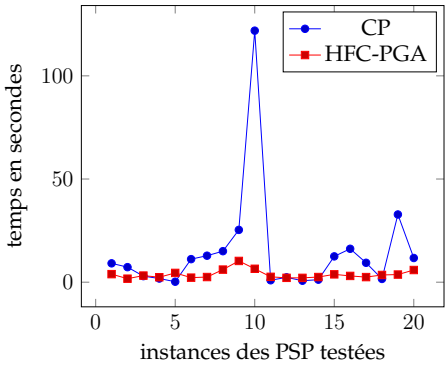

FIGURE 3.5 – Performances comparées de CP et HFC-PGA en temps

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.

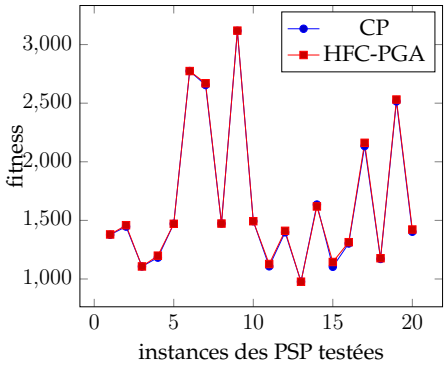

FIGURE 3.6 – Performances comparées de CP et HFC-PGA en fitness moyen des solutions trouvées

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)

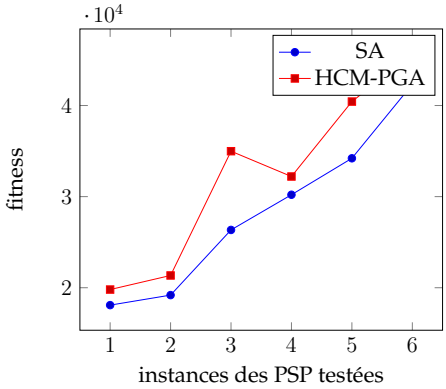

FIGURE 3.7 – Performances comparées de SA et HCM-PGA en fitness de la meilleure solution trouvée

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.

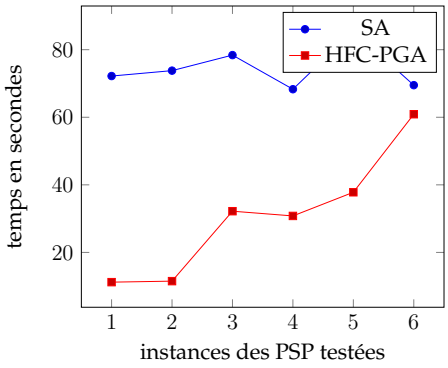

FIGURE 3.8 – Performances comparées de SA et HCM-PGA en temps

Nous remarquons 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, 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 le montre la figure 3.10.

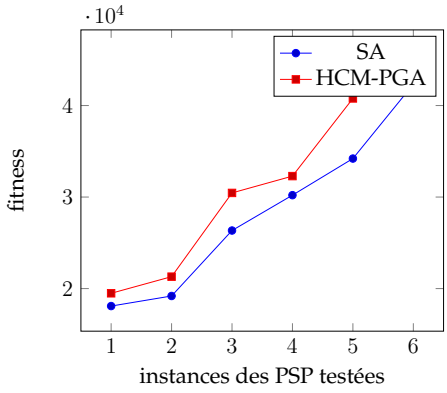

FIGURE 3.9 – Performances comparées de SA et HFC-PGA en fitness de la meilleure solution trouvée

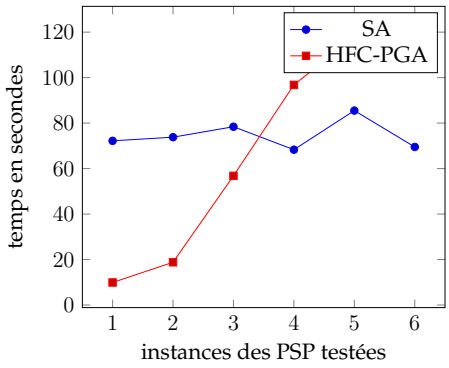

FIGURE 3.10 – Performances comparées de SA et HFC-PGA en temps

3.4.5 HCM-PGA et HFC-PGA

Dans cette sous-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.

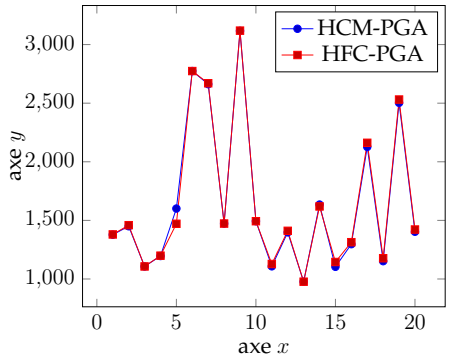

FIGURE 3.11 – Performances comparées de HCM-PGA et HFC-PGA en temps

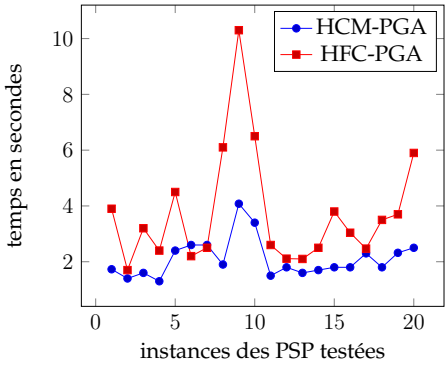

FIGURE 3.12 – Performances comparées de HCM-PGA et HFC-PGA en fitness moyen des solutions trouvées

Conclusion

Les expérimentations et 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

Dans ce travail, 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 PSP. Le *Pigment Sequencing Problem* (PSP) fait partie de ces types de problèmes plus précisément du DLSP. Plusieurs méthodes de résolution du PSP ont été testées au cours de récentes recherches. Il s'agit des approches 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 l'occurrence 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.

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 ou encore penser à des moyens de croisement plus intelligents qui améliorent la population. Et enfin, l'application des algorithmes génétiques parallèles et hiérarchiques à des problèmes encore plus complexes à plusieurs machines ou à plusieurs niveaux.

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, 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. Python (langage), 2017. fr.wikipedia.org.

[26] 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.

[27] 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.

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

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

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

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

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

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

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

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

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

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

[38] 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

Dans cette section, 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     #Code de la fonction ici
12
13   def init2(self, solution, itemsRank):
14     #Code de la fonction ici
15
16   def __lt__(self, chromosome):
17     #Code de la fonction ici
18
19   def _get_fitnessValue(self):
20     #Code de la fonction ici
21
22   def _get_solution(self):
23     #Code de la fonction ici
24
25   def _get_itemsRanks(self):
26     #Code de la fonction ici
27
28   def _get_hashSolution(self):
29     #Code de la fonction ici
```

```
30
31    def _set_hashSolution(self, new_value):
32      #Code de la fonction ici
33
34    def _set_itemsRanks(self, new_value):
35      #Code de la fonction ici
36
37    def _set_solution(self, new_solution):
38      #Code de la fonction ici
39
40    def _set_fitnessValue(self, new_value):
41      #Code de la fonction ici
42
43    def __repr__(self):
44      #Code de la fonction ici
45
46    def __eq__(self, chromosome):
47      #Code de la fonction ici
48
49    def __ne__(self, chromosome):
50      #Code de la fonction ici
51
52    def isFeasible(self):
53      #Code de la fonction ici
54
55    def mutate(self):
56      #Code de la fonction ici
57
58    def advmutate(self):
59      #Code de la fonction ici
60
61    def getFeasible(self):
62      #Code de la fonction ici
63
64    def getCostof(cls, indice, item, rank,solution, secondIndice = -1):
65      #Code de la fonction ici
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      #Code de la fonction ici
15
16    def run(self):
17      #Code de la fonction ici
18
19    def getPopImproved(self):
20      #Code de la fonction ici
21
22    def exploit(self, chromosome):
23      #Code de la fonction ici
24
25    def initSearch(self, queue, parameter = "main"):
26      #Code de la fonction ici
27
28    def sendMigrants(self):
29      #Code de la fonction ici
30
31    def receiveMigrants(self, chromosomes):
32      #Code de la fonction ici
33
34    def replace(self, chromosome):
35      #Code de la fonction ici
36
37    def getFitnessData(self):
38      #Code de la fonction ici
39
40    def mate(self, chromosome1, chromosome2):
41      #Code de la fonction ici
42
43    def crossover(self, randValue1, randValue2):
44      #Code de la fonction ici
45
46    def crossPopulation(self):
47      #Code de la fonction ici
48
49    def insert(self, chromosome):
50      #Code de la fonction ici
```

```
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     #Code de la fonction ici
17
18   def start(self):
19     #Code de la fonction ici
```

```
20
21   def printResults(self):
22     #Code de la fonction ici
```

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]
```

```
                k+=1

              self._solution[indice] = i+1

              itemDemandPeriods = self.manufactItemsPeriods[i]
              itemDemandPeriods[j] = indice

            else:

              # experimental code

              # if there's no place to put this item, then i check all the other times in order to put this item there

              lbound = 0
              p = 1
              for deadline in Chromosome.problem.deadlineDemandPeriods[i]:

                zeroperiods = []
                k = lbound
                while k <= deadline:
                  if self._solution[k] == 0:
                    zeroperiods.append(k)
                  k += 1
                lbound = deadline + 1

                nbZeroPeriods = len(zeroperiods)
                if nbZeroPeriods > 0:

                  cost1 = Chromosome.getCostof(zeroperiods[0], i+1, p, copy_solution)
                  #print(" cost1 : ", cost1 )

                  k = 1
                  indice = zeroperiods[0]
                  while k < nbZeroPeriods:
                    cost2 = Chromosome.getCostof(zeroperiods[k], i+1, p, copy_solution)
                    #print(" cost2 : ", cost2 , zeroperiods[k])
                    if cost2 < cost1:
                      #print(" cost2 < cost1 : ", cost1 , cost2 )
                      indice = zeroperiods[k]
                    k+=1

                  self._solution[indice] = i+1

                  itemDemandPeriods = self.manufactItemsPeriods[i]
                  itemDemandPeriods[j] = indice

bre
ak
                p += 1


          j+=1
        i+=1
```

Table des matières

Sommaire **vi**

Remerciements **vii**

Liste des algorithmes **viii**

Liste des sigles et abréviations **ix**

Résumé **1**

Abstract **2**

Introduction **3**

1 État de l'art **5**
Introduction 5
1.1 Le dimensionnement de lots en planification de production 5
1.2 Le *Pigment sequencing problem* (PSP) 9
1.3 Les algorithmes génétiques 13
Conclusion 17

2 Matériel et solutions **18**
Introduction 18
2.1 Outils de test 18
2.2 Modèle et formulation utilisés 19
2.3 Aspects généraux aux deux méthodes de recherche proposées 20
2.4 Méthodes de recherche proposées 28
Conclusion 34

3 Résultats et Discussion **35**
Introduction 35
3.1 Données et paramètres de test 35

3.2 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques "fine-grained" et "coarse-grained" 37
3.3 Résultats expérimentaux des algorithmes génétiques parallèles hiérarchiques "coarse-grained" et "master-slave" 37
3.4 Discussion 38
Conclusion 46

Conclusion et perspectives **47**

A Annexe 1 **52**
A.1 Classes et implémentation 52

Table des matières