Algorithmes génétiques hybrides en optimisation combinatoire

Pascal Rebreyend

▶ To cite this version:

Pascal Rebreyend. Algorithmes génétiques hybrides en optimisation combinatoire. Autre [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 1999. Français. NNT : . tel-00010950

HAL Id: tel-00010950

https://theses.hal.science/tel-00010950

Submitted on 10 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Numéro d'ordre : 108
Numéro attribué par la bibliothèque : 99ENSL0108

THESE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir

le Titre de Docteur de l'École Normale Supérieure de Lyon
spécialité: **Informatique**
au titre de la formation doctorale d'Informatique de Lyon

par

PASCAL REBREYEND

Algorithmes génétiques hybrides en optimisation combinatoire

Soutenue le 14 janvier 1999

Après avis de :	EVREPIDIS	Bampis
	JEAN-CLAUDE	Konig

Devant la commission d'examen formée de :

EVREPIDIS	Bampis	
ALEXANDRE	Caminada	
AFONSO	Ferreira	(Co-directeur de thèse)
JEAN-CLAUDE	Konig	
YVES	Robert	(Président du jury)
STÉPHANE	Ubéda	(Co-directeur de thèse)

Thèse préparée au sein du
Laboratoire d'Informatique du Parallélisme - **LIP**

Table des matières

1 Introduction **3**
- 1.1 Les problèmes combinatoires . . . 3
  - 1.1.1 La complexité . . . 3
  - 1.1.2 Les classes de problèmes. . . . 4
  - 1.1.3 La théorie de la NP-complétude . . . 5
  - 1.1.4 Généralisation . . . 5
- 1.2 Organisation du document . . . 6
  - 1.2.1 Les algorithmes . . . 6
  - 1.2.2 Les algorithmes génétiques . . . 6
  - 1.2.3 L'ordonnancement de tâches . . . 7
  - 1.2.4 Le placement de composants . . . 7
  - 1.2.5 L'affectation d'émetteurs et de fréquences dans les réseaux cellulaires . . . 8

2 Algorithmes **9**
- 2.1 Une classification des algorithmes . . . 9
- 2.2 Algorithmes exacts . . . 9
  - 2.2.1 Séparation-évaluation . . . 10
  - 2.2.2 Evaluation-élagage . . . 12
- 2.3 Les heuristiques . . . 12
  - 2.3.1 Les heuristiques gloutonnes . . . 12
  - 2.3.2 Les algorithmes de listes . . . 12
  - 2.3.3 Les heuristiques aléatoires . . . 13
  - 2.3.4 Les heuristiques ou méthodes spécifiques . . . 13
- 2.4 Les méta-heuristiques . . . 13
  - 2.4.1 La descente du gradient . . . 14
  - 2.4.2 Le recuit simulé . . . 14
  - 2.4.3 Le tabou . . . 16
  - 2.4.4 GRASP . . . 17
  - 2.4.5 Le génétique . . . 18
  - 2.4.6 Les algorithmes de fourmis . . . 18
- 2.5 Conclusion . . . 20

3 Les algorithmes génétiques **21**
- 3.1 Principes . . . 21
- 3.2 Algorithme type . . . 22
- 3.3 Variantes . . . 22
  - 3.3.1 Représentation . . . 22

3.3.2 Sélection . . . 23
3.3.3 Croisement . . . 24
3.3.4 Mutation . . . 25
3.3.5 Le critère d'arrêt . . . 25
3.4 Champs d'utilisation . . . 26
3.5 Utilisation conjointe d'autres algorithmes . . . 27
3.5.1 La représentation indirecte . . . 28
3.5.2 La représentation directe . . . 28
3.6 Le parallélisme . . . 29
3.6.1 Le style maître-esclave . . . 29
3.6.2 Les îles . . . 30
3.7 Conclusion . . . 31

4 Ordonnancement 33
4.1 Introduction . . . 33
4.2 Modélisation . . . 34
4.3 Etat de l'art . . . 35
4.3.1 Les heuristiques . . . 35
4.3.2 Algorithmes de chemins critiques . . . 36
4.3.3 Les algorithmes de séparation-évaluation . . . 36
4.3.4 Un algorithme génétique . . . 37
4.4 Notre approche . . . 38
4.4.1 Le modèle . . . 38
4.4.2 La méthode . . . 39
4.5 Notre jeu de tests . . . 39
4.6 L'algorithme FSG . . . 41
4.6.1 Les heuristiques MISF . . . 41
4.6.2 La représentation . . . 42
4.6.3 La santé . . . 42
4.6.4 Le croisement . . . 43
4.6.5 La mutation . . . 47
4.6.6 Résultats . . . 47
4.7 L'algorithme génétique à représentation indirecte . . . 50
4.7.1 Travaux déjà effectués . . . 50
4.7.2 L'algorithme d'Ahmad . . . 50
4.7.3 L'algorithme de Frode . . . 51
4.8 L'algorithme IS . . . 53
4.8.1 Le croisement à un point . . . 53
4.8.2 Le croisement à deux points . . . 53
4.8.3 Le croisement de Ferland . . . 53
4.8.4 La mutation . . . 53
4.8.5 Le sélection . . . 54
4.8.6 L'heuristique HL . . . 54
4.9 Résultats . . . 55
4.10 Conclusion . . . 60

5 Placement **63**
5.1 Introduction . . . 63
5.2 Les problèmes de placement . . . 63
5.2.1 Le placement de portes logiques . . . 63
5.2.2 Le placement de cellules standards . . . 64
5.2.3 Le placement de cellules diverses . . . 64
5.3 Les travaux déjà effectués . . . 65
5.4 Notre modélisation . . . 65
5.4.1 La plaquette . . . 65
5.4.2 Les composants . . . 66
5.4.3 Les connexions . . . 67
5.4.4 La longueur des connexions . . . 67
5.4.5 Une heuristique . . . 68
5.5 Un algorithme génétique à représentation indirecte . . . 71
5.6 Un algorithme génétique à représentation directe . . . 72
5.6.1 La représentation . . . 72
5.6.2 Initialisation . . . 72
5.6.3 Croisement . . . 72
5.6.4 Mutation . . . 74
5.7 Résultats . . . 75
5.8 Conclusion . . . 77

6 Réseaux cellulaires **79**
6.1 Description du problème . . . 79
6.2 Modélisation . . . 80
6.2.1 Le terrain . . . 80
6.2.2 La couverture . . . 80
6.2.3 La demande . . . 80
6.2.4 Les cellules . . . 81
6.2.5 La zone d'interférences . . . 82
6.2.6 Les intercellules . . . 83
6.3 Le problème . . . 83
6.4 Etat de l'art . . . 83
6.4.1 Extraction du réseau . . . 84
6.4.2 L'allocation de fréquences . . . 86
6.5 L'algorithme génétique en une passe . . . 87
6.5.1 La santé . . . 87
6.5.2 L'heuristique . . . 88
6.5.3 Le croisement . . . 88
6.5.4 La mutation . . . 89
6.5.5 La sélection . . . 89
6.6 Résultats . . . 89
6.7 Conclusion . . . 90

7 Conclusion **93**
7.1 Résumé . . . 93
7.1.1 Les problèmes d'optimisation combinatoire . . . 93
7.1.2 Les algorithmes génétiques . . . 93

7.1.3 L'ordonnancement 94
7.1.4 Le placement 94
7.1.5 Les réseaux cellulaires 94
7.2 Travaux futurs 94
7.2.1 Le parallélisme 95
7.2.2 Les algorithmes d'équipes 95

8 Annexes **97**
8.1 Structure des graphes utilisés 97
8.2 Résultats pour le problème de l'ordonnancement 102
8.3 Résultats pour le problème du placement 112

Liste des figures

2.1 Séparation-évaluation 11
2.2 Algorithme de la descente du gradient 14
2.3 Algorithme du recuit simulé 15
2.4 Algorithme tabou 16
2.5 Algorithme GRASP 17
2.6 Les colonies de fourmis. 18

3.1 Algorithme génétique type 22
3.2 La sélection par roulette biaisée. 23
3.3 Le principe du croisement. 24
3.4 Exemple de mutation. 25
3.5 Algorithme parallèle de style maître-esclave. 29
3.6 Algorithme parallèle de style îles. 30

4.1 Exemple d'un ordonnancement valable. 34
4.2 Graphe dont la solution optimale n'est pas atteignable. 38
4.3 Algorithme d'une heuristique. 41
4.4 Le croisement de l'algorithme FSG 43
4.5 Exemple de croisement 45
4.6 Exemple de graphe augmenté 46
4.7 La mutation de l'algorithme FSG 48
4.8 Importance du choix du processeur. 52
4.9 Exemple du croisement de Ferland 54

5.1 Le placement de portes logiques 64
5.2 Le placement de cellules standards 65
5.3 Le placement de cellules diverses 66
5.4 Une plaquette 67
5.5 Un composant 68
5.6 La longueur des connexions 69
5.7 Espace libre 70
5.8 Le croisement 73
5.9 La mutation 74

6.1 Exemple de cellules 81
6.2 Zone d'interférences et graphe d'interférences 82
6.3 Résultats: réseaux cellulaires 90

8.1 Structure de divconq et itératif 97

8.2 Structure de diamond1 et diamond2 97
8.3 Structure de bellford et gauss 98
8.4 Structure de fft et prolog 98
8.5 Structure de diamond3 99
8.6 Structure de diamond4 99
8.7 Structure de qcd 100
8.8 Graphe ssc5 101
8.9 Résultats: bellford2-b 102
8.10 Résultats: diamond1-b 103
8.11 Résultats: diamond2-b 103
8.12 Résultats: diamond3-b 104
8.13 Résultats: diamond4-b 104
8.14 Résultats: divconq 105
8.15 Résultats: itératif2-b 105
8.16 Résultats: fft2-b 106
8.17 Résultats: gauss-b 106
8.18 Résultats: ms-gauss 107
8.19 Résultats: prolog 107
8.20 Résultats: qcd2-b 108
8.21 Résultats: elbow et cstanford 108
8.22 Résultats: ssc5 et 6 109
8.23 Résultats: ssc7 et 8 109
8.24 Résultat: ssc9 110
8.25 Temps des heuristiques suivant le nombre de tâches 111
8.26 Résultats: 1 112
8.27 Résultats: 2 113
8.28 Résultats: 3 et 4 113
8.29 Résultats: 5 114
8.30 Résultats: 6 114
8.31 Résultats: 7 et 8 115
8.32 Résultats: 9 et 10 115
8.33 Résultat: 11 116

Liste des tableaux

2.1 Tableau des distances 10

4.1 Résultats en 2 heures, algorithmes à représentation directe. . . 49
4.2 Résultats en 2 heures, représentation indirecte. 55
4.3 Résultats en 2 heures, algorithmes IS à représentation indirecte. 56
4.4 Heuristiques : résultats en une itération 57
4.5 Accélérations obtenues 58
4.6 Temps moyen d'exécution des heuristiques en millisecondes . . 59

5.1 Temps moyen de l'heuristique 75
5.2 Le jeu de tests 76
5.3 Résultats 77

6.1 Résultats: réseaux cellulaires 89

Remerciements

Cette thèse représente trois années de travail et ce travail a pu se dérouler dans de bonnes conditions et une agréable ambiance au sein du LIP.

Je tiens donc à remercier les différentes secrétaires qui m'ont permis de sortir vivant de cette jungle administrative avec efficacité. Ainsi, suivant l'ordre alphabétique des prénoms, Bénédicte, Christelle[1], Claire, Jocelyne, Marie, Nadège, Nicole, Sylvie, Valérie, et j'en oublie (il y a même des chercheurs dans notre labo...) ont toujours su comment résoudre les problèmes et tracas administratifs.

Je tiens aussi à remercier le compétent personnel du CRI qui fait le maximum pour que 24h sur 24h, 365 jours par an, l'on puisse travailler et surfer sur du code C ou latex sans problème, en dehors des pannes, des coupures de courant,...

Et puis, ayons une pensée pour ceux qui ont été obligés de rester dans le bureau que j'occupais, en essayant de se concentrer entre deux de mes phrases : Johanne, qui perdait de temps en temps son calme, Pascal (Berthomé) (pratique pour le téléphone...), Ahmed, Christian, spécialiste du débogage *C* et Olivier R., spécialiste de *XEmacs*. Et en plus, ils ont dû supporter la vision de mes posters préférés.

Je remercie aussi Olivier B., Lionel T. et Raymond, qui m'ont toujours accueilli dans leur bureau pour bavarder de la pluie et du beau temps.

Plus sérieusement, je remercie aussi mes directeurs de thèse, Afonso Ferreira et Stéphane Ubéda pour tous les conseils et avis pertinents qu'ils ont pu me donner. Je tiens à remercier aussi ceux avec qui j'ai pu collaborer : Frode Sandnes, Vassil Alexandrov, Ricardo Corrêa (qui m'a beaucoup aidé lors du début de ma thèse), et Blaise Chamaret, spécialiste du *C++* qui ne s'est jamais impatienté quand je n'étais pas en état de marche. Je remercie aussi les membres du jury et notamment mes rapporteurs pour les conseils qu'ils m'ont donnés et leur lecture attentive de mes travaux.

Enfin, je remercie toutes ces personnes qui ont écrit des documentations sur tel ou tel langage ou logiciel et qui les mettent à disposition de la communauté scientifique.

Pour finir, je remercie ma famille et mes amis qui reçoivent malgré eux mes nombreux courriers.

Et puis, je n'oublie pas les autres mais j'aime trop la forêt pour en mettre des pages.

[1] Gestionnaire TTN pour être exact.

CHAPITRE 1 Introduction

1.1 Les problèmes combinatoires

L'outil informatique est utilisé dans un grand nombre de domaines (*traitements de textes, communications, calculs scientifiques, bases de données* ...) et ce nombre ne cesse de s'accroître. La puissance de calcul actuelle permet, si une modélisation est possible, de résoudre des problèmes de taille de plus en plus grande.

On s'intéressera aux problèmes d'optimisation combinatoire.

Parmi les problèmes posés par la recherche opérationnelle, les problèmes d'optimisation combinatoire sont fréquemment rencontrés [QPPW98]. Ces problèmes consistent à trouver la meilleure solution suivant un critère donné parmi un ensemble discret (mais de taille très importante) de solutions possibles. L'optimisation combinatoire comprend des problèmes très divers : le voyageur de commerce, les problèmes d'ordonnancement, des problèmes de la théorie des graphes (coloriage, ...), ... Certains de ces domaines présentent de forts enjeux économiques ou pratiques comme les problèmes d'ordonnancement d'ateliers ou ceux de gestion d'une flotte de véhicules de livraison par exemple. Cette importance m'a conduit à considérer les problèmes d'optimisation combinatoire comme thème de recherche.

Comme il est souvent très difficile de trouver la meilleure solution possible (celle **optimale**), on se contente souvent de chercher une solution **sous-optimale**. Une solution est dite sous-optimale quand son optimalité n'a pas pu être prouvée. Il s'agit donc d'une solution approchée. Une méthode courante pour obtenir une solution approchée est d'utiliser une *heuristique*. Dans notre cas, on considérera qu'une heuristique est un algorithme qui ne garantit pas que la solution donnée soit optimale.

La qualité d'un algorithme dépend de son temps de calcul ainsi que, dans le cas d'une solution approchée, de la qualité de la solution fournie. Pour modéliser le temps de calcul, on utilise la notion de *complexité*.

1.1.1 La complexité

La complexité d'un algorithme est le nombre d'opérations élémentaires que celui-ci exécute.

On suppose que, pour chaque problème que l'on veut résoudre, l'on dispose d'une *mesure* de la taille du problème. Par exemple, on utilisera comme mesure le nombre de sommets ou d'arêtes lorsqu'une instance du problème est représentée sous forme d'un graphe. Le nombre d'opérations élémentaires effectuées par un algorithme est donc une fonction qui dépend de n, la taille des données. Cette fonction s'appelle la **complexité**. Ainsi, la somme élément

par élément de deux matrices carrées $n * n$ (on dira que leur taille est n) effectue de l'ordre de n^2 opérations et sa complexité est donc de l'ordre de n^2.

Cependant, il n'est pas nécessaire de compter avec précision ce nombre d'opérations. En effet, ce qui nous intéresse dans cette modélisation, c'est à la fois de pouvoir comparer la vitesse de deux algorithmes ainsi que de prédire le comportement de l'algorithme sur des instances de taille différente. Cela n'est pas pertinent de vouloir comparer deux algorithmes a et b de complexité proche comme, par exemple, $2*n^2+4*n+100$ et $3*n^2+2*n+50$. Le lien entre le nombre d'opérations effectuées et le temps de calcul n'est pas assez précis dans ce cas. D'autre part, il est parfois difficile de compter avec précision le nombre d'opérations, notamment lorsque celui-ci dépend de l'instance du problème (recherche d'un élément dans une liste ordonnée par exemple). Dans ce dernier cas, on parlera de complexité en moyenne par opposition à la complexité au pire cas. On cherche donc seulement à estimer le comportement de l'algorithme sur des instances de grandes tailles. Pour cela, on étudie le comportement **asymptotique** de la complexité.

Seul le comportement asymptotique de la complexité est étudié.

Soient f et g deux fonctions définies des entiers naturels vers les entiers naturels. On pose que $f \preceq g$ si et seulement s'il existe un entier n_o tel que, $\forall n \geq n_o, f(n) \leq g(n)$. On dira que la fonction f est *bornée asymptotiquement* par g.

$$O(g(n)) = \{f(n) \mid \exists k, n_0 \in N^* : f(n) \preceq k.g(n) \quad \text{pour tout} \quad n \geq n_0\}$$

Ainsi défini, l'ensemble $O(g(n))$ est l'ensemble des fonctions qui admettent comme borne asymptotique $g(n)$. La complexité d'un algorithme sera dorénavant représentée par une borne asymptotique de son nombre d'opérations élémentaires. Par exemple, on dira que les deux algorithmes a et b sont de complexité[1] $O(n^2)$. Par souci de clarté, dire qu'un algorithme est de complexité $O(n^i)$ implique qu'il n'est pas de complexité $O(n^j)$, $i < j$. En effet, si une fonction f vérifie $f \preceq O(n^i)$, alors $f \preceq O(n^j)$ pour tout $j \geq i$.

La complexité permet donc d'estimer le comportement d'un algorithme face aux instances de grandes tailles [Wil89].

1.1.2 Les classes de problèmes.

Définition 1.1.1
La complexité d'un problème est la complexité de son meilleur algorithme connu.

On peut ainsi classer les problèmes suivant leur complexité. Cependant, on distingue deux familles de problèmes : les problèmes qui ont une complexité polynomiale, et les autres, ceux dont la complexité est exponentielle [GJ79]. Les algorithmes polynomiaux sont considérés comme "efficaces" en ce sens que leur coût de calcul augmente de manière raisonnable avec la taille du problème [Cor97]. Par contre, les algorithmes non polynomiaux seront considérés comme "inefficaces" car leur temps d'exécution augmente de façon spectaculaire avec la taille de l'instance traitée. Par exemple, si pour un problème de taille 10,

On distingue les algorithmes de complexité polynomiale de ceux de complexité exponentielle.

[1] Ainsi décrit, un algorithme de complexité $O(g(n))$ peut effectuer plus de $g(n)$ opérations, mais il effectuera au plus $k * g(n)$ opérations, k étant une constante.

n^2 représente $0,0001$ seconde et pour 2^n $0,001$ seconde, ces temps passent respectivement à $0,0025$ seconde et 36 ans pour un problème de taille 50 [GJ79]. Ceci explique la partition des problèmes en deux ensembles : ceux à complexité polynomiale et ceux à complexité exponentielle.

1.1.3 La théorie de la NP-complétude

On a expliqué précédemment quelle était la complexité d'un algorithme et on a posé que la complexité d'un problème était la complexité de son meilleur[2] algorithme **connu**. Ainsi défini, dire qu'un problème a telle complexité ne prouve pas qu'il n'existe pas d'algorithme de complexité inférieure mais seulement que si celui existe, il n'a pas été découvert. Il existe nombre de problèmes pour lesquels il a été démontré qu'il n'existait pas de meilleurs algorithmes en terme de complexité que ceux existants. Ainsi, pour le problème de la somme élément par élément de deux matrices, la complexité de l'algorithme ($O(n^2)$) correspond aussi au nombre minimum d'opérations nécessaires. Par contre, il existe des problèmes dont on ne sait pas s'il existe de meilleurs algorithmes. La théorie de la NP-complétude nous permet de classer ces problèmes.

La théorie de la NP-complétude est basée sur les problèmes de **décision**. Un tel problème admet seulement deux solutions : oui et non.[3]

Les problèmes NP-complets sont "difficiles" à résoudre.

Un problème de décision sera dit polynomial ou appartenant à la classe P s'il existe un algorithme polynomial qui le résout. Un problème appartient à la classe NP s'il existe un algorithme polynomial qui permet de vérifier qu'une réponse est correcte. Par exemple, dans le cas problème du voyageur de commerce, il n'existe pas, pour l'instant, d'algorithme polynomial qui permette de répondre à la question : "Existe-t-il un parcours de longueur inférieure à L ? ". Par contre, on peut vérifier par un algorithme polynomial si une réponse *oui* est juste : il suffit de calculer sa longueur. Ainsi définis, $P \subseteq NP$.

La théorie de la NP-complétude montre qu'il existe, parmi les problèmes de décision NP, des problèmes qui sont équivalents face à la question

"$P = NP$?".

Ces problèmes sont dits NP-complets. Il a été démontré que, si pour un problème de cette classe, on découvrait un algorithme polynomial, tous les problèmes de NP appartiendraient alors à P. Réciproquement, si on montre qu'un de ces problèmes ne peut être résolu de manière polynomiale, alors aucun problème de NP-complet n'appartient à P.

1.1.4 Généralisation

Ces résultats sont étendus aux algorithmes d'optimisation.

Les considérations du paragraphe précédent sur la théorie de la NP-complétude sont basées sur des problèmes de décision. Or, beaucoup de problèmes sont des problèmes d'optimisation qui cherchent la meilleure solution parmi un ensemble de solutions possibles. Pour un problème quelconque, on dira qu'il est *NP-difficile* si on peut montrer qu'il ne peut pas se résoudre polynomialement si $P \neq NP$.

[2] En terme de complexité.

[3] Cependant, on peut toujours reformuler un problème d'optimisation en : "existe-t-il une solution meilleure que ... ".

1.2 Organisation du document

Le coeur du document est découpé en cinq parties principales. Le premier chapitre présente les différentes méthodes et algorithmes qui permettent d'obtenir des solutions sous-optimales pour les problèmes d'optimisation. Le chapitre suivant détaille les algorithmes génétiques sur lesquels repose le travail de recherche présenté ici. Les trois derniers chapitres présentent l'utilisation de divers algorithmes génétiques sur trois problèmes distincts : l'ordonnancement de tâches, le placement de composants électroniques et la conception d'un réseau cellulaire. Ces problèmes ont été choisis suivant plusieurs critères. Le problème de l'ordonnancement étant un problème couramment étudié, il permet d'obtenir des jeux de tests réalistes et de comparer facilement avec d'autres méthodes les résultats obtenus. Cependant, afin de pouvoir mieux tirer des conclusions sur les méthodes proposées ici, il est nécessaire de les appliquer sur plus d'un problème. Ainsi, il fut choisi d'étudier aussi un problème de placement et un autre concernant la conception de réseaux cellulaires. Comme le problème de l'ordonnancement a été largement étudié, on dispose pour les cas les plus courants de modèles réalistes et couramment utilisés. Par contre, les deux autres problèmes étant moins courants ou plus récents et très liés à des cas concrets, il n'existe pas de modélisation universelle. Il faut donc, avant de les résoudre, effectuer une modélisation.

1.2.1 Les algorithmes

Cette partie présente de manière succincte les grandes familles d'algorithmes pour résoudre les problèmes combinatoires. Dans cette partie, les algorithmes sont regroupés en trois catégories suivant leur principe de fonctionnement: les algorithmes "exacts" qui recherchent la solution optimale, les heuristiques et les méta-heuristiques. Les algorithmes les plus usités (séparation-évaluation, algorithme de liste, recuit simulé, colonies de fourmis,...) seront présentés afin que le lecteur puisse voir quels sont les avantages et les inconvénients de chaque famille d'algorithmes. Pour chaque algorithme présenté, la question de sa parallélisation sera abordée.

1.2.2 Les algorithmes génétiques

Un algorithme hybride est le mélange de deux algorithmes distincts.

Il s'agit d'une introduction et d'une présentation générale des algorithmes de type génétique ainsi que leurs avantages et inconvénients. Ce chapitre a pour but de familiariser le lecteur à ce type d'algorithme évolutionnaire. Cette partie ne présente pas de manière exhaustive toutes les variantes possibles et imaginables. En outre, diverses méthodes de parallélisation sont abordées et le concept des algorithmes *hybrides* est expliqué. Les algorithmes hybrides sont le résultat du mélange de deux algorithmes distincts. En l'occurrence, nous examinerons comment utiliser à la fois un algorithme génétique et une heuristique.

Il existe deux approches pour réaliser ceci : celle à *représentation directe* et celle à *représentation indirecte*. Elles diffèrent sur deux caractéristiques : dans le premier cas, le chromosome représente directement une solution du problème tandis que dans le second cas, le chromosome représente une solution qui n'a

pas de lien direct avec le problème originel. L'autre différence est au niveau de l'utilisation de l'heuristique. Dans le cas de la représentation indirecte, l'heuristique est introduite via la fonction de *santé* (ou fonction objectif) tandis que dans l'autre type de représentation, l'heuristique est utilisée en modifiant un ou plusieurs opérateurs utilisant les chromosomes (mutation, croisement).

Il existe deux méthodes pour coupler un algorithme génétique avec une heuristique: la représentation directe et celle indirecte.

1.2.3 L'ordonnancement de tâches

Il s'agit d'un problème bien formalisé qui se pose lorsque l'on veut exécuter de manière efficace un programme parallèle sur un ordinateur parallèle. Le programme est décomposé en *tâches* qui communiquent entre elles. Ces tâches doivent être réparties sur les processeurs afin de minimiser le temps d'exécution du programme. Ce problème est modélisé par un graphe orienté acyclique valué. Nous nous placerons dans le cas du placement **statique** où tous les paramètres sont connus avant l'exécution du programme. De même, la duplication de tâches sera interdite. Le temps de communication sera modélisé en $\tau L + \beta$, où β est le coût de l'initialisation de l'envoi du message, τ le temps de transfert d'une information de taille unité et L la taille du message.

Pour ce problème, divers heuristiques et algorithmes génétiques hybrides seront expliqués et testés. Les algorithmes génétiques permettront de mieux comprendre la différence entre la représentation directe et celle indirecte ; différence au niveau de la conception, mais aussi au niveau du ratio temps de calcul / résultats. Les tests porteront sur des graphes différents. Certains d'entre eux représentent des programmes réalistes simulés sur une machine existante. Les résultats obtenus permettent de montrer l'intérêt de l'approche génétique hybride par rapport aux heuristiques.

Le jeu de tests sera choisi pour être le plus fidèle possible à la réalité.

1.2.4 Le placement de composants

Ce problème consiste à optimiser le placement de composants sur un circuit électronique afin de réduire la longueur des connexions électriques entre les composants. La modélisation et les hypothèses faites pour simplifier le problème seront présentées. Ce problème n'a pas de modélisation universelle et reconnue, contrairement au problème précédent. Ceci est dû, en outre, au fait que suivant l'entreprise où il apparaît et le type de circuit, des hypothèses peuvent être faites ou non. Avant de résoudre ce problème, il faut le modéliser en réalisant le meilleur compromis *simplicité de la modélisation/précision de la modélisation* pour la situation qui nous intéresse. Il faut en effet que la modélisation soit la plus réaliste possible sans pour autant que sa complexité empêche de trouver une bonne solution. Ceci pose des problèmes lorsqu'on doit comparer les résultats obtenus par deux méthodes différentes basées sur des modèles différents. Dans ce cas, soit on a la chance de pouvoir comparer les solutions dans la réalité en questionnant l'homme de l'art soit on est obligé d'effectuer la comparaison dans un même modèle. Dans ce dernier cas, cela peut être facilité si l'une des modélisation est obtenue par raffinement de l'autre.

Il n'existe pas de modélisation reconnue pour le problème de placement de composants électroniques.

Le problème du placement de composants est décomposé en trois catégories suivant que les cellules sont de taille identique, de forme rectangulaire avec un côté de même dimension ou alors de forme et de dimension quelconques.

Dans notre cas, nous nous placerons dans le problème du placement de cellules diverses, c'est-à-dire de cellules de forme rectangulaire mais de dimensions quelconques. Nous négligerons la place utilisée par les connexions. Ceci entraîne donc que les chemins empruntés par ces connexions ne seront pas calculés. Nous chercherons à minimiser la longueur des connexions, longueurs mesurées en utilisant la distance dite de *Manhattan*. Une fois cette modélisation expliquée, un algorithme glouton de liste sera présenté. Il servira de base aux algorithmes génétiques proposés. Enfin, deux algorithmes génétiques hybrides utilisant les deux types de représentation seront présentés et comparés.

1.2.5 L'affectation d'émetteurs et de fréquences dans les réseaux cellulaires

Les réseaux cellulaires doivent à la fois assurer une couverture importante et offrir à chaque mobile une fréquence si besoin.

Les réseaux cellulaires sont très utilisés (téléphonie, radio, télévision,...) car ils permettent d'obtenir une grande couverture tout en utilisant des fréquences permettant des communications à haut-débit. En effet, dans un tel réseau, le territoire est décomposé en *cellules* qui correspondent à la couverture d'un émetteur fixe (BTS, pour Base Transceiver Station). Cependant, nous traiterons ici le cas des réseaux cellulaires bi-directionnels (téléphonie par exemple). Ainsi, un bon réseau cellulaire bi-directionnel doit, à la fois, avoir une bonne couverture du territoire (comme tout réseau cellulaire) mais aussi fournir à chaque mobile une fréquence si besoin. Les solutions classiques de réseaux de diffusion qui consistent à utiliser des émetteurs très puissants couvrant un maximum de territoire ne sont donc pas adaptées à ce problème car dans ce cas, chaque émetteur ne dispose pas suffisamment de fréquences pour satisfaire la demande importante de la zone couverte. En effet, il ne peut utiliser, pour cause d'interférences, les fréquences employées par les émetteurs voisins. Il est donc nécessaire de chercher un compromis entre la surface couverte par un émetteur et sa capacité à satisfaire la demande. Il faut donc trouver les meilleurs emplacements possibles pour chaque émetteur mais aussi, pour chacun d'eux, l'ensemble des fréquences qu'il peut utiliser. Dans ce problème, une part importante du travail a consisté à effectuer la modélisation nécessaire.

L'algorithme génétique proposé détermine en même temps la localisation des émetteurs et l'allocation de fréquences.

Cette modélisation est basée sur les concepts de *cellules* et d'*intercellules* qui caractérisent respectivement la zone de couverture d'un émetteur et celle d'un ensemble d'émetteurs. Cette modélisation étant faite, un algorithme génétique hybride a pu être mis au point. Sa particularité est d'effectuer à la fois le placement des émetteurs et l'allocation des fréquences, contrairement aux autres méthodes qui dissociaient ces deux phases. La fonction de santé utilisée est basée sur le concept de *gens mécontents*. Autrement dit, elle mesure les failles de la solution à évaluer.

CHAPITRE 2

Algorithmes

2.1 Une classification des algorithmes

Pour tenter de résoudre de manière approchée un problème $NP-complet$, il existe plusieurs méthodes possibles, méthodes de nature radicalement différente. On peut cependant distinguer plusieurs catégories parmi ces algorithmes ou méthodes :

Il existe trois catégories principales d'algorithmes dédiés aux problèmes de classe NP

- les algorithmes **exacts**.
- les **heuristiques** qui sont des algorithmes spécifiques à un problème donnant un résultat approché.
- les **méta-heuristiques** qui sont des méthodes générales.

Par la suite, une présentation succincte de chaque catégorie d'algorithmes est faite.

2.2 Algorithmes exacts

Ces algorithmes ont comme but la solution optimale mais fournissent en général une solution sous-optimale.

Il s'agit d'algorithmes dont le but n'est ni plus ni moins que de fournir la solution optimale. Bien sûr, ils échouent faute de temps dans ce but dès que la taille du problème devient un rien conséquente. Cependant, leur arrêt prématuré n'empêche pas d'obtenir une solution sous-optimale. La figure 2.1 représente l'arbre permettant d'obtenir toutes les solutions qui correspondent au problème du voyageur de commerce pour 4 villes, décrit par le tableau 2.1. Un algorithme exact de recherche exhaustive parcourera toutes les feuilles de l'arbre pour trouver une solution optimale (parcours dans l'ordre des villes A,B,D et C puis retour à la ville A, solution représentée par ABDCA par la suite).

Tout algorithme de recherche exhaustive fait partie de cette famille. En effet, dans un tel algorithme, on mémorise la meilleure solution trouvée jusqu'alors et donc, en cas d'arrêt anticipé, on peut la récupérer. Il s'agit peut-être de la solution optimale mais rien ne le prouve. Elle est donc sous-optimale. Cependant, il existe des algorithmes de recherche exhaustive spécialement prévus pour cette situation comme les algorithmes de séparation-évaluation.

2.2.1 Séparation-évaluation

L'algorithme de séparation-évaluation effectue un parcours "intelligent" de l'arbre des solutions.

Ce type d'algorithme, plus connu sous le vocable anglais de "**branch and bound**", est très étudié à l'heure actuelle car il présente notamment un parallélisme potentiel élevé [Cor97, BCD+96]. Sa structure est simple: il ne s'agit ni plus ni moins que d'essayer de faire une recherche exhaustive mais "intelligente" de toutes les solutions.

En fait, on parcourt l'arbre des solutions en essayant de "*visiter*" tout d'abord les régions (sous-arbres) intéressantes et en essayant d'élaguer les sous-arbres dont on sait qu'ils ne peuvent pas fournir une meilleure solution que celle courante ou que celle fournie par un autre sous-espace. En fait, lors d'une recherche exhaustive classique des solutions, on "visite" les solutions dans un ordre fixé, généralement l'ordre préfixe obtenu par le parcours de l'arbre des solutions. L'algorithme de séparation-évaluation effectue deux modifications profondes:

1. Un sous-espace n'est pas exploré si on sait par avance qu'il ne peut pas améliorer la meilleure solution courante.

2. L'ordre préfixe est remplacé par un ordre lié à la qualité estimée de la meilleure solution du sous-espace de recherche considéré.

	A	B	C	D
A	-	3	1	5
B	3	-	7	3
C	1	7	-	2
D	5	3	2	-

Tableau 2.1: Tableau des distances

Dans l'exemple de la figure 2.1 correspondant au tableau 2.1 qui décrit un problème du voyageur de commerce pour 4 villes, toutes les solutions du problème sont représentées. Si le voyageur part de la ville A, il a 3 possibilités pour choisir où aller (ville B,C ou D). L'algorithme présenté ici évalue les solutions provisoires suivant la longueur du trajet déjà construit. L'algorithme maintient une liste *ordonnée* des solutions provisoires. Ainsi, au départ, la liste contient un seul élément, le noeud A. La valeur associée à ce noeud est 0. L'algorithme enlève ce noeud de la liste, le sépare, c'est-à-dire construit les trois noeuds AB, AC et AD, les évalue et les stocke dans la liste ordonnée. AC est le premier élément puisque c'est le meilleur candidat. Le noeud AC est ensuite séparé pour former les deux noeuds ACB et ACD. A ce moment-là, la liste continent les noeuds AB, ACB, ACD et AD. Les noeuds de valeur minimale 3 AB et ACD sont alors séparés et les noeuds ABC, ABD et ACDB intègrent la liste. La liste comporte alors les noeuds (dans l'ordre) AD, ACDB, ABD, ACB et ABC. Le noeud AD est alors séparé et la liste de vient: ACDB, ADC, ADB, ABD, ACB et ABC. Le noeud ACDB est alors séparé et une solution (ACDBA) de valeur 9 trouvée. A ce moment-là, l'algorithme ne sait pas qu'il s'agit d'une solution optimale. Cependant, tous les noeuds de valeur supérieure peuvent être supprimés de la liste car on est certain qu'il ne peuvent

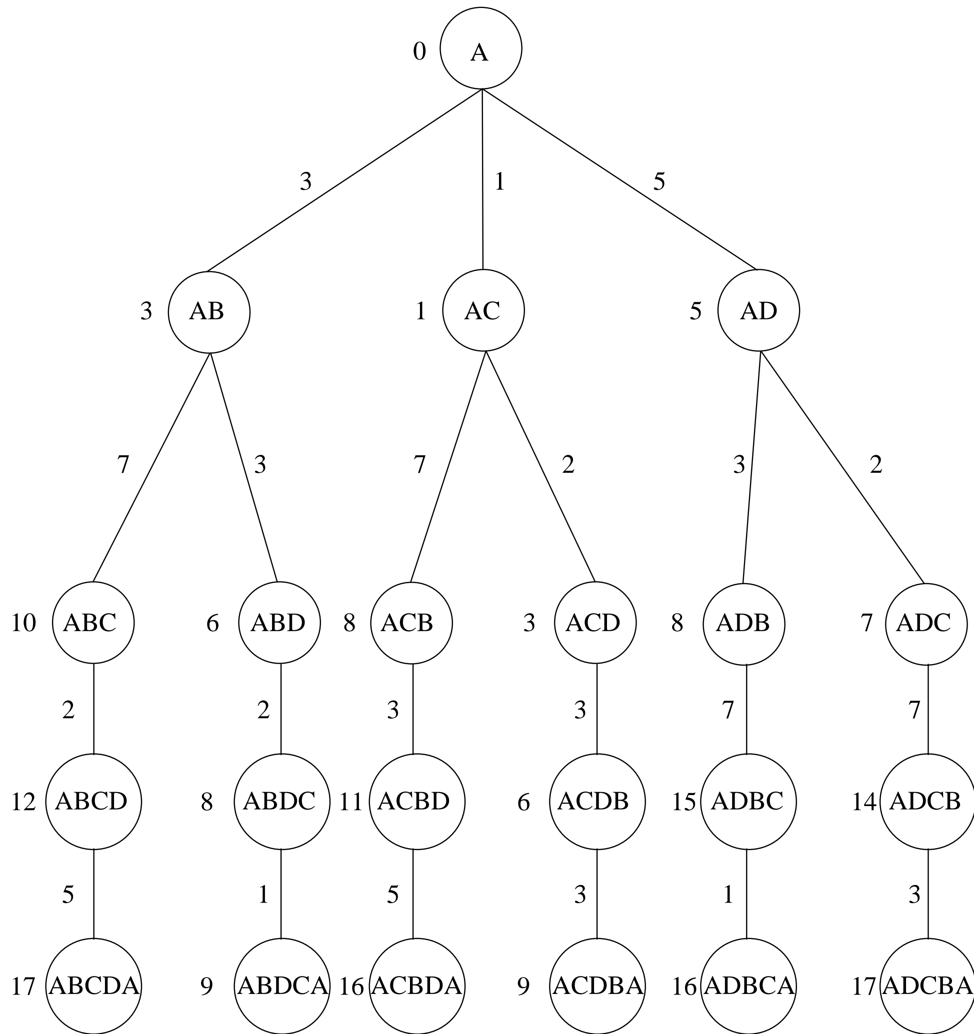


Figure 2.1: Séparation-évaluation

conduire à une solution meilleure. L'algorithme continue jusqu'à ce que la liste deviennent vide. A ce moment-là, l'optimilité de la solution est garantie. On peut avec cet algorithme résoudre des problèmes de taille relativement importante. Cependant, la durée de calcul dépend très fortement de l'instance considérée. Cette méthode permet ainsi de résoudre des problèmes de taille importante. Ainsi, on peut résoudre en temps limité (moins de 10 minutes sur un ordinateur parallèle) des problèmes contenant environ 250 villes [TLM95].

Nombre de bibliothèques parallèles d'évaluation - séparation existe car cet algorithme présente un parallélisme intrinsèque important.

Dans le cas de problèmes plus importants, des modifications génèrent rapidement des solutions complètes. Cela peut se faire en forçant l'algorithme à descendre dans l'arbre, ou en complétant la solution partielle représentée par un noeud interne de l'arbre.

Ce type d'algorithme a deux points forts. Le premier, c'est sa forte potentialité pour être parallélisé puisqu'il s'agit du parcours d'un arbre. Cette potentialité explique que nombre de bibliothèques parallèles d'évaluation-séparation ont été mises au point car si le parallélisme existe de manière intrinsèque, il

n'en est pas moins vrai que son exploitation de manière optimale n'est pas forcément triviale sur des machines parallèles quelconques [Cor97, BCD+96]. En outre, les bibliothèques proposées permettent facilement à l'utilisateur d'obtenir un algorithme parallèle, notamment en gérant le passage de messages nécessaire lors de l'utilisation d'une machine à mémoire distribuée. Le second point fort, qui peut avoir son importance, c'est qu'en plus de fournir une bonne solution, on dispose d'une borne sur la solution optimale ce qui permet soit de garantir l'optimalité de la solution trouvée, soit d'avoir une estimation de sa qualité. En effet, l'algorithme calcule pour chaque sous-espace un majorant de la pire solution contenu dans ce sous-espace.

Ce type d'algorithme n'est pas adapté à certains problèmes.

En revanche, par sa nature même, cette méthode nécessite des adaptations et perd rapidement de son efficacité lorsque la taille des problèmes s'accroît. Ainsi, on ne connaît pas, par exemple, d'implémentation performante pour le problème du partitionnement de graphe. La méthode parallèle proposée par [CT91] se limite par exemple à des graphes de taille très réduite (50 noeuds environ).

2.2.2 Evaluation-élagage

Il s'agit d'un algorithme voisin de celui de la séparation-évaluation, connu sous le nom anglais de "branch-and-cut". La différence essentielle vient du fait que si l'algorithme de séparation-évaluation fait des choix provisoires (il choisit en fait un ordre de parcours de l'arbre des solutions), l'algorithme d'évaluation-élagage essaie de faire des choix définitifs, c'est-à-dire couper l'arbre de manière définitive [PR91]. Cet algorithme ne sera pas détaillé ici car il est voisin de par sa structure de celui de l'évaluation-séparation.

2.3 Les heuristiques

Les heuristiques calculent rapidement des solutions sous-optimales.

Les heuristiques sont des algorithmes qui cherchent avant tout à trouver une solution sous-optimale **très rapidement**. Ainsi, leur complexité est souvent en $O(n^i)$, avec i assez faible. Diverses heuristiques générales sont présentées ici.

2.3.1 Les heuristiques gloutonnes

Les heuristiques gloutonnes optimisent à chaque pas le coût de la solution provisoire.

Il s'agit d'un style d'heuristiques très souvent utilisées. Ces heuristiques sont employées quand on peut construire la solution pas-à-pas, comme dans le cas du voyageur de commerce. Ces heuristiques choisissent à chaque fois la solution qui optimise sur le moment le critère de qualité. Ainsi, dans le cas du voyageur de commerce, on choisit par exemple de sélectionner la ville la plus proche comme prochaine destination.

2.3.2 Les algorithmes de listes

Les algorithmes de listes utilisent un ordre prédéterminé sur les élements de la solution.

Les algorithmes de listes sont des heuristiques particulières. Cette méthode construit comme une heuristique gloutonne la solution pas-à-pas. Nous considérerons ici que la particularité d'un algorithme de liste est d'utiliser un ordre sur les éléments de la solution **prédéterminé** et **non arbitraire**. La

différence réside dans le fait que l'algorithme de liste choisit l'élément à placer dans la solution en se basant sur cet ordre plutôt que sur d'autres critères comme les heuristiques gloutonnes. Ce type d'algorithme est souvent utilisé pour les problèmes d'ordonnancement. Dans ce cas, on trie les tâches suivant un critère (date limite d'exécution, priorité,...) et on construit la solution en les ordonnançant dans cet ordre. Cet ordre n'est pas modifié au cours de la création de la solution.

2.3.3 Les heuristiques aléatoires

On a vu qu'une heuristique gloutonne construisait la solution pas-à-pas en utilisant un critère de recherche comme le choix de la ville la plus proche. Cependant, quelle ville choisir si deux villes sont à la même distance ? Plusieurs politiques sont possibles :

1. utiliser un autre critère pour départager les candidats.
2. utiliser un ordre total arbitraire (par exemple l'ordre lexicographique).
3. choisir au hasard.

On peut facilement utiliser de l'aléatoire dans une heuristique.

Les heuristiques aléatoires sont celles qui utilisent la politique décrite au point 3. Leur particularité est de ne pas être **déterministe** : deux exécutions de la même heuristique peuvent donner deux résultats différents si le générateur de nombres aléatoires ne fournit pas la même séquence.

Les algorithmes de listes, qui sont des heuristiques, peuvent quant à eux être modifiés pour devenir des heuristiques aléatoires : Il suffit que l'ordre sur les éléments soit généré aléatoirement ou que l'insertion de cet élément dans la solution utilise un tirage aléatoire.

2.3.4 Les heuristiques ou méthodes spécifiques

Il existe des algorithmes dédiés à un problème.

Les méthodes précédemment expliquées sont utilisées et donnent des résultats intéressants dans nombre de problèmes très divers. De plus, elles sont applicables de par leur principe à tous les problèmes ou presque. Ainsi, il est facile d'imaginer une heuristique gloutonne pour la plupart des problèmes d'optimisation (voyageur de commerce, ordonnancement, ...). Mais il existe des méthodes dont le principe même ou dont les résultats limite leur applications à des problèmes bien spécifiques. C'est par exemple le cas du partitionnement de graphe. Ce problème consiste à partitionner un graphe en deux (ou plus) sous-ensembles de façon à minimiser un critère. Pour ce problème, il existe une heuristique nommée partitionnement multi-niveaux et elle donne des résultats très intéressants [Pel96, KK96]. Cependant, cette méthode semble difficilement applicable de manière efficace à d'autres problème comme celui du voyageur de commerce.

2.4 Les méta-heuristiques

Les heuristiques présentées ci-dessus sont des méthodes basées sur le problème lui-même. Il existe des méthodes, qu'on appelle méta-heuristiques, qui

reposent sur un concept non lié au problème. On remarquera l'utilisation fréquente de l'aléatoire parmi ces méthodes. Les méta-heuristiques englobent les algorithmes *évolutionnaires* que sont les algorithmes génétiques et les colonies de fourmis.

2.4.1 La descente du gradient

(C(s): coût d'une solution s)
1. Choisir une solution x
2. Choisir une solution s, voisine de x
   telle que $C(s) < C(x)$
   Si aucun voisin ne verifie cette inégalité
   alors x est un minimum local, arrêt.
3. $x \leftarrow s$
4. retour en 2.

Figure 2.2: Algorithme de la descente du gradient

L'algorithme de la descente du gradient calcule un minimum **local**.

Il s'agit d'une heuristique qui donne comme résultat une solution qui est un minimum (ou un maximum) **local**. Le résultat dépend du voisinage retenu, de la solution initiale x et de la façon de choisir le voisin d'une solution (cf figure 2.2). En effet, suivant la solution initiale ou l'ordre de visite des voisins, on peut obtenir des solutions différentes. Cette particularité permet de le considérer à la limite comme un algorithme aléatoire[1]. Cette heuristique, facile à mettre en oeuvre, sert de base aux algorithmes du recuit simulé et tabou expliqués par la suite. Ces deux algorithmes s'inspirent de cette heuristique en essayant de ne pas s'arrêter dans un optimum local qui n'est peut-être pas la solution optimale globale.

2.4.2 Le recuit simulé

L'algorithme du recuit simulé est une version "lâche" de celui de la descente du gradient.

Le principe de cette méthode est calqué sur une manipulation sidérurgique [KGV83, CRP96, Reb95]. Cette manipulation a pour but, lors de la solidification de certains métaux, d'éviter que les molécules se retrouvent dans une configuration qui pourrait entraîner des faiblesses dans la structure du métal. Pour éviter ceci, le refroidissement est fait de plus en plus lentement au cours de la phase de solidification afin de laisser le temps aux molécules de trouver une position stable, c'est-à-dire celle qui minimise l'énergie. En effet, plus la température est élevée, plus les molécules en phase gazeuse ou liquide se déplacent vite (agitation moléculaire).

Il s'agit donc d'un algorithme itératif qui fait évoluer une solution courante (cf figure 2.3).

[1] Si, par exemple, la solution initiale est choisie de manière aléatoire.

```
T ← température initiale
x ← état initial
Tant que (non arrêt)
    y ← Etat-voisin(x)
    Δ C ← C(y) − C(y)
    P ← min(1, e^(−ΔC/T))
    Si alea(0,1) ≤ P alors x ← y
    T ← α(T)
FIN
```

Figure 2.3: Algorithme du recuit simulé

Cet algorithme est voisin de celui de la descente du gradient mais au lieu de n'accepter que des *transitions* qui améliorent la solution, il autorise des transitions vers des solutions plus mauvaises suivant une loi de probabilité fonction de la température du système. La fonction de température α vérifie $\alpha(T) \leq T$, sauf cas très particuliers non présentés dans ce document (pour sortir d'un minimum local par exemple). Ainsi, à chaque étape, la température décroit (au sens large) et l'algorithme accepte de moins en moins facilement des transitions vers des solutions plus mauvaises.

Le recuit simulé a comme principal inconvénient d'avoir des performances qui dépendent beaucoup du voisinage défini et du réglage de la fonction de température, notamment pour éviter que l'algorithme ne reste prisonnier d'un minimum local.

Par contre, c'est un algorithme simple qui est facile à mettre en oeuvre. Cependant, il n'est pas évident de choisir une fonction de décroissance pour la température [Aze92, KA97].

De même, il est difficilement parallélisable vu son déroulement intrinsèquement séquentiel. Ainsi, généralement, les implémentations parallèles ne sont ni plus ni moins que le lancement sur chaque noeud du même algorithme avec des solutions initiales différentes.

L'algorithme du recuit simulé est difficile à paralléliser de manière efficace.

Les différentes techniques de parallélisation sont:

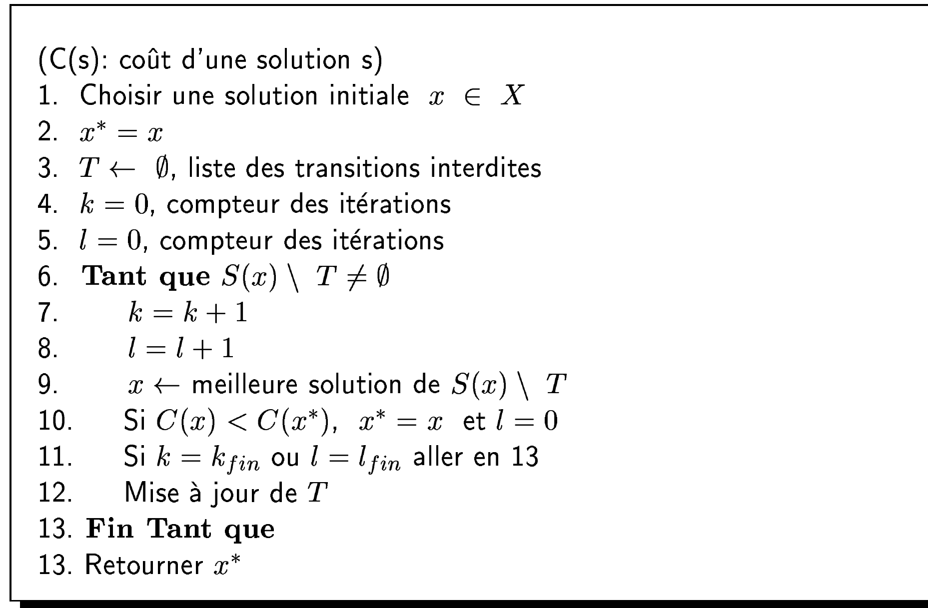
1. avoir sur chaque processeur un algorithme indépendant (une seule communication nécessaire pour rapatrier à la fin des calculs le meilleur résultat sur un processeur fixé).

2. sur chaque processeur avoir un algorithme indépendant qui, régulièrement, choisit pour continuer entre sa solution courante et celle de son voisin.

3. La solution est décomposée sur les processeurs et chacun d'entre eux optimise sa partie de solution en utilisant un recuit simulé. Cette parallélisation est réservée à certains problèmes particuliers où cela est possible, comme par exemple certains problèmes de placement [Aze92]

Cependant, dans l'algoritme du recuit simulé, il peut facilement se produire une succession de transitions qui soient un cycle, c'est-à-dire qui ramènent à un état précédent. C'est ce qui a conduit à developper les algorithmes tabous expliqués ci-après.

2.4.3 Le tabou



(C(s): coût d'une solution s)
1. Choisir une solution initiale $x \in X$
2. $x^* = x$
3. $T \leftarrow \emptyset$, liste des transitions interdites
4. $k = 0$, compteur des itérations
5. $l = 0$, compteur des itérations
6. **Tant que** $S(x) \setminus T \neq \emptyset$
7. $k = k + 1$
8. $l = l + 1$
9. $x \leftarrow$ meilleure solution de $S(x) \setminus T$
10. Si $C(x) < C(x^*)$, $x^* = x$ et $l = 0$
11. Si $k = k_{fin}$ ou $l = l_{fin}$ aller en 13
12. Mise à jour de T
13. **Fin Tant que**
13. Retourner x^*

Figure 2.4: Algorithme tabou

L'algorithme présenté à la figure 2.4 est une version simplifiée de l'algorithme tabou présenté par [Glo89, Glo90]. X représente l'ensemble des solutions. x^* est la meilleure solution courante. $S(x)$ représente l'ensemble des solutions voisines de la solution x, k_{fin} le nombre maximum d'itérations autorisées et l_{fin} le nombre d'itérations autorisées sans amélioration. La liste T des transitions interdites est continuellement mise à jour à chaque itération. Nombre de politiques de mise à jour de T existent. Toutes ont pour but d'éviter que l'algorithme revienne à une solution courante déjà visitée. Cependant, pour des problèmes de stockage et de temps de vérification, la liste T ne contient qu'une partie des transitions qui ramèneraient à une solution déjà obtenue.

L'algorithme tabou est un algorithme du recuit simulé où l'on interdit à l'algorithme d'effectuer des cycles.

Tel que décrit ici, il ne s'agit pas d'un algorithme aléatoire mais on utilise généralement des versions utilisant une recherche aléatoire[2]. Contrairement à l'algorithme de la descente du gradient, on ne s'arrête pas si l'on a obtenu un minimum local. Dans certains cas où l'ensemble $S(x) \setminus T$ est grand, au lieu de prendre le meilleur voisin autorisé (étape 9), on peut choisir le premier voisin autorisé de coût inférieur à la solution courante. En général, la liste T des transitions interdites a pour but d'éviter de suivre des cycles pour les différentes solutions courantes. Ainsi, on interdit en général les transitions qui

[2] De la même manière que la descente du gradient.

ramèneraient à un état que l'algorithme aurait eu au cours des n itérations précédentes.

L'efficacité d'un tel algorithme dépend surtout de la définition des transitions interdites, de la solution initiale et du voisinage défini. Concernant son éventuelle parallélisation, les remarques faites pour le recuit simulé restent valables [NF97, PPMR95, Glo89, Glo90].

2.4.4 GRASP

algorithme principal
(C(x): coût d'une solution x)
(N(x): voisinage d'une solution x)
$x^* = \infty$;
Tant que critère d'arrêt n'est pas atteint
  $x \leftarrow$ construire-solution()
  $\tilde{x} \leftarrow$ minimum-local(x)
  si $C(\tilde{x}) < C(x)$ alors $x^* = \tilde{x}$
Fin Tant que
Retourner la solution x^*

construire-solution()
$S \leftarrow \emptyset$
tant que la construction n'est pas finie
  avec une fonction *gloutonne*,
    construire la liste L des candidats
  choisir un élément s *au hasard* dans L
   $S = S \cup \{s\}$
  mettre à jour la fonction gloutonne
Fin Tant que
retourner la solution x correspondant à S

minimum-local(x)
tant que x n'est pas un optimal local
  trouver $\tilde{x} \in N(x), \quad C(\tilde{x}) < C(x)$
  $x = \tilde{x}$
Fin Tant que
retourner solution localement optimale $\tilde{x}$;

Figure 2.5: Algorithme GRASP

GRASP est le sigle de "Greedy Randomized Adaptive Search Procedure" et détaillé par [FR95]. Il s'agit d'un algorithme itératif qui mélange recherche gloutonne et recherche aléatoire comme le montre son algorithme détaillé à la figure 2.5. Cet algorithme est basé sur deux procédures. L'une, appelée ici construire-solution, construit aléatoirement une solution. L'autre, appelée

minimum-local, est en fait la méthode de descente du gradient expliquée à la section 2.4.1. L'avantage de cet algorithme est de combiner la recherche gloutonne de la descente du gradient avec un processus aléatoire. Ainsi, la solution fournie par cette méthode est un minimum local et l'appel à chaque itération de la procédure construit-solution permet de ne pas retourner un mauvais minimum local. Par contre, cela ne change pas son aptitude à être parallélisé qui reste faible [PPMR95].

2.4.5 Le génétique

Il s'agit d'une famille d'algorithmes qui s'inspirent de la sélection biologique décrite par Darwin [Gol89, Hol92]. Les principaux avantages sont un parallélisme intrinsèque important et une certaine robustesse par rapport aux différentes instances du problème. Par contre, son efficacité par rapport à son besoin en calculs est parfois moyenne. Il sera expliqué plus en détail au chapitre suivant car il s'agit du sujet de cette thèse. En effet, nous chercherons à résoudre les problèmes d'optimisation combinatoire en utilisant cette méthode.

2.4.6 Les algorithmes de fourmis

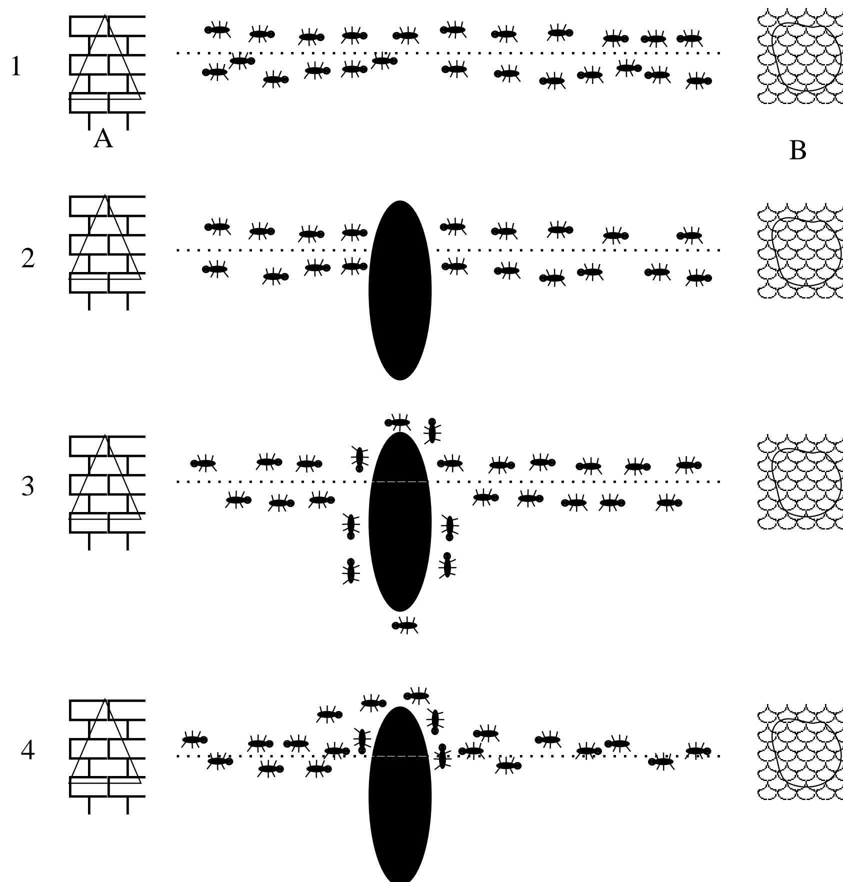


Figure 2.6: Les colonies de fourmis.

Les fourmis vivent dans les fourmilières et ont bâti une vie en société très élaborée. Notamment, ces animaux sont obligés d'aller chercher à l'extérieur de la fourmilière la nourriture et, comme toute société bien gérée, les fourmis collaborent entre elles pour trouver le plus près possible cette nourriture. Ainsi, les fourmis suivent toujours le même chemin pour atteindre la nourriture et ce chemin est le plus court chemin possible. Beaucoup de recherches ont été conduites pour appliquer ces observations, notamment au problème du *voyageur de commerce* [DG97, CDM92].

Les fourmis vont chercher à l'extérieur de la fourmilière la nourriture indispensable à leur survie. Les fourmis ont la particularité de déposer lors de leurs déplacements une substance appelée la **phéromone** qu'elles peuvent détecter. Ainsi, le chemin régulièrement emprunté par les fourmis est marqué et repéré par cette substance.

Les algorithmes de colonies de fourmis s'appliquent bien au problème du voyageur de commerce.

Supposons que les fourmis aient trouvé le plus court chemin entre la fourmilière et la nourriture. Ce chemin est reconnu par elles grâce à la phéromone qui est déposée sur le sol, comme montré dans le cas *1* de la figure 2.6. Si un obstacle vient à obstruer ce chemin, les fourmis vont contourner cet obstacle par la droite ou la gauche, en se répartissant de manière équitable entre les deux choix possibles (cas *2*) car aucune information ne permet aux fourmis d'effectuer un choix. Mais si un chemin est plus court que l'autre, les fourmis enpruntant ce chemin vont aller plus vite et donc plus de fourmis l'emprunteront par rapport à l'autre (cas *3*) dans un laps de temps donné[3]. Et ainsi, plus de phéromone sera déposée sur ce chemin que sur l'autre et donc, de plus en plus de fourmis l'utiliseront et ainsi de suite. La société finira donc par retrouver le chemin optimal (cas *4*).

Un exemple : Le problème du voyageur de commerce

Dans cette partie, on s'intéresse plus particulièrement à la résolution du problème du voyageur de commerce en utilisant les *colonies de fourmis*. Pour cela, un algorithme a été proposé dans [DMC96, DG97].

Une instance de ce problème est représentée par un graphe $D = (V, E)$ où V est l'ensemble des n villes (représentées par les sommets) et E l'ensemble des arêtes entre les villes. Les arêtes sont valuées pour représenter la distance entre les villes. Soit $b_i(t), 1 \leq i \leq n$, le nombre de fourmis dans la ville i à l'instant t (le temps est discrétisé). $m = \sum_{i=1}^{n} b_i(t)$ est donc le nombre total de fourmis qui est constant au cours du temps.

A chaque étape, chaque fourmi choisit la ville où aller suivant les critères suivants:

- Le choix est une fonction probalité de la distance de la ville et de la quantité de phéromone présente sur le chemin menant à cette ville.
- Il est interdit d'aller dans une ville déjà visitée tant qu'un tour n'a pas été fini.

[3] On suppose par exemple qu'une fourmi emprunte le même chemin pour l'aller que pour le retour.

Une fois un tour effectué, la fourmi “dépose” sa substance sur toutes les arêtes visitées. Chaque fourmi dépose au total la même quantité de phéromone donc plus un chemin sera court, plus la densité de substance déposée par la fourmi sera élevée. Soit $\tau_{ij}(t)$ la quantité de substance à l'instant t sur l'arête (i, j). A chaque étape, toutes les fourmis se déplacent d'une ville à l'autre. Toutes les fourmis achèvent donc un tour toutes les n itérations de l'algorithme. A ce moment-là, la quantité de substance est mise à jour: $\tau_{ij}(t+n) = \rho\tau_{ij}(t) + \delta\tau_{ij}$.

$1-\rho$ correspond à l'*évaporation* de la phéromone entre les instants t et $t+n$. $\delta\tau_{ij} = \sum_{k=1}^{m} \tau_{ij}^k$ où τ_{ij}^k est la quantité par unité de longueur de la phéromone déposée sur l'arête (i, j) par la fourmi k entre les instants t et $t+1$.

$$\tau_{ij}^k = \begin{cases} \frac{Q}{L_k} \text{ si la fourmi } k \text{ utilise l'arête } (i,j) \text{ dans ce tour.} \\ 0 \text{ sinon} \end{cases}$$

Q est une constante et L_k la longueur du tour de la fourmi k.

Ainsi, à chaque étape, la substance déposée par une fourmi est inversement proportionnelle à la distance parcourue donc proportionnelle à la qualité de la solution. Avant la première itération, toutes les arêtes reçoivent la même quantité de phéromone. Ainsi, au cours des itérations, si la population est assez nombreuse, le plus court chemin est de plus en plus marqué par la phéromone.

Particularités

Il s'agit d'un algorithme au parallélisme intrinsèque élevé mais difficilement adaptable à n'importe quel problème.

Cet algorithme a comme avantage de présenter, comme beaucoup d'autres algorithmes évolutionnaires, un parallélisme intrinsèque élevé au niveau des agents que sont les fourmis. Cependant, de par sa structure, il n'est pas facilement transposable à d'autres problèmes, surtout quand la représentation du problème sous forme de graphe n'est pas évidente ou ne modélise pas fidèlement le problème.

2.5 Conclusion

Pour résoudre de manière approchée un problème combinatoire, il existe nombre d'algorithmes d'horizons très divers. Suivant ce que l'utilisateur recherche, il s'orientera vers tel ou tel algorithme.

Le choix de l'algorithme dépend en outre du problème et des critères de rapidité.

D'une manière générale, si on dispose d'un algorithme spécifique efficace, c'est celui-ci qui sera privilégié. Sinon, si on recherche avant tout un algorithme très rapide, on privilégiera l'utilisation d'heuristiques. Sinon, si le but est d'obtenir une très bonne solution, l'utilisation d'algorithme de séparation-évaluation ou d'algorithme génétique sera indiquée. Ces deux algorithmes ont l'avantage d'améliorer au cours de leur exécution la meilleure solution trouvée ainsi que d'être adaptés au parallélisme. Un avantage des algorithmes de séparation-évaluation est de fournir en même temps une estimation de la qualité de la solution.

Cependant, les algorithmes génétiques semblent très intéressants, notamment par leur robustesse par rapport aux différentes instances du problème. En effet, la qualité de la solution trouvée dépend peu de l'instance du problème. Le chapitre suivant les détaille et des exemples d'utilisation sont expliqués.

CHAPITRE 3 Les algorithmes génétiques

3.1 Principes

Il s'agit d'une famille d'algorithmes basés sur l'évolution des espèces. L'évolution d'une espèce est simplement une suite successive d'améliorations afin qu'elle soit la mieux adaptée au milieu dans lequel elle évolue. Cette adaptation est réalisée grâce à la sélection naturelle et aux mécanismes de la reproduction [Hol92, Gol89, Koz92].

En effet, chaque animal, ou végétal, est le résultat d'une série d'évolutions afin d'assurer le but principal qu'est la survie de l'espèce. Pour cela, l'espèce évolue afin d'être le plus possible adaptée à son environnement et pour réagir à d'éventuelles modifications (climat, nouveaux prédateurs, maladies, surpopulation,...).

Les algorithmes génétiques reposent sur le principe de la sélection naturelle et de l'évolution des espèces.

Cette sélection naturelle est basée essentiellement sur la reproduction et sur le codage génétique qui stocke les informations décrivant l'individu sous forme de **gènes**. Cette amélioration, ou évolution de l'espèce, repose sur trois caractéristiques de la reproduction. Tout d'abord, pour se reproduire, l'individu doit atteindre un certain stade de maturité, ce qui empêche les individus peu adaptés de se reproduire. En outre, lors des périodes de reproduction, notamment chez les animaux évolués comme les mammifères, le choix des partenaires privilégie les animaux les plus adaptés à leur milieu, c'est ce que l'on appelle la **sélection** naturelle . Ainsi, la reproduction favorise les individus les plus sains.

Lors de la reproduction, il existe deux phénomènes importants qui la caractérisent. Tout d'abord, les enfants possèdent des gènes avec des caractéristiques provenant de chacun des deux parents. Ceci provient du **croisement** entre les gènes des deux parents. Ensuite, lors de la reproduction, des "erreurs" peuvent survenir. Ces erreurs sont la modification des informations contenues dans les chromosomes. Ces modifications peuvent avoir des conséquences plus ou moins importantes, soit en améliorant l'individu, soit en le dégradant. Il s'agit du phénomène de **mutation**.

Ce chapitre a pour but de présenter la structure et les bases des différents algorithmes génétiques existants.

Populations: P_0, P_1
Individus: $P_0^1, \ldots, P_0^n$
Pour $1 \leq i \leq n$ $P_0^i \leftarrow$ solution aléatoire
tant que fin non décidée
  Pour $1 \leq i < \frac{n}{2}$ croisement(P_0^{2*i}, P_0^{2*i+1})
  Pour $1 \leq i \leq n$ mutation(P_0^i)
  $P_1 \leftarrow$ sélection(P_0)
  $P_0 \leftarrow P_1$

Figure 3.1: Algorithme génétique type

3.2 Algorithme type

La figure 3.1 représente la structure générale d'un algorithme génétique. Comme expliqué précédemment, il s'agit de simuler l'évolution des espèces. Pour ce faire, on simulera la reproduction des individus.

La **population** est un ensemble d'**individus**. Un individu est caractérisé par ses **chromosomes**. La phase essentielle à l'évolution est la reproduction et elle est guidée par la sélection naturelle. La sélection naturelle est simulée par la fonction de sélection. Le but de cette fonction est de "favoriser" les "bons" individus.

L'algorithme génétique est basé sur les fonctions de sélection, de croisement et de mutation.

La sélection utilise un opérateur appelé fonction de santé ou d'évaluation qui associe à un individu sa qualité. Généralement, la qualité est représentée par un réel ou un entier.

Dans un algorithme génétique, la population évolue grâce à trois phases principales: La sélection favorise les "bons" individus en supprimant ceux "mauvais". Au besoin, les "bons" individus peuvent être dupliqués. La mutation altère l'individu sur lequel elle s'applique tandis que le croisement génère pour chaque couple deux enfants qui héritent des caractéristiques de chacun de leurs parents.

En fait, les opérateurs de croisement et de mutation s'appliquent seulement avec une certaine probabilité. Sinon, les individus sont conservés tels quels.

3.3 Variantes

En fait, les algorithmes génétiques sont une famille d'algorithmes, basés autour des mêmes idées. Cependant, il existe beaucoup de variantes possibles suivant la représentation choisie, les opérateurs de croisement, de mutation et de sélection,... Cette partie présente les choix les plus courants qui définissent les variantes les plus usitées.

Beaucoup de variantes existent.

3.3.1 Représentation

Généralement, un chromosome est une chaîne ou un ensemble de chaînes, souvent de bits ou d'entiers, parfois de réels. C'est ce qui se rapproche le plus

de la double hélice d'ADN des chromosomes biologiques.

D'autres représentations sont aussi possibles. Ainsi, pour [WK95], un chromosome est une matrice carrée et on peut imaginer nombre de représentations possibles, comme par exemple des arbres,... [Wal96]. Cependant, la représentation choisie doit satisfaire nombre de contraintes et essayer d'atteindre certains objectifs. Tout d'abord, il faut tenir compte de l'ensemble des solutions possibles : il peut être intéressant de restreindre l'ensemble des solutions représentables (par exemple en se servant des éventuelles symétries ou des équivalences). Cependant, il faut veiller à ne pas entraver l'algorithme dans sa recherche en lui empêchant d'explorer certains sous-espaces en rendant non représentables les solutions de ce sous-espace.

Il n'existe pas de représentation fixée mais, en général, les chromosomes ont la forme de chaîne(s).

Ensuite, une autre contrainte concerne les opérateurs de mutation et de croisement. Il faut que la représentation s'adapte bien à ces deux opérateurs car le choix de ceux-ci repose sur la structure des chromosomes. En effet, les opérateurs de mutation et de croisement utilisent la structure même des chromosomes. Ainsi, les opérateurs les plus courants sont prévus pour être appliqués sur des chromosomes en forme de chaînes en se servant de cette structure pour obtenir des performances.

Outre les soucis de performances, il est préférable que la représentation soit la plus "proche" possible de la solution réelle. Ceci facilite la conception de l'algorithme et permet de mieux se rendre compte de ce qui se passe. Il est ainsi plus facile d'ajuster les paramètres.

3.3.2 Sélection

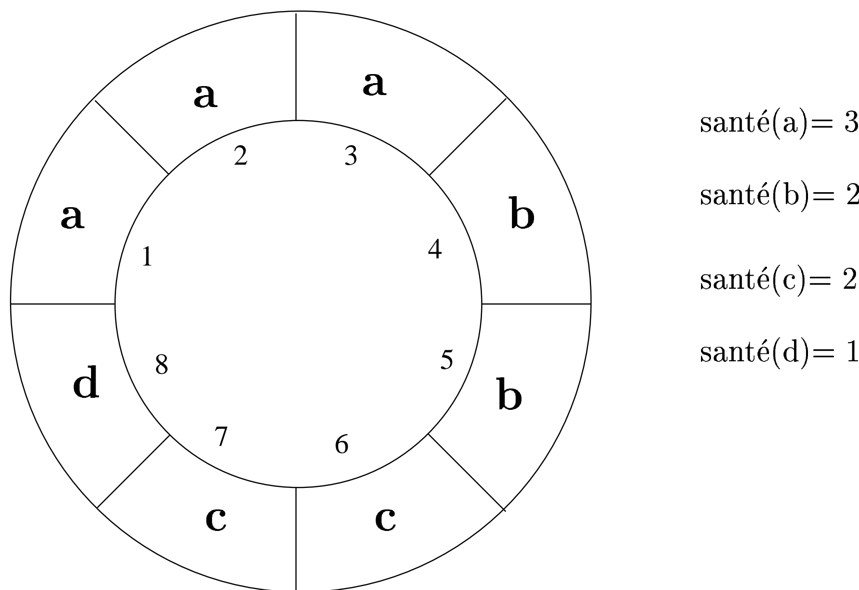


Figure 3.2: La sélection par roulette biaisée.

La sélection favorise les "meilleurs" individus.

C'est l'opérateur qui est chargé de "favoriser" les meilleurs individus. Deux principaux types de sélection existent : la sélection par tournois et celle par roulette "biaisée".

Un exemple de sélection par roulette biaisée où l'on cherche la valeur de santé la plus élevée est montré par la figure 3.2. Dans cet exemple, la population comporte 4 individus, la somme de leur santé vaut 8. On décompose donc la roulette en 8 cases. Chaque individu a pour nombre de cases la valeur de sa santé. La sélection consiste donc à lancer la roulette 4 fois et à chaque fois à insérer dans la nouvelle population l'individu choisi par la bille virtuelle.

Il existe aussi une autre sélection, celle par tournoi. Elle consiste à comparer deux individus et à ne garder que le meilleur des deux. Elle peut ainsi être faite à chaque création ou modification d'un individu. Elle peut donc être employée après chaque croisement ou chaque mutation pour déterminer si les nouveaux individus générés doivent être ou non retenus.

De plus, suivant les cas, on peut décider de toujours garder le meilleur individu, ou les deux meilleurs... De même, on peut choisir de ne jamais garder le plus mauvais individu. En outre, divers mécanismes peuvent être ajoutés pour garder une certaine diversité parmi la population comme par exemple limiter la duplication des meilleurs individus.

Généralement, la taille de la population est constante mais cela n'est pas une règle.

3.3.3 Croisement

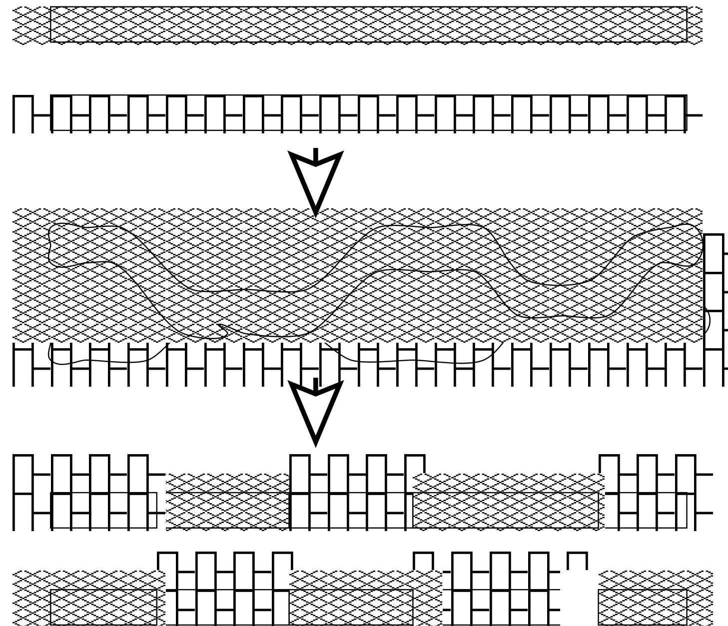


Figure 3.3: Le principe du croisement.

Le croisement génère des individus qui sont le résultat du mélange de plusieurs solutions.

Le croisement, à l'instar de son équivalent biologique, est chargé de construire un individu (ou une solution) qui soit le mélange de plusieurs solutions. Plusieurs, car rien ne limite le nombre de parents à deux et certaines implémentations utilisent des croisements à plus de deux parents mais ceci est rare.

Dans le croisement montré par la figure 3.3, les chromosomes (sous forme de chaînes) échangent des séquences (ou des gènes) entre eux. Cependant, suivant le problème donné, il peut exister des contraintes sur les chromosomes afin qu'ils représentent des solutions valides. Dans ce cas, le croisement (comme la mutation), doit générer des individus corrects. Des croisements courants seront expliqués à la section 4.8.

3.3.4 Mutation

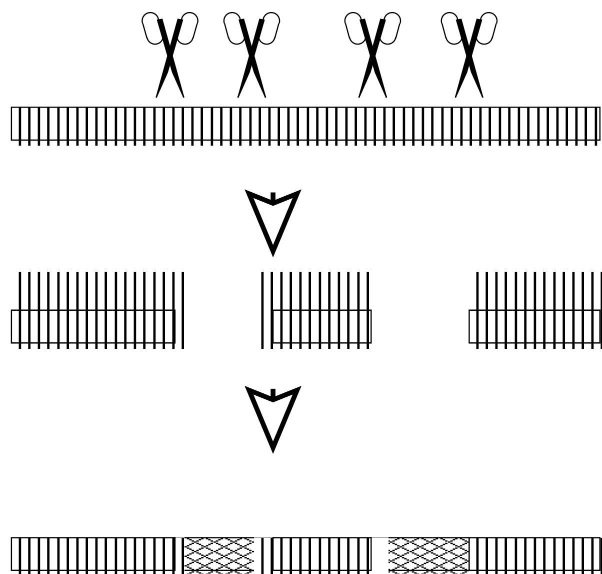


Figure 3.4: Exemple de mutation.

Cet opérateur a pour but de diversifier la population afin d'examiner d'autres parties de l'espace de recherche. Une mutation est donc l'altération (car elle a souvent un effet négatif, comme les mutations biologiques) d'une partie du chromosome. L'individu qui subit une mutation a donc des gènes qui sont modifiés.

La mutation est une modification aléatoire d'un individu.

La figure 3.4 montre un exemple de mutation. Le chromosome, représenté par une chaîne, est coupé en deux endroits. Les deux parties coupées peuvent être échangées ou reconstruites. Dans ce dernier cas, si le chromosome est une chaîne de bits ou de nombres, la reconstruction se fait en général par le stockage de valeurs déterminées aléatoirement.

3.3.5 Le critère d'arrêt

Différentes politiques sont possibles pour décider quand arrêter l'algorithme car les algorithmes génétiques, comme tous les algorithmes évolutionnaires, sont, dans l'absolu, sans fin. La politique la plus courante et la plus simple est sans conteste d'effectuer un nombre prédéfini d'itérations mais d'autres sont possibles. On peut distinguer trois grandes familles de critères d'arrêt :

1. le temps ou le nombre d'itérations voulu est atteint.

2. la fonction de santé est constante depuis quelque temps.

3. la population est dominée par quelques individus.

La famille de critères d'arrêt 1 représente la majeure partie des critères d'arrêt employés. En effet, ils sont très faciles à mettre en oeuvre. Le temps est lié au nombre d'itérations suivant la taille des données.

La famille de critères d'arrêt 2 est aussi assez souvent utilisée. Cela permet de caractériser un algorithme qui n'arrive pas à trouver de meilleures solutions et qui est dans une configuration de "minimums locaux".

En général, l'algorithme s'arrête au bout d'un temps ou d'un nombre de générations donné.

Les critères d'arrêt de type 3 sont en général assez complexes et peu fiables. En effet, suivant les opérateurs choisis, il est très difficile, voire impossible de déceler si, effectivement, la population est dominée par un ensemble d'individus ou non. Il faut pouvoir suivre leur trace durant plusieurs générations afin de savoir si c'est vraiment le cas ou pas. Ce type de critère est très rarement employé. En fait, son intérêt serait plutôt de décider quand modifier quelques paramètres de l'algorithme ou de déclencher une procédure spécifique afin de diversifier la population.

La politique que nous utiliserons souvent sera d'arrêter l'algorithme au bout d'un temps donné. En effet, cela correspond au souhait de l'utilisateur qui, s'il ne recherche pas forcément l'interactivité, est prêt à laisser un certain temps l'algorithme monopoliser une ressource mais ce temps doit être borné.

En fait, comme on ne sait pas ce qui va se passer dans la suite de l'algorithme, il n'y a pas de politique qui se justifie plus qu'une autre si ce n'est par le souhait de l'utilisateur.

3.4 Champs d'utilisation

Les algorithmes génétiques sont utilisés dans des problèmes très divers [QPPW98], qui vont des problèmes formels comme le voyageur de commerce à des problèmes plus concrets, comme la conception de cames pour les moteurs....

Les algorithmes génétiques sont souvent employés pour leur simplicité de conception.

Cette popularité s'explique par plusieurs facteurs. Tout d'abord, ce sont des algorithmes simples à mettre en oeuvre. En effet, dans les cas les plus courants, le travail nécessaire pour développer un code est limité (en général, on manipule des chaînes) et l'on peut utiliser des bibliothèques [Wal96]. Cette simplicité est confortée par le fait que peu de connaissances du problème soient nécessaires pour développer un algorithme génétique. On doit juste connaître :

- comment générer aléatoirement une solution.
- comment quantifier la qualité de la solution.
- comment modifier de manière aléatoire une solution.
- comment mixer deux solutions.

Le premier point est indispensable pour générer la population initiale qui doit être la plus diverse possible. Ensuite, il est nécessaire de pouvoir estimer la qualité d'une solution (de manière absolue si l'on utilise une sélection par roulette biaisée, de manière relative dans le cas de la sélection par tournoi où l'on compare les solutions deux à deux). Les deux derniers points permettent de construire la mutation et le croisement. En résumé, dès qu'on est capable de construire pas-à-pas une solution (heuristique gloutonne par exemple), on peut obtenir facilement un algorithme génétique.

En outre, ce sont des algorithmes assez robustes de par leur nature. Les résultats sont assez indépendants des paramètres dès lors que ceux-ci ont des valeurs proches des valeurs courantes utilisées. Pour fixer un ordre d'idée, on dispose souvent d'une population de l'ordre de la centaine d'individus. Le croisement est généralement appliqué avec une forte probabilité (de l'ordre de 0.7 à 1) tandis que la mutation n'a qu'une probabilité assez faible (0,1 à 0,3).

3.5 Utilisation conjointe d'autres algorithmes

Les algorithmes génétiques tels que décrits précédemment ont montré leur efficacité dans nombre de problèmes d'horizons très divers. Cependant, ce type d'algorithme nécessite en général un grand nombre d'évaluations de la fonction de calcul de santé (grand nombre d'itérations, population importante) pour obtenir de bons résultats. Pour cette raison, leur efficacité est forte quand la fonction de calcul de santé est simple et rapide en terme de temps de calcul tout comme les opérateurs de croisement et de mutation. Ainsi, leur efficacité par rapport au temps de calcul nécessaire peut être faible comparée à d'autres algorithmes, notamment ceux dédiés à une application particulière.

On peut rendre les algorithmes génétiques plus performants en les croisant avec d'autres méthodes.

Pour rendre les algorithmes génétiques performants en terme de coût de calcul, une solution consiste à construire des algorithmes **hybrides** [RB94, FF94]. L'hybridation revient à combiner plusieurs algorithmes de nature différente. Par exemple, on peut croiser un algorithme génétique avec une heuristique, mais aussi un algorithme de séparation-évaluation avec un recuit simulé,... En général, le résultat est meilleur que celui qui serait obtenu par une des méthodes choisies.

Les algorithmes génétiques n'utilisant pas de connaissances particulières [Gol89], ils sont souvent couplés soit à des méthodes de recherche de minimums locaux soit à des heuristiques. En effet, la puissance des algorithmes génétiques provient du fait qu'ils sont capables de balayer de manière globale l'espace des solutions contrairement à la méthode de descente du gradient ou aux heuristiques qui explorent une petite zone de cet espace. L'utilisation d'heuristiques permet d'introduire des techniques performantes propres au problème à résoudre. Nous allons détailler par la suite les deux méthodes pour coupler un algorithme génétique à une heuristique.

En effet, il existe deux méthodes pour effectuer ce “mélange”. Leur différence essentielle est dans le choix de ce que représente (code) le chromosome. Suivant qu'il code directement une solution du problème ou non, on parlera de représentation directe ou de représentation indirecte.

3.5.1 La représentation indirecte

Il s'agit de la méthode de couplage la plus simple à mettre en œuvre, surtout quand on dispose d'un algorithme de liste. On a vu précédemment qu'un tel algorithme utilise de manière implicite un ordre sur les éléments qui composent la solution. L'idée est d'utiliser un algorithme génétique pour déterminer l'ordre le plus approprié sur les éléments afin que l'algorithme de liste donne la meilleure solution possible [AD96].

Un algorithme génétique à représentation indirecte utilise comme fonction d'évaluation une heuristique.

L'algorithme génétique est donc un algorithme classique dont la fonction d'évaluation est modifiée. Cette fonction déduit du chromosome à évaluer un ordre total sur les éléments du problème et construit une solution en utilisant l'heuristique ou l'algorithme de liste. La valeur de santé du chromosome est la valeur de la solution correspondante. L'algorithme utilise donc une représentation sous forme de chaînes ou de listes ordonnées, où chaque élément est un élément du problème. L'ordre de chaque élément est donné par son ordre dans la liste.

L'avantage de ce type de représentation est d'être facile à mettre en œuvre et d'avoir une bonne efficacité en général [RSM98]. Ceci sera montré dans la suite de ce document grâce aux tests réalisés.

3.5.2 La représentation directe

La méthode précédente a l'inconvénient de reposer intégralement sur une seule heuristique ou un unique algorithme de liste. Souvent, l'utilisation d'un tel algorithme peut limiter l'espace de recherche. En effet, l'espace de recherche n'est que l'ensemble des solutions que peut générer l'algorithme de liste ou l'heuristique utilisé.

Pour essayer de ne pas avoir les inconvénients de la méthode précédente, les algorithmes à représentation directe se basent sur un algorithme génétique classique et conserve la même représentation: c'est-à-dire chaque chromosome code une solution du problème de manière directe et la fonction de santé reste inchangée [GGRG85].

La représentation directe repose sur une modification des opérateurs de croisement et/ou de mutation.

Les modifications portent sur, au moins, un des opérateurs d'évolution (croisement, mutation). Au lieu de faire des changements purement aléatoires comme dans les algorithmes génétiques classiques, les changements sont effectués en utilisant une heuristique ou un algorithme de liste tout en restant dans l'"esprit" génétique: les modifications ne doivent pas être trop "dirigistes" afin que l'exploration de l'espace des solutions ne soit pas trop limitée. En outre, il faut que les opérateurs génèrent des individus qui gardent certaines caractéristiques de leur(s) parent(s).

Si ce type de représentation nécessite plus de travail pour sa mise au point, elle a l'avantage d'être plus évolutive et plus souple d'emploi [RSM98]. En effet, on peut facilement utiliser plus d'une heuristique dans le même algorithme ou modifier l'heuristique. Mon travail de recherche a beaucoup porté sur ce type de représentation et, par la suite, je présenterai plusieurs algorithmes hybrides de ce type.

3.6 Le parallélisme

Le parallélisme est un axe de recherche prisé dans ce domaine. En effet, la taille des problèmes considérés impose des calculs importants. Le parallélisme, soit issu de machines multiprocesseurs, soit de réseaux de stations, offre à faible coût une puissance de calcul inégalée, du moins en théorie. En effet, il faut pouvoir modifier chaque algorithme afin que celui-ci puisse tirer profit du parallélisme offert. Ainsi, la recherche s'est, depuis quelques années, orientée vers des algorithmes parallèles pour ce genre de problèmes, notamment concernant les algorithmes de séparation-évaluation qui offrent de par leur structure un parallélisme potentiel élevé [Cor97, BCD+96].

Les algorithmes génétiques offrent un parallélisme potentiel élevé.

De part leur nature, les algorithmes génétiques se prêtent bien à une implémentation parallèle [CP95]. Suivant le problème et le matériel disponible, plusieurs techniques de parallélisation sont possibles. Les deux techniques les plus importantes sont expliquées ci-après.

3.6.1 Le style maître-esclave

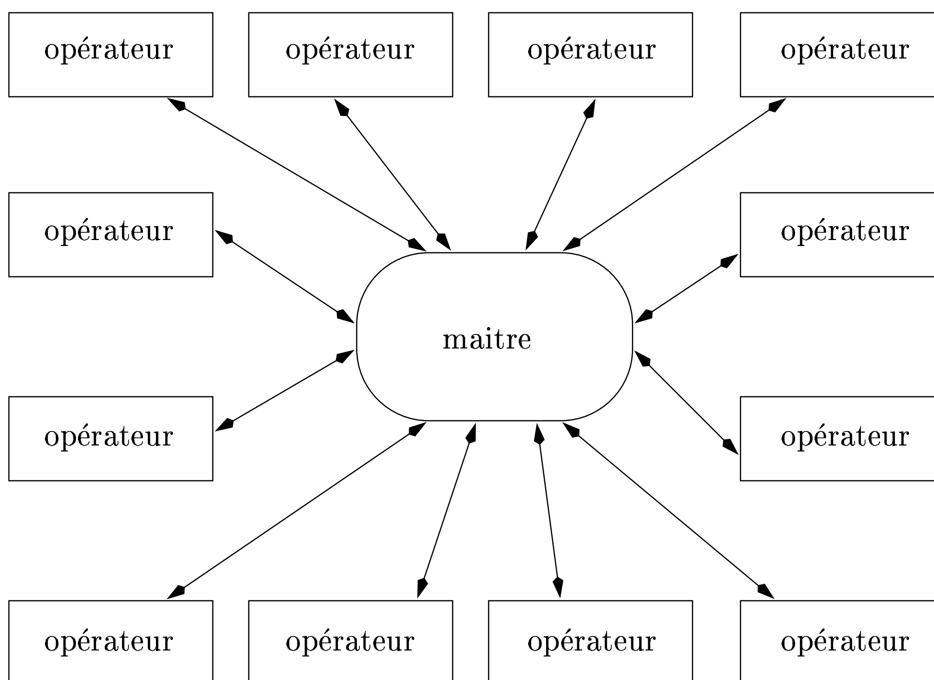


Figure 3.5: Algorithme parallèle de style maître-esclave.

Le style maître-esclave est peu usité car peu performant.

Un algorithme génétique est composé de plusieurs opérateurs ou fonctions (croisement, mutation, calcul de la santé, sélection ...). Si en général la sélection est très rapide, les autres opérateurs peuvent être lents (par exemple le calcul de la santé[1]). Dans ce cas, un noeud de calcul va être considéré comme maître et gérer l'algorithme et va utiliser les autres noeuds comme des esclaves pour calculer les fonctions gourmandes en temps de calculs. Ceci est illustré par

[1] cas des algorithmes à représentation indirecte, par exemple.

la figure 3.5. Dans ce cas, l'exécution est similaire à celle séquentielle, sauf en ce qui concerne le générateur de nombres aléatoires. Ce type d'implémentation est peu utilisé car il est peu performant et peu extensible à un nombre élevé de processeurs. De même, les communications sont critiques dans ce style de parallélisation [CP95].

3.6.2 Les îles

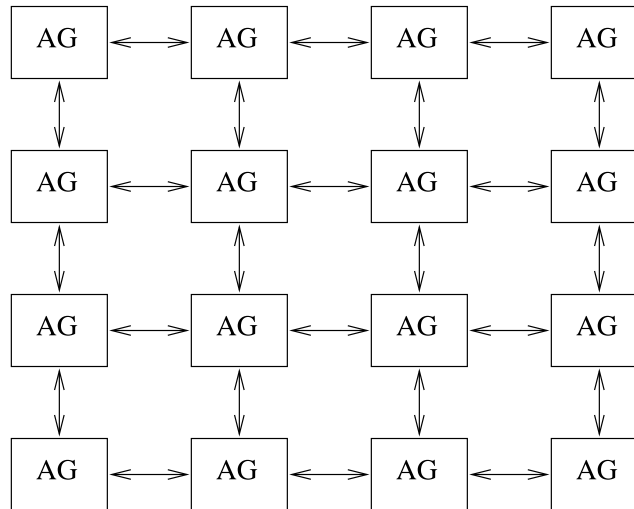


Figure 3.6: Algorithme parallèle de style îles.

Il s'agit d'une technique qui s'inspire directement du milieu biologique. Dans la nature, à cause des frontières naturelles délimitant des territoires, les membres d'une même espèce sont divisés en populations, de part et d'autre des mers, des massifs montagneux, des rivières,. . . , populations dont les échanges en termes d'individus sont rares.

C'est une technique purement génétique.

Dans notre cas, les territoires sont assimilés aux noeuds de l'ordinateur et notre algorithme génétique parallèle ne consiste, ni plus ni moins, qu'à une population d'algorithmes génétiques séquentiels, chacun utilisant un noeud de calcul [PPMR95]. L'efficacité de l'ensemble vient des communications entre les noeuds. Plusieurs schémas de communications sont possibles. En général, seul le meilleur individu de chaque population est transmis. Les communications peuvent se faire de deux manières disctinctes : soit le meilleur individu est *diffusé* à toutes les autres populations, soit il est envoyé seulement à un voisinage restreint comme c'est le cas de l'exemple de la figure 3.6.

Elle permet aussi de garder une certaine diversité dans la population.

Cependant, cette technique peut aussi être intéressante dans le cas séquentiel pour garder une certaine diversité dans la population. En subdivisant la population et en limitant les échanges, on évite qu'un type de solution envahisse toute la population. En effet, si le meilleur individu d'une île peut facilement dominer la sous-population correspondante, les échanges limités avec les autres sous-population va retarder sa propagation aux autres îles. En outre, les autres îles risquent aussi d'être dominées par des individus tout aussi adaptés, mais différents.

3.7 Conclusion

Les algorithmes génétiques présentent des caractéristiques intéressantes les rendant populaires, notamment grâce à leur facilité d'emploi. Leurs points forts sont une certaine robustesse due à l'utilisation de l'aléatoire, une facilité de mise en oeuvre mais aussi un parallélisme intrinsèque important.

Les algorithmes génétiques sont faciles à utiliser, efficaces et facilement modifiables.

Les algorithmes génétiques sont une grande famille et deux membres de cette famille peuvent être très différents. Cependant, ils peuvent être moins performants que des méthodes dédiées à un problème spécifique, comme par exemple, le partitionnement multi-niveau.

Cependant, la grande liberté qu'ils offrent permet de construire facilement des algorithmes hybrides grâce à l'utilisation conjointe de méthodes évolutionnaires et de méthodes heuristiques par exemple.

Dans les trois prochains chapitres, je présente divers algorithmes génétiques hybrides pour résoudre trois problèmes combinatoires distincts. Mon travail de recherche a porté essentiellement sur la conception et la mise au point de ces algorithmes. Les implémentations réalisées sont volontairement séquentielles. En effet, la parallélisation d'algorithmes génétiques, si elle a montré son efficacité, n'améliore en rien la qualité intrinsèque de l'algorithme séquentiel (son rapport *qualité de la solution/quantité de calculs*). Elle permet seulement d'accélérer la vitesse d'exécution de l'algorithme donc d'obtenir une solution plus rapidement (ou de trouver une meilleure solution dans un même laps de temps). Mais toute solution trouvée par un algorithme parallèle peut être fournie par une simulation séquentielle de celui-ci en un temps plus long (abstraction faite des problèmes matériels comme la mémoire,...). Il faut donc, avant toute parallélisation, examiner les différentes méthodes séquentielles. Comme les algorithmes génétiques hybrides sont avant tout des algorithmes génétiques, on peut supposer que les techniques de parallélisation peuvent s'appliquer avec efficacité, comme pour les autres algorithmes génétiques.

CHAPITRE 4

Ordonnancement

4.1 Introduction

Dans ce chapitre, on va s'intéresser au problème *de l'ordonnancement de tâches sur machine parallèle*. Il s'agit d'un problème qui se pose lors de l'exécution d'un programme parallèle sur un ordinateur parallèle. Il fait partie de la grande famille des problèmes d'ordonnancement [Bou94].

Bien sûr, suivant la modélisation faite et le problème initial, le problème peut être quelque peu différent. Pour cela, il faut bien spécifier quelles sont les **hypothèses** prises.

Notre problème est le placement **statique** de tâches sur une machine multiprocesseur **homogène**, c'est-à-dire que l'on dispose d'un programme parallèle découpé en tâches et l'on veut, avant son exécution, savoir où et quand chaque tâche sera exécutée. Les communications nécessaires entre les tâches induisent des dépendances entre elles. La machine cible est une machine multiprocesseur où tous les processeurs sont identiques et reliés entre eux par un réseau de communication. La machine est donc **homogène**.

On s'intéresse au placement statique de tâches sur machines multi-processeurs.

On supposera que le coût d'une communication est nul si elle est intraprocesseur. Sinon, son coût sera indépendant de la distance et sera une fonction affine du volume de données à transmettre. Ces suppositions sont raisonnables pour les machines parallèles actuelles, comme, par exemple, les réseaux de stations mais aussi pour les machines parallèles spécifiques.

Les tâches sont le résultat de la décomposition d'un programme parallèle en blocs ne contenant pas d'envoi ou de réception de message. Un programme parallèle est donc un ensemble de tâches reliées par des communications. Ces communications peuvent commencer à la fin de l'exécution de la tâche émettrice et doivent être terminées avant que la tâche réceptrice ne puisse commencer.

De nombreux algorithmes ont été utilisés pour résoudre ce problème. L'intérêt de l'ordonnancement statique est de disposer d'informations précises (du moins en théorie) sur le problème contrairement à l'ordonnancement *dynamique*. Cependant, comme il faut que le gain apporté par l'algorithme pour trouver une bonne solution soit supérieur à celui du surcoût engendré par cet algorithme, de nombreuses heuristiques ont été mises au point [CCLL95]. Cependant, compte tenu du coût élevé d'utilisation des machines parallèles et du coût très faible des machines classiques, il peut être intéressant de monopoliser

une machine séquentielle[1] pour optimiser l'utilisation d'une machine parallèle. De plus, on utilise souvent le même programme sur des données différentes d'où l'intérêt à optimiser son exécution.

D'autres problèmes d'ordonnancement plus ou moins proches ont conduit à développer des techniques coûteuses en calculs comme les algorithmes de séparation-évaluation [PC94b, Cor97] et les algorithmes évolutionnaires [HAR94, AD96, WK95]. Nous allons, dans ce chapitre, présenter, expérimenter et comparer plusieurs approches à base d'algorithmes génétiques et d'heuristiques.

4.2 Modélisation

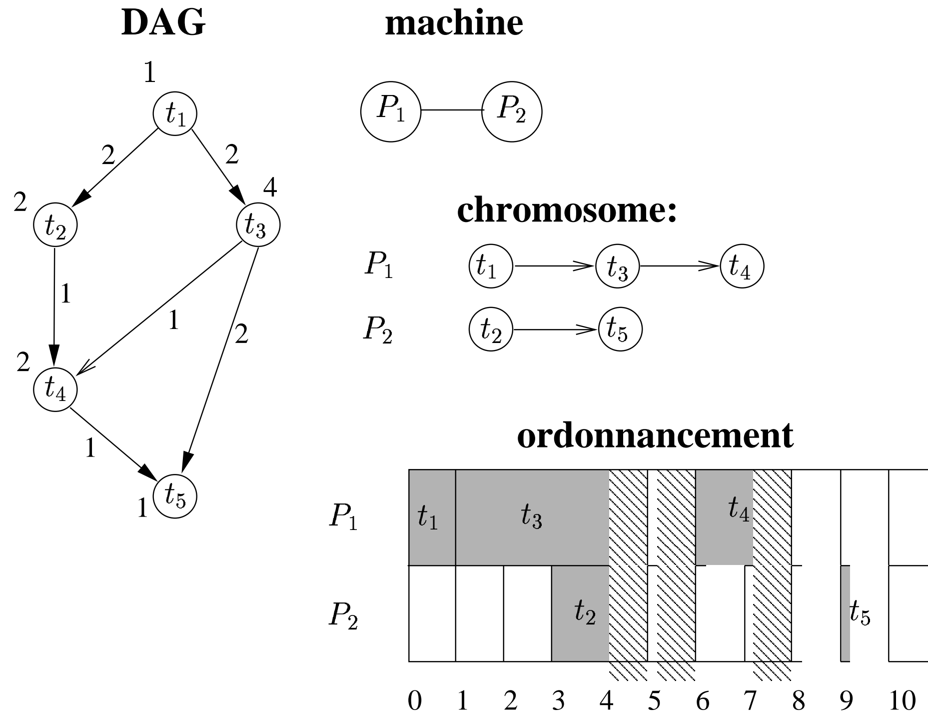


Figure 4.1: Exemple d'un ordonnancement valable.

Le problème est modélisé par un graphe orienté acyclique valué.

La figure 4.1 représente un tel problème. L'ordinateur parallèle est représenté par un ensemble homogène P de m processeurs : $P = \{P_1, \ldots, P_m\}$. Le programme parallèle est représenté par un graphe orienté acyclique (DAG) $D = (T, A)$. L'ensemble T des noeuds représente l'ensemble des n tâches. A chaque tâche t_i est associée sa durée d_i. Une arête (t_i, t_j) représente le fait que la tâche t_i envoie à la fin de son exécution un message à la tâche t_j et celle-ci doit avoir terminé sa réception avant de pouvoir commencer son exécution. Dans ce cas, on dit que t_j est un **successeur immédiat** de t_i et t_i un **prédécesseur immédiat** de t_j. Chaque arête est valuée par la taille $s(i, j)$ du message à communiquer. L'ordonnancement est représenté sous la

[1] Une station de travail la nuit par exemple.

forme d'un graphique de Gantt [DT96].

Toutes ces informations sont connues à la compilation du programme, avant son exécution. Pour calculer la durée nécessaire pour transmettre un message, nous utilisons le modèle $\tau L + \beta$ où β est la durée d'initialisation et τ celle de transfert d'une information de taille unité.

Un **ordonnancement** est un vecteur $o = \{o_1, \ldots, o_m\}$ où o_i est la liste *ordonnée* des couples (tâche-date de début) du processeur i.

Définition 4.2.1
Un noeud source (respectivement puits) est un noeud sans prédécesseur (respectivement successeur).

Généralement, pour des raisons pratiques, on considère que le graphe admet un unique noeud source et un unique noeud puits[2].

4.3 Etat de l'art

Ce problème a déjà été étudié de manière approfondie par nombre de chercheurs. Les solutions proposées font appel à un nombre impressionnant de techniques différentes [CCLL95]. Cependant, ce problème appartient, dans son cas général, à la classe des problèmes *NP-difficile* [MKTM94].

4.3.1 Les heuristiques

Nous détaillons ici cette famille d'algorithmes car notre approche est basée sur ce type d'algorithme. Un algorithme de ce type construit la solution *pas-à-pas*. A chaque étape, l'ensemble des tâches est subdivisé en trois parties:

- les tâches **ordonnancées**, c'est-à-dire que l'on sait où et quand elles seront exécutées.
- les tâches **ordonnançables**, dont toutes les tâches prédécesseurs sont ordonnancées.
- les tâches **non-ordonnançables** qui ont au moins une tâche prédécesseur qui n'est pas une tâche *ordonnancée*.

Les heuristiques construisent la solution pas-à-pas en utilisant des règles.

Le fonctionnement de l'algorithme est simple: au départ, aucune tâche n'est ordonnancée, les tâches sans prédécesseur (les tâches sources) sont ordonnançables tandis que les autres ne le sont pas puisqu'une tâche prédécesseur n'est pas ordonnancée. A chaque pas, une tâche t est choisie parmi l'ensemble des tâches ordonnançables selon une règle R_1 et un processeur p est choisi selon une règle R_2. Cette tâche est ordonnancée sur le processeur p aussitôt que possible. Les trois ensembles de tâches sont alors mis à jour et l'algorithme continue tant que l'ensemble des tâches ordonnançables n'est pas vide.

Si la règle R_1 est basée sur des critères dont la valeur change au cours de la construction de la solution (date d'introduction,...) il s'agit d'une heuristique. Par exemple, si $R1$ consiste à choisir une tâche dont la date d'introduction est minimale, il s'agit d'une heuristique de type gloutonne. En effet, à chaque

[2] Si ce n'est pas le cas, on rajoute deux noeuds fictifs de durée nulle.

étape de la construction, l'algorithme calcule ou recalcule pour chaque tâche ordonnançable sa date d'introduction au plus tôt. Par contre, si cette règle est basée sur un ordre déterminé à l'avance entre les tâches, il s'agit d'un algorithme de liste[3]. Ainsi, si la règle $R1$ consiste à prendre les tâches dans un ordre prédeterminé (par exemple suivant la longueur du plus long chemin entre la tâche et une tâche sans succeseur), l'heuristique est un algorithme de liste.

DSC

Il s'agit d'une heuristique développé pour ce problème [YG92]. Cette heuristique construit la solution en deux phases. La première consiste à affecter à chaque tâche un processeur virtuel. Le nombre de processeurs virtuel est illimité. La seconde phase consiste à regrouper entre eux des processeurs virtuels afin d'obtenir autant de processeurs virtuels que de processeurs physiques. La complexité de cette méthode est $O((v + e)logv$ où v est le nombre de tâches et e le nombre d'arêtes du graphe.

4.3.2 Algorithmes de chemins critiques

Définition 4.3.1
Un chemin critique est le chemin de plus long poids entre un noeud source et un noeud puits. Le poids d'un chemin est la somme des poids des noeuds traversés et des arêtes empruntées.

Pour construire la solution, ce type d'algorithme essaie de réduire la longueur du chemin critique en supprimant des communications (ce qui revient à mettre à zéro certaines arêtes) en plaçant sur le même noeud deux tâches. Cet algorithme appartient à la classe des heuristiques. Il est expliqué en détail par [CC88].

4.3.3 Les algorithmes de séparation-évaluation

Un algorithme de séparation-évaluation a été proposé par [Cor97] et cet algorithme prend en compte les temps de communcations. A chaque étape une tâche est ordonnancée. La séparation consiste à énumérer tous les couples possibles (tâche-processeur) qu'on puisse ordonnancer. Dans cette approche, il faut, pour obtenir un algorithme efficace, commencer par réduire le plus possible la taille du problème. Ceci est fait en détectant et supprimant les symétries, les situations équivalentes, ... Ensuite, il faut avoir une politique efficace du choix du sous-arbre qu'on va explorer. Dans ce cas, elle est basée sur l'opérateur qui minore la meilleure solution du sous-espace correspondant. Enfin, comme l'algorithme n'a pas le temps de trouver une solution optimale, on peut calculer à chaque noeud une solution en utilisant une heuristique. On peut donc dire qu'il s'agit d'un algorithme hybride [Cor97].

[3] Un algorithme de liste est aussi une heuristique, surtout quand le choix du processeur fait appel à des critères objectifs.

4.3.4 Un algorithme génétique

Hou, Ansari et Ren ont présenté en 1994 un algorithme génétique pour ce problème [HAR94]. Le problème est modélisé par un graphe de tâches mais les durées de communications sont négligées. Seules les dépendances induites par les communications sont prises en compte.

Cet algorithme génétique ne tient pas compte du coût des communications.

Les auteurs codent la solution par autant de chaînes qu'il y a de processeurs. Chaque chaîne représente pour un processeur donné l'ensemble **ordonné** des tâches que celui-ci devra exécuter. L'ordonnancement proprement dit (la date de démarrage de chaque tâche) est tout simplement déduit du chromosome par un algorithme de liste qui assigne chaque tâche à son processeur dès que possible. Cependant, n'importe quel ensemble de chaînes ne permet pas forcément de déduire un ordonnancement valide. En effet, la tâche t_3 de la figure 4.1 doit communiquer avec t_4 et donc, si ces deux tâches sont sur le même processeur, t_3 doit être exécuté avant t_4. Ceci implique que, si ces deux tâches sont sur la même chaîne dans un chromosome, t_3 doit être avant t_4 sur cette chaîne.

Pour éviter de se retrouver face à ce problème, un précalcul est fait : à chaque tâche est associée sa *hauteur*.

Définition 4.3.2
La hauteur d'une tâche est la longueur maximum du chemin maximum (en nombre de noeuds traversés) entre un noeud source du graphe et la tâche.

Les phases d'initialisation, de croisement et de mutation sont conçues pour que dans chaque chaîne, l'ordre des tâches respecte l'ordre des hauteurs. Si, sur chaque processeur, les tâches respectent l'ordre des hauteurs, l'algorithme de liste peut en déduire un ordonnancement valide. En effet, tous les prédécesseurs à une tâche ont une hauteur strictement inférieure à celle de cette tâche.

Cependant, cet algorithme présente des faiblesses. D'abord, lors de l'initialisation, tous les processeurs ne sont pas chargés équitablement. Ceci peut être remédié par une modification de la phase d'initialisation ou des opérateurs afin d'obtenir une charge uniforme sur les processeurs.

Cette méthode ne permet pas forcément de représenter des solutions optimales.

Ensuite, la solution optimale peut ne pas être atteignable. Le contre-exemple suivant en fournit la preuve. En effet, si on considère le programme décrit présenté à la figure 4.2, la représentation choisie ne permet pas d'exprimer une solution optimale sur une machine bi-processeur. Dans cet exemple, les temps de communications sont nuls mais le problème reste valide dans le cas contraire. En effet, à cause des contraintes sur les hauteurs, les tâches de hauteur 0 doivent être exécutées avant celles de hauteur 1 sur chaque processeur .

La solution optimale pour ce problème a une durée de 13 unités de temps (tâches t_1,t_4,t_7 et t_{10} sur le processeur P_1, et t_2,t_5,t_3,t_6,t_8 et t_9 sur le processeur P_2.). Or, les contraintes sur les hauteurs empêchent l'algorithme de [HAR94] d'atteindre cette valeur. En effet, pour obtenir cette durée, il faut que la tâche t_7 démarre au temps 2. Ceci implique que les tâches t_1,t_2, t_4 et t_5 soient exécutées avant. Or, il existe un processeur qui va accueillir t_3. A cause des contraintes des hauteurs, t_3 devra être exécutée avant toute tâche de hauteur 1 du même processeur. Comme, pour obtenir une solution optimale, il est

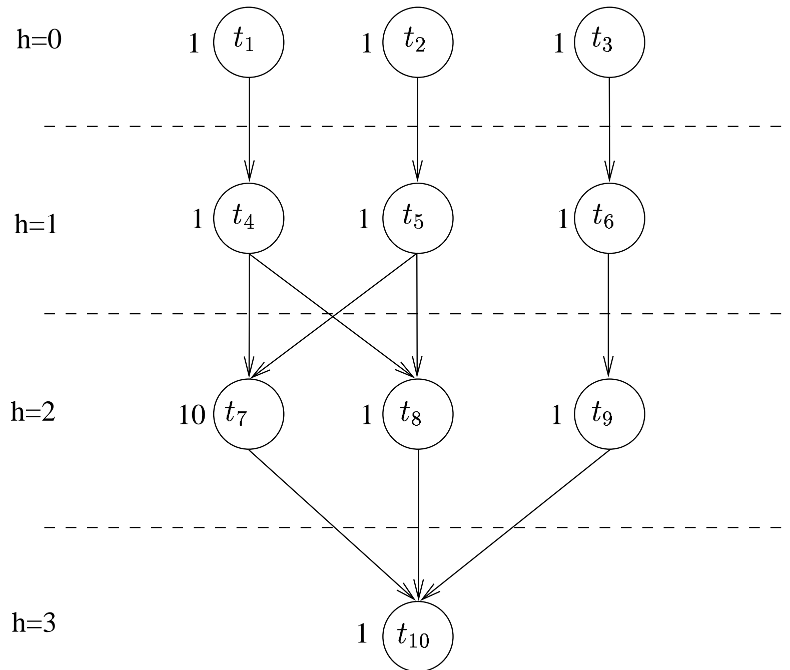


Figure 4.2: Graphe dont la solution optimale n'est pas atteignable.

nécessaire que t_4 et t_5 soient sur deux processeurs différents, cela implique que t_4 ou t_5 soient exécutés sur le même processeur que t_3 et qu'une autre tâche de hauteur 0. Dans ce cas, sur ce processeur, t_4 ou t_5 ne pourra pas commencer au temps 1. Ainsi, on démontre que l'algorithme proposé par HAR peut ne pas pouvoir représenter, et donc atteindre, une solution optimale dans certains cas.

De plus, les temps de communications ne sont pas pris en compte. Cela semble loin de la réalité et ces délais sont bien un problème des machines parallèles. Cependant, il est facile de modifier la fonction de santé afin de prendre en compte la durée de communication [CFR96].

4.4 Notre approche

4.4.1 Le modèle

Les coûts de communications sont pris en compte.

Pour être le plus réaliste possible, j'ai choisi de tenir compte des temps de communications. Ainsi, à chaque communication entre deux tâches est associée la taille des données à transférer. Le coût de la communication est en $\beta + \tau L$, L étant la taille du message, β le temps d'initialisation de l'envoi et τ le temps de transfert d'une information de taille unité, une fois l'initialisation effectuée. Cela permet de déduire le temps nécessaire à la communication connaissant la taille du message. En effet, la plupart des graphes utilisés comme jeu de tests sont valués par la taille des messages à échanger. En connaissant les paramètres de la machine cible, on peut en déduire le temps de communication.

Il serait facile, si besoin est, d'utiliser une modélisation plus complexe des communications. On pourrait facilement tenir compte de la topologie en ren-

dant le temps de communication dépendant des noeuds émetteur et récepteur.

4.4.2 La méthode

Comme expliqué précédemment, j'ai choisi d'utiliser les algorithmes génétiques pour résoudre ce problème. Cependant, outre les réglages des paramètres et de la sélection, il existe deux méthodes pour améliorer un algorithme génétique : soit on cherche de nouveaux opérateurs ou une nouvelle représentation soit on change radicalement l'algorithme pour mettre au point un algorithme hybride. La première voie n'offre pas, à priori, une perspective d'amélioration très significative comme, par exemple, le montrent les tests réalisés dans [FF94] dans le cas du problème de coloriage de graphe. Si l'on veut obtenir des résultats plus intéressants, il est nécessaire d'effectuer des changements plus radicaux et de mieux adapter l'algorithme au problème. C'est pour cette raison que la seconde voie, celle de l'hybridation, a fait l'objet de mes recherches. De plus, on cherche plutôt à optimiser la qualité de la solution que le temps d'exécution.

4.5 Notre jeu de tests

Dans le cas des problèmes appartenant à la classe $NP{-}difficile$, la qualité des algorithmes proposés se mesure principalement suivant le ratio *qualité de la solution / temps de calcul* puisque démontrer qu'on obtient la solution optimale avec un algorithme de complexité linéaire revient à démontrer que $P = NP$. Ceci conduit les utilisateurs à ne pas chercher la solution optimale mais une solution sous-optimale d'où ce critrère pour estimer la qualité. Pour ce faire, il est nécessaire de voir comment l'algorithme étudié se comporte sur plusieurs instances du problème et de le comparer à d'autres algorithmes connus. Dans la littérature, en l'absence de jeux de tests reconnus, la plupart des auteurs utilisent des graphes générés aléatoirement [HAR94]. Si cela peut sembler à première vue un choix judicieux, ces graphes représentent-ils les ensembles de tâches que l'on cherche à ordonnancer ? Autrement dit, sont-ils proches des instances réelles du problème ? En outre, la façon de générer ces graphes ne risque-t-elle pas d'influencer le comportement de l'algorithme ?

Le jeu de tests comprend des graphes synthétiques représentant des programmes réalistes sur une machine réelle.

Pour que notre jeu de tests soit le plus proche possible de la réalité, j'ai choisi d'utiliser l'outil ANDES-SYNTH [KP94, KTP93, Kit94]. Cet outil permet de générer les graphes correspondants aux applications le plus souvent parallélisées [Bou94]. Ces graphes sont valués ce qui permet de connaître le nombre de calculs que chaque noeud doit effectuer ainsi que le volume de données pour chaque communication. Cet outil permet de régler les paramètres afin de modéliser le plus fidèlement possible un ordinateur réel. Ici, les paramètres sont ajustés pour représenter les caractéristiques de l'ordinateur IBM SP-1 de l'IMAG, à 16 processeurs. On utilise le modèle de communication décrit au paragraphe 4.4.1. Pour ce jeu de tests, j'ai choisi de sélectionner les graphes suivants qui représentent un large éventail des programmes couramment utilisés. Chaque graphe est décliné en deux versions de taille différente (**m** pour taille moyenne et **l** pour large). La structure générale de ces graphes est présentée en annexe, à partir de page 97.

1. **divconq:** représente la structure d'un programme basé sur la méthode du diviser-pour-régner (division par 2 à chaque étape).

2. **itératif:** il s'agit de la structure d'un programme itératif. A chaque itération, il y a diffusion des informations vers les noeuds de l'étage suivant.

3. **bellford:** représente la structure de l'algorithme de Bellman-Ford qui cherche les plus courts chemins entre tous les noeuds d'un graphe orienté valué et un noeud destination sans successeur.

4. **diamond1:** représente la structure d'un calcul systolique générique.

5. **diamond2:** représente la structure d'un calcul systolique de multiplication de matrices.

6. **diamond3:** représente la structure d'un calcul systolique de multiplication de matrices de manière plus fine que diamond2.

7. **diamond4:** représente la structure d'un calcul systolique de la fermeture transitive d'une relation sur un ensemble d'éléments.

8. **fft:** il s'agit d'un graphe représentant une transformée de Fourier unidimensionnelle.

9. **gauss:** élimination de Gauss utilisée pour la résolution de systèmes linéaires.

10. **ms-gauss:** programme parallèle, qui est la concaténation d'un programme *maître-esclave* avec une élimination de Gauss. La phase maître-esclave peut être un traitement sur une matrice et la phase Gauss la résolution du système linéaire représenté par la matrice traitée précédemment.

11. **prolog:** struture d'un programme prolog.

12. **qcd:** il s'agit de la méthode du gradient conjugué pour systèmes linéaires.

Outre son côté réaliste, j'ai aussi choisi ce jeu de tests afin de pouvoir plus facilement faire des comparaisons entre les diverses méthodes.

D'autres graphes font aussi partie du jeu de tests. Ainsi, la famille des graphes appelée **ssc** sont des graphes qui représentent un contrôleur espace-temps utilisé par [KN85]. Les graphes **elbow** et **stanford** de Kasahira et Narita représentent le contrôleur de l'épaule et d'un bras d'un robot [KN84]. Ces graphes ont été fournis par F. Sandnes [SM96]. Ces graphes permettent de diversifier le jeu de tests et pour autoriser des comparaisons avec d'autres méthodes.

Les graphes qui représentent un programme parallèle (*1 à 12*) sont simulés sur un ordinateur IBM-SP1 à 16 processeurs. Les autres graphes sont aussi simulés sur une machine à 16 processeurs. Le nombre de tâches varie suivant les graphes de 90 à 1482 comme indiqué par le tableau 4.5, page 58.

4.6 L'algorithme FSG

L'algorithme présenté ici est un algorithme que j'ai obtenu après de multiples perfectionnements. Pour le problème posé, il est clair que l'obtention de résultats intéressants passe par l'utilisation de connaissances sur le problème. Pour cela, on peut utiliser, par exemple, la connaissance contenue dans les heuristiques, comme l'heuristique MISF ("Most Immediate Successor First", successeur le plus immédiat d'abord) présentée par la suite. Je l'ai nommée FSG pour "Full Search Genetic". Il s'agit d'un algorithme à représentation directe.

4.6.1 Les heuristiques MISF

Une heuristique construit généralement la solution pas-à-pas. Quand on construit un ordonnancement tâche par tâche, à chaque étape on doit choisir une tâche parmi celles ordonnançables et un processeur pour exécuter la tâche choisie.

Les heuristiques MISF reposent sur l'usage de règles et ordonnancent les tâches suivant leur date d'introduction.

Utiliser une heuristique consiste à faire ce choix suivant un ou plusieurs critères "*sensés*" au lieu d'avoir recours au hasard ou à un ordre à priori comme l'ordre des indices des tâches.

Ensemble de tâches ordonnancées T_o
$T_o \leftarrow \emptyset$
Ensemble de tâches non ordonnancées T_{no}
$T_{no} \leftarrow T_i,\ 0 \leq i \leq n$
Ensemble de tâches ordonnançables $T_{libre} \leftarrow \emptyset$
Ensemble de règles $R_i,\ 0 \leq i \leq m$
Pour $i = 1$ à n
  calculer T_{libre}
  pour $j = 1$ à m
    appliquer R_j sur T_{libre}
  choisir $T_{ch} \in T_{libre}$
  $T_o \leftarrow T_{ch}$
  choisir le processeur P suivant la règle R_P
  ordonnancer T_{ch} sur P au plus tôt
Fin Pour

Figure 4.3: Algorithme d'une heuristique.

La figure 4.3 présente la struture générale des heuristiques que nous utiliserons comme base dans les algorithmes à représentation directe pour le problème de l'ordonnancement. Ce type d'heuristique construit la solution pas-à-pas comme tout algorithme de liste.

MISF est l'abréviation de "Most Immediate Successor First". Cette heuristique repose sur un algorithme de liste. L'application d'une règle R_j sur un ensemble de tâches consiste à ne garder dans cet ensemble que les tâches

qui satisfont à un critère donné. La règle R_1 consiste à ne garder dans l'ensemble des tâches ordonnançables (ou libres) que les tâches qui ont la même date minimale d'introduction possible sur un ou plusieurs processeurs. Il existe d'autres règles afin d'essayer d'améliorer la solution trouvée. Par exemple, la règle R_2 proposée consiste à ne garder que les tâches de plus grande hauteur (celle dont le chemin minimal à un noeud puit est le plus long, en terme de durée de calcul). Enfin, R_3 ne garde que les tâches qui ont un nombre maximum de successeurs. Si après application des règles, l'ensemble des tâches ordonnançables n'est pas réduit à un élément, alors un choix aléatoire est pris. Ceci explique que deux exécutions de la même heuristique peuvent donner des résultats différents. Par la suite, nous appellerons heuristique-l[4] l'heuristique utilisant seulement R_1 et heuristique-r[5] en cas de l'utilisation successive de R_1,R_2 et R_3. Le choix du processeur se fait dans les deux cas suivant la règle R_P qui choisit aléatoirement un processeur parmi ceux qui minimisent la date d'introduction de la tâche choisie.

4.6.2 La représentation

Un chromosome est un ensemble de chaînes qui représente pour chaque processeur l'ordre d'exécution des tâches sur ce processeur.

J'ai choisi de conserver la même représentation que celle utilisée dans [HAR94]. En effet, cette représentation a l'avantage d'être proche de la solution. Ainsi, un chromosome nous permet de savoir pour chaque processeur quelles sont les tâches qu'il doit traiter et dans quel ordre. Il est donc facile de voir qu'à un ordonnancement correspond un et un seul chromosome possible. De plus, la structure de chaîne permet de représenter facilement et directement l'ordre des tâches sur chaque chromosome. Il suffit de regarder la figure 4.1 pour voir les analogies entre la représentation faite par le chromosome de la solution et par le tableau représentant l'ordonnancement.

Par contre, comme dans le cas de l'algorithme HAR, un chromosome ne code pas directement les moments de début d'exécution de chaque tâche, donc la durée totale d'exécution. Le contenu de la section 4.6.3 explique comment, à partir d'un chromosome, obtenir un ordonnancement et sa durée.

4.6.3 La santé

Cette fonction a pour but de déterminer, à partir d'un chromosome, la valeur de l'ordonnancement associé. Cela revient à calculer pour chaque tâche sa date d'introduction et à faire en sorte que toutes les contraintes soient respectées. Pour ce faire, on utilise un algorithme de liste où les règles de priorité choisissent à chaque pas le couple tâche-processeur voulu.

Au moins une solution optimale est représentable.

L'intérêt de la représentation et de la fonction de santé choisie est de permettre qu'au moins un ordonnancement optimal soit représentable [CFR96]. En effet, il est trivial de montrer que l'ordonnancement optimal peut être représenté : il suffit de mémoriser sur chaque processeur l'ordre des tâches exécutées sur celui-ci. Réciproquement, l'algorithme de liste appliqué sur ce chromosome fournira un ordonnancement optimal. En effet, si l'ordonnancement fourni n'était pas optimal, il existerait une tâche T_i dont le début

[4] l signifie large, en terme d'espace de recherche.
[5] r pour restreint.

d'exécution serait retardé par rapport à l'ordonnancement optimal fourni. Or, comme l'algorithme ordonnance chaque tâche aussitôt que possible, cela implique qu'une autre tâche t_j, prédécesseur de t_i soit aussi dans ce cas. De proche en proche, on en déduit qu'une tâche source est retardée d'où la contradiction.

Cependant, pour des raisons de performances, la durée d'un ordonnancement sera calculée directement lorsque les opérateurs de croisement et de mutation modifient un chromosome. En effet, cela peut être fait facilement à ce moment-là comme on le verra par la suite.

4.6.4 Le croisement

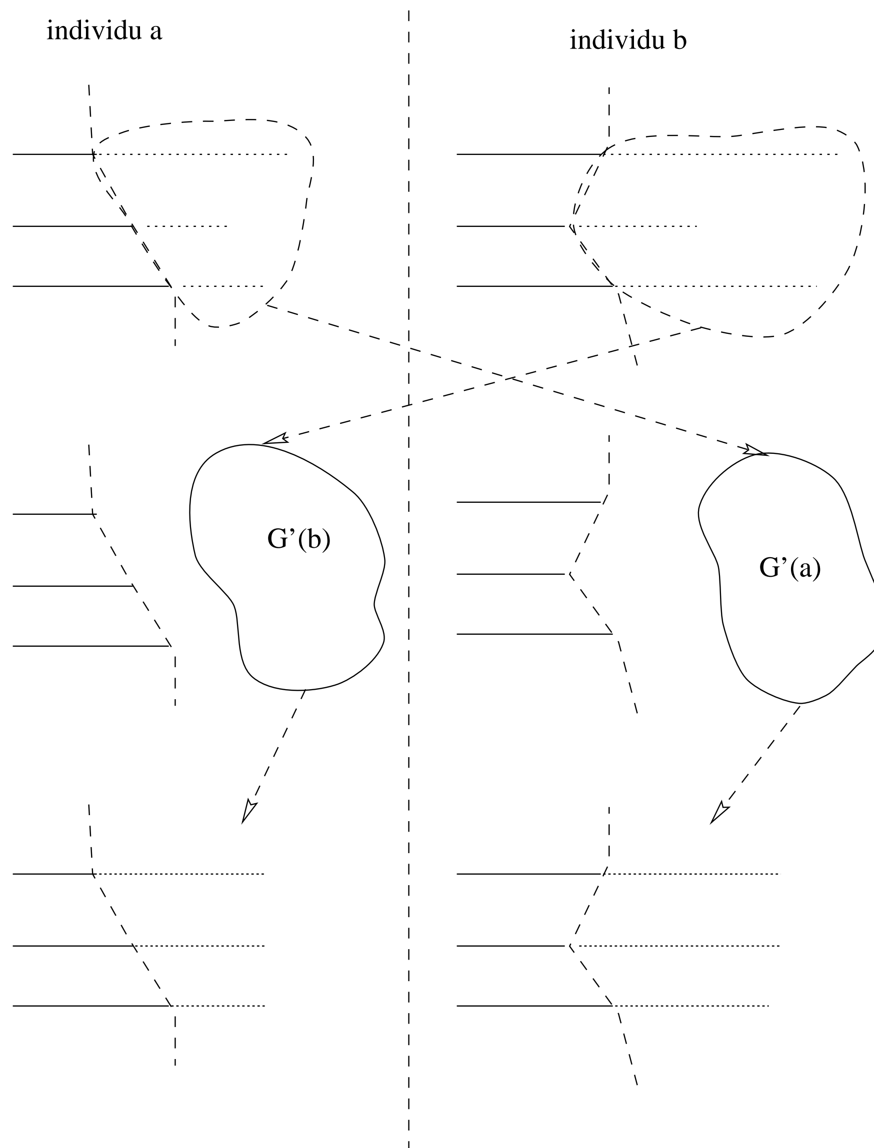


Figure 4.4: Le croisement de l'algorithme FSG

Cet opérateur doit générer un individu qui contient des caractérisques de chacun de ses deux parents. En outre, on veut utiliser les relativement bonnes performances obtenues par les heuristiques. La méthode généralement utilisée lors du croisement pour conserver une certaine ressemblance avec un des parents est de, lorsque celui-ci est, par exemple, représenté par une chaîne, garder intacte une portion de cette chaîne comme c'est le cas à la figure 3.3, page 24. Cependant, afin de pouvoir représenter au moins une solution optimale, la contrainte sur les hauteurs est supprimée. J'explique donc ici le nouvel opérateur de croisement que j'ai mis au point. Dans notre cas, nous avons bien une représentation sous forme de chaînes. Nous pouvons donc, sans problème, garder tels quels les débuts de chaque chaîne pour le nouvel individu généré.

La coupure

On partitionne l'ensemble des tâches en deux parties.

On veut donc couper chaque chromosome en deux parties. De plus, il faut que pour les deux chromosomes la partition soit la même afin de pouvoir facilement générer une solution valide. Pour arriver à cela, on construit la partition pas-à-pas. Soit T l'ensemble des tâches du graphe (ou programme) considéré. $T = T_1 + T_2 + T_3$ où T_1 représente les tâches qui seront dans la première partition, T_2 l'ensemble des tâches de la deuxième partition et T_3 les tâches non encore affectées à une des deux parties. Au départ, $T_1 = \emptyset$, $T_2 = \emptyset$ et $T_3 = T$. A chaque étape, une tâche est choisie aléatoirement parmi celles de T_3 de même que l'ensemble $T_j, j \in \{1, 2\}$ où cette tâche sera stockée.

Ensuite, une fermeture transitive est effectuée sur T_j en utilisant les relations de précédences induites par les arêtes de G ainsi que celles induites par les deux chromosomes concernés (Par exemple, si une tâche est dans T_1, toutes les tâches ordonnancées antérieurement sur le même processeur doivent être aussi dans T_1.). Ceci est itéré jusqu'à ce que $T_3 = \emptyset$.

La figure 4.5, page 45, permet d'illustrer un tel croisement, et notamment la phase de coupure. Supposons que l'on veuille croiser les deux chromosomes A et B représentés. Initialement, T et T_3 sont vides et T contient les 5 tâches. L'algorithme de coupure peut alors se dérouler de la manière suivante par exemple :

1. La tâche t_4 est tirée au sort et l'on choisit de la placer dans T_2. Comme t_5 lui est successeur, elle est aussi placée dans T_2. On met alors T_3 à jour et donc $T_3 = \{t_1, t_2, t_3\}$.

2. La tâche t_2 est choisie de même que l'ensemble T_1. La tâche t_1 est alors placée dans ce même ensemble. En effet, elle est prédecesseur de t_2 dans le graphe. De plus, elle est ordonnancée avant elle sur le même processeur dans le cas du chromosome B. (Une seule de ces deux conditions est suffisante). Après mise à jour, $T_3 = \{t_3\}$.

3. L'algorithme choisit de placer t_3 dans T_2. La phase de coupure est alors terminée car toutes les tâches sont soit dans T_1 soit dans T_2. ($T_3 = \emptyset$).

La phase de coupure proprement dite est alors achevée. Cette coupure est représentée par une ligne discontinue sur la figure 4.5.

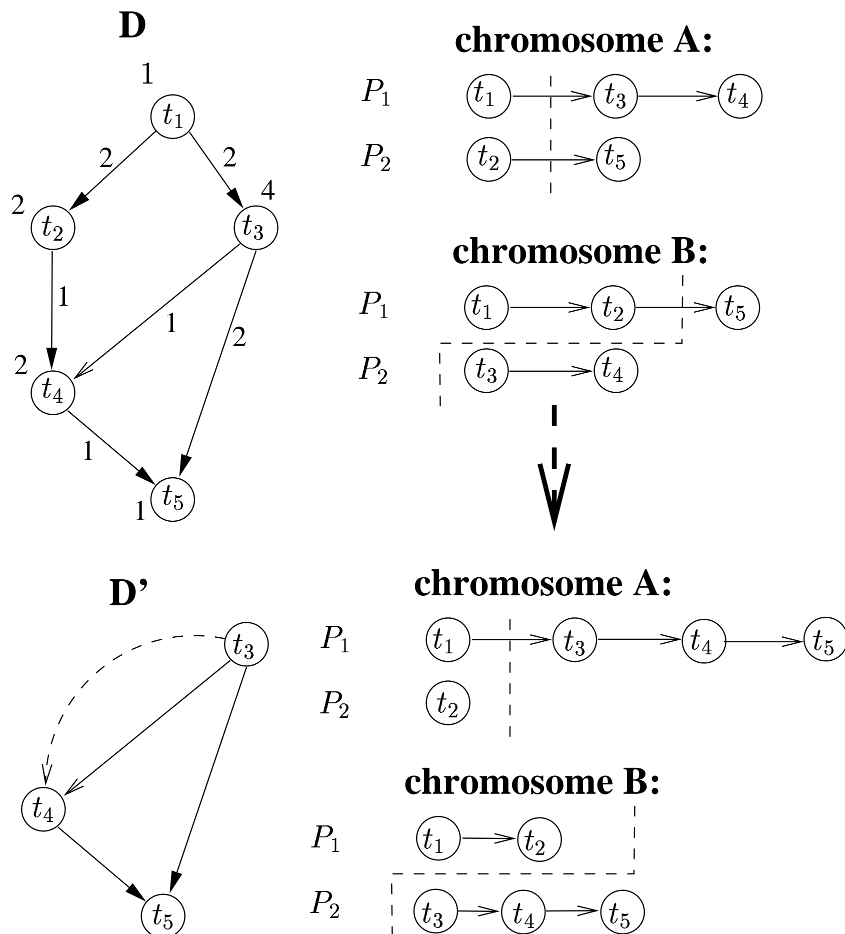


Figure 4.5: Exemple de croisement

Cette technique permet de garantir que pour chaque chaîne de chaque chromosome, il n'y a qu'un seul point de coupure séparant les tâches de T_1 de celles de T_2.

Cette méthode nécessite à chaque étape des calculs assez complexes (on effectue une fermeture transitive) mais elle permet de garantir qu'au moins une solution optimale soit représentable. A contrario, la méthode proposée par [HAR94] pour effectuer la coupure est beaucoup moins coûteuse puisqu'effectuer la coupure revient à partitionner les tâches suivant leur hauteur. La hauteur d'une tâche est définie comme étant le plus long chemin (en nombre de tâches) entre cette tâche et une tâche sans prédécesseur.

La recombinaison

On utilise une heuristique pour reconstruire la fin de chaque chromosome.

Une fois la partition construite, il est facile de générer un chromosome complet. On peut, par exemple, directement récupérer la partie droite de l'autre parent pour construire un individu-fils. Cependant, ceci ne donne pas des résultats intéressants car il y a alors inadéquation entre les deux parties, notamment au niveau de la coupure.

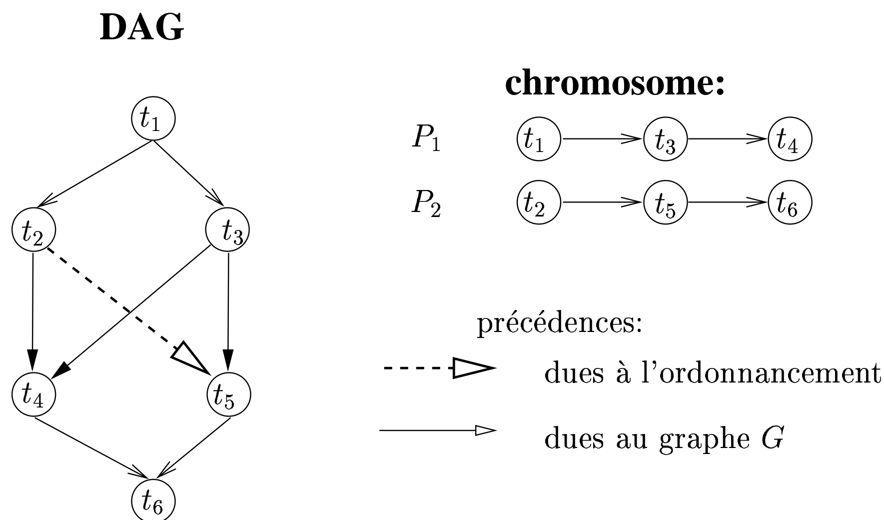


Figure 4.6: Exemple de graphe augmenté

Pour obtenir des résultats convaincants, on peut utiliser les connaissances via une heuristique. Il faut, dans ce cas, à la fois utiliser la puissance de l'heuristique mais aussi conserver une certaine mémoire en gardant des informations des parents. Notre algorithme reconstruit donc la partie droite grâce à une heuristique.

Dans notre cas, on construit le graphe **augmenté** D' de dépendances entre tâches correspondant à un ordonnancement donné. Ceci est expliqué par la figure 4.6. Dans cet exemple, le graphe contient deux sortes d'arêtes:

- Celles en trait continu qui sont les arêtes représentant les dépendances dues aux communications entre les tâches. Il s'agit des arêtes du graphe G qui modélise le programme.
- Celles en trait pointillé qui représentent le fait que deux tâches soient ordonnancées l'une après l'autre (de manière immédiate) sur le même processeur dans l'ordonnancement o.

Dans l'exemple de la figure 4.6, les arêtes du deuxième type ne sont pas représentées lorsque les dépendances qu'elles ajoutent sont déjà induites par les arêtes du premier type. D' contient toutes les dépendances dues aux communications mais aussi toutes les dépendances induites par l'ordonnancement de deux tâches sur le même processeur.

Une fois le graphe D' construit en se servant d'un parent, la partie droite de l'autre individu est reconstruite en utilisant une des deux heuristiques présentées précédemment sur le graphe D' comme expliqué par la figure 4.4, page 43. Comme la partie droite est reconstruite par une heuristique qui calcule pour chaque tâche placée sa date d'introduction au plus tôt et que la partie gauche est conservée intacte[6], on dispose facilement à la fin du croisement de la valeur de santé de l'individu.

[6]On conserve dans le chromosome de manière cachée les dates d'exécution de chaque tâche.

Si l'on reprend l'exemple illustré par la figure 4.5, on construit le graphe D'. Il est restreint aux tâches de T_2 (pour les autres, leur ordonnancement restera inchangé). Il contient les arêtes de D mais aussi une arête entre deux tâches chaque fois que ces deux tâches sont sur le même processeur dans un des deux chromosomes parents. (Par exemple t_3 et t_4 dans le chromosome A). A ce moment-là, on considère l'ordonnancement partiel obtenu en conservant les parties gauches de chaque chromosome et l'on va compléter chaque chromosome en utilisant une heuristique et en considérant comme graphe de dépendances D'. Deux solutions possibles pour les chromosomes A et B après croisement sont représentées sur la figure 4.5.

Ainsi, dans le cas du chromosome A, la partie inchangée représente un ordonnancement partiel où la tâche t_1 s'exécute sur le processeur P_1 au temps 0 et la tâche t_2 sur P_2 au temps 3. L'heuristique (par exemple l'heuristique-l décrite précédemment) va compléter cet ordonnancement. Ainsi, compte-tenu du graphe de dépendance D' utilisé, l'heuristique sera contrainte d'ordonnancer les tâches t_3, t_4 et t_5 dans cet ordre. Dans le cas du chromosome B, les deux tâches t_1 et t_2 seront ordonnancées sur le même processeur P_1.

Le croisement ainsi défini permet donc de conserver des caractéristiques de deux parents, mais aussi d'utiliser une heuristique. Le choix de la coupure se faisant aléatoirement, on peut ne conserver intacte qu'une toute petite partie gauche. On remarquera que l'heuristique s'applique d'autant plus souvent qu'on est à la fin de l'ordonnancement. Ceci peut paraître à première vue un défaut mais il faut tenir compte de deux facteurs: une heuristique est en général plus performante pour achever de construire une solution que pour commencer à la construire[7]. Ensuite, si l'on imagine l'arbre représentant toutes les solutions possibles, plus l'on est bas dans l'arbre (plus l'on est proche de la fin de l'ordonnancement), plus les choix possibles sont nombreux (l'arbre est plus large). Il est donc logique, à chaque application de l'heuristique pour construire le début de l'ordonnancement, d'appliquer plusieurs fois l'heuristique afin de construire la fin de l'ordonnancement.

L'heuristique est privilégiée pour déterminer la fin de l'ordonnancement.

4.6.5 La mutation

La mutation reconstruit une solution en utilisant une heuristique MISF.

La mutation est bâtie sur le même principe que le croisement. L'individu qui subit la mutation permet de générer le graphe *augmenté* G' correspondant à l'ordonnancement qu'il représente. L'individu est ensuite reconstruit intégralement par l'utilisation d'une heuristique sur le graphe G' construit comme montré par la figure 4.7, page 48.

4.6.6 Résultats

Le tableau 4.1 résume le temps d'exécution de la meilleure solution obtenue après 2 heures de calculs sur un PC équipé d'un processeur Pentium Pro à 200 MHz sous Solaris. L'algorithme FSG1 utilise l'heuristique-l pour le croisement et l'heuristique-r pour la mutation alors que FSG2 utilise l'heuristique-l pour les deux opérateurs. La taille de la population est, dans les deux cas, de 100 individus. Pour chaque graphe, on calcul l'écart en pourcents entre la

[7] Elle est même optimale pour ordonnancer la dernière tâche.

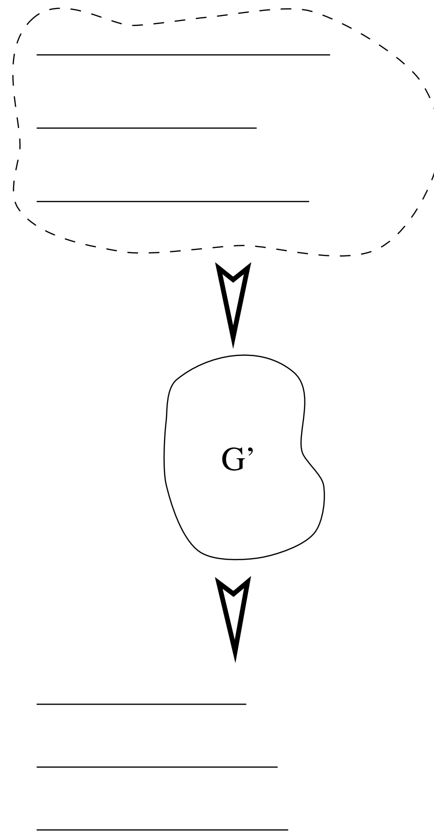


Figure 4.7: La mutation de l'algorithme FSG

solution trouvée et le meilleur résultat connu. La ligne appelée "écart moyen" représente la valeur moyenne de cet écart et celle nommée "pire écart" contient la valeur du plus mauvais résultat de la méthode concernée. Les résultats en caractères gras indiquent qu'il s'agit du meilleur résultat obtenu parmi l'ensemble des algorithmes testés présentés dans ce chapitre. La dernière colonne du tableau 4.4, page 57 résume les résultats obtenus par l'heuristique DSC qui s'avère plus performante, notamment pour les graphes de type ssc. Ce point sera discuté par la suite.

Les algorithmes génétiques sont aussi efficaces que les heuristiques qu'ils utilisent.

Le tableau 4.6 indique le temps de calcul moyen pour chaque graphe des heuristiques utilisées. D'une manière générale, l'heuristique-l est plus lente que la version-r. Ceci s'explique par le fait qu'à chaque étape cette dernière doit appliquer en général plus de règles avant de choisir la tâche à ordonnancer.

La première remarque est que les quatre méthodes obtiennent des résultats très voisins, meilleurs que ceux obtenus suivant la méthode proposée par Hou, Ansari et Ren [RSM98]. Il faut préciser que cette méthode n'est pas adaptée à ce type de problème car à l'origine elle ne tient pas compte du coût des communication. Les algorithmes génétiques obtiennent des résultats presque aussi bons que les heuristiques sur lesquels ils reposent comme le montre l'écart moyen calculé. Cependant, compte-tenu du coût important de la coupure nécessaire, l'heuristique est exécuté un nombre inférieur de fois dans les algo-

graphe		heuristique-r	heuristique-l	FSG-1	FSG-2
bellford	m	**71 936 050**	77 830 400	76 900 150	78 994 250
	l	194 182 200	209 308 100	202 792 200	211 790 150
diamond1	m	**131 940 750**	**131 940 750**	**131 940 750**	134 422 800
	l	286 453 550	292 815 100	297 157 650	305 071 000
diamond2	m	128 769 500	145 276 950	156 849 600	167 058 750
	l	254 504 850	260 949 900	283 927 300	292 618 700
diamond3	m	180 576 400	**176 982 100**	179 412 550	180 730 750
	l	229 754 350	**228 590 500**	230 296 650	230 503 050
diamond4	m	**132 875 550**	149 187 800	148 643 150	154 564 700
	l	**164 264 000**	175 181 950	176 435 400	179 486 950
divconq	m	108 044 840	**97 307 880**	98 7105 40	98 782 780
	l	290 972 340	**169 043 350**	170 757 740	169 936 980
fft	m	**29 888 250**	30 650 550	30 650 550	30 650 550
	l	108 813 250	109 911 900	105 180 950	104 418 650
gauss	m	237 709 950	250 378 050	243 602 850	245 784 800
	l	325 456 750	359 896 750	326 144 800	341 412 200
itératif	m	22 824 200	21 8087 00	21 808 700	22 099 550
	l	51 524 200	52 241 700	52 952 050	53 540 900
ms-gauss	m	4 633 100 500	4 633 674 500	4 605 112 750	4 629 991 300
	l	2 936 022 450	2 896 605 150	2 941 542 700	2 960 308 600
prolog	m	63 292 900	**60 499 350**	**60 499 350**	**60 499 350**
	l	261 246 450	**258 529 350**	**258 529 350**	**258 529 350**
qcd	m	1 176 047 050	1 176 047 050	1 176 047 050	1 176 047 050
	l	3 928 300 600	3 928 300 600	3 831 179 300	3 730 347 050
elbow		**6 630**	**6 630**	**6 630**	**6 630**
stanford		668	656	647	661
ssc	5	110	105	109	109
	6	126	125	127	127
	7	139	141	138	141
	8	164	160	170	168
	9	173	171	177	179
Ecart moyen en %		27.238088	23.793542	20.604982	21.872149
Pire écart en %		168.5382	118.0870	97.6744	95.3488

Tableau 4.1: Résultats en 2 heures, algorithmes à représentation directe.

rithmes génétiques de type FSG que dans les méthodes purement heuristiques. A titre d'exemple, pour le graphe prolog-l, l'heuristique-l s'exécute 1315 fois en deux heures tandis que l'algorithme FSG-1 ne l'exécute que 902 fois. Par contre, la ligne "pire écart" montre qu'ils sont un peu moins sensibles à des cas particuliers ou pathologiques.

Cette ligne montre surtout la différence entre les deux heuristiques proposées. Le fait d'utiliser des critères très sélectifs limite l'utilisation de l'aléatoire ce qui peut favoriser l'existence de cas pathologiques quand les critères retenus ne sont pas valables sur un graphe particulier. L'heuristique-r est moins performante aussi à cause de sa lenteur relative qui réduit le nombre de solutions explorées par l'utilisation de l'aléatoire.

4.7 L'algorithme génétique à représentation indirecte

Comme le montrent les résultats précédents, l'utilisation conjointe d'un algorithme génétique et d'une heuristique donne des résultats intéressants. Cependant, une grande partie du temps de calcul est dépensée lors du croisement afin d'effectuer une coupure satisfaisant certains critères. En outre, dans notre cas, le résultat est essentiellement construit par l'heuristique car celle-ci intervient à la fois dans le croisement et dans la mutation.

Il existe un autre type d'algorithme génétique hybride, présenté à la section 3.5.1, page 28, qui utilise la représentation indirecte.

Comme cet algorithme a pour but de fournir un ordre total sur les tâches, un chromosome est une liste ordonnée de toutes les tâches.

4.7.1 Travaux déjà effectués

Des algorithmes génétiques à représentation indirecte ont été développés notamment dans le cas de réseaux non homogènes [DAS95, AD96, SM96]. Certains de ces travaux sont expliqués dans la suite.

4.7.2 L'algorithme d'Ahmad

Ahmad, Dhodhi et Storer ont développé deux algorithmes génétiques pour le problème de l'ordonnancement statique de tâches sur machine parallèle. Le premier [DAS95] essaie de déterminer le nombre optimal de processeurs nécessaires dans un système hétérogène. Le chromosome est dans ce cas divisé en deux parties : la première indique la disponibilité de chaque processeur par un tableau de booléens qui indique si le processeur est utilisable ou non dans la solution (l'hétérogénéité provient du fait que chaque processeur peut être différent). La deuxième partie du chromosome est une liste de priorités. Ainsi, en décodant le chromosome, la fonction de santé utilise cette liste de priorités pour construire l'ordonnancement sur les processeurs disponibles indiqués par la première partie du chromosome. L'algorithme utilise le croisement à un point qui sera expliqué à la section 4.8.1. Les opérateurs de croisement et de mutation sont appliqués sur chacune des deux parties du croisement. Cependant, cet algorithme ne nous intéresse guère car dans notre cas, le nombre de processeurs disponibles est une *donnée* du problème.

L'algorithme d'Ahmad et Dhodhi utilise une heuristique de liste.

Ahmad et Dhodhi ont aussi mis au point un algorithme où le nombre de processeurs est fixé [AD96]. Dans ce cas, le chromosome ne code que la liste de priorités entre les tâches. Un chromosome est donc une liste ordonnée (une chaîne) de toutes les tâches du problème. Le coeur de l'algorithme génétique reste similaire à celui précédemment décrit. L'ordonnancement est déterminé en utilisant une heuristique d'ordonnancement au plus tôt. L'ordonnancement est construit de la façon suivante : l'ensemble des tâches *libres* est construit puis **toutes** les tâches de cet ensemble sont ordonnancées dans l'ordre fixé par la liste de priorité. L'ensemble des tâches libres est ensuite remis à jour. Ce processus est répété jusqu'à ce que toutes les tâches soient ordonnancées. La version décrite dans [AD96] ne tient pas compte du temps de communications,

comme celle de [HAR94]. Les auteurs signalent qu'ils obtiennent de meilleurs résultats que Hou, Ansari et Ren. Cependant, la version que nous avons testée tient compte du temps de communications entre deux processeurs distants. Elle a été mise au point par Sandnes, de l'université de Reading [SM96] et est expliquée ci-après.

4.7.3 L'algorithme de Frode

Sandnes a étendu l'algorithme de Ahmad et Dhodhi pour prendre en compte les communications comme expliqué précédemment. En outre, il a aussi modifié les informations codées par le chromosome pour pouvoir modéliser des réseaux de communication partiellement connectés, c'est-à-dire qu'un message entre deux noeuds doit être relayé par des noeuds intermédiaires.

Ajout des coûts de communication

Ceci se fait en modifiant légèrement l'heuristique utilisée par Ahmad et Dhodhi. L'heuristique alloue la tâche à ordonnancer sur le processeur qui permet à la tâche d'être exécutée le plus tôt possible. Ceci est déterminé en calculant pour la tâche sa date de début d'exécution sur chaque processeur et en ne gardant que la meilleure. Pour connaître cette date, on calcule pour chaque tâche prédécesseur sa date de fin de calcul additionnée du temps de communication. Ce temps de communication prend en compte le nombre de noeuds intermédiaires que doit utiliser le message pour parvenir au destinataire.

La topologie du réseau est représentée sous forme d'une matrice *diagonale* de taille $m * m$, où m est le nombre de processeurs disponibles. Un zéro signifie que les deux processeurs correspondants ne sont pas connectés directement et que, par conséquent, les messages entre eux doivent transiter par des noeuds intermédiaires. L'algorithme que l'on nommera Ahmad par la suite se référera à l'algorithme d'Ahmad décrit précédemment étendu par cette méthode pour tenir compte du temps de communication. Le routage des messages n'est pas présenté ici car nous utiliserons cet algorithme sur des réseaux entièrement connectés.

Agrandissement de l'espace de recherche

L'allocation implicite du processeur utilisée par Ahmad n'est pas une stratégie optimale en ce sens que, pour un même ordre sur les tâches, on peut obtenir un ordonnancement meilleur en utilisant une autre stratégie. Dans le cas d'Ahmad, on choisit le processeur qui permet à la tâche de s'exécuter au plus tôt. Mais ce critère ne permet pas de choisir entre plusieurs processeurs qui auraient la même date de début d'exécution pour la tâche considérée. Dans ce cas, on choisit généralement le processeur de plus petit indice. Ceci limite l'espace de recherche.

La figure 4.8 montre un exemple de ce qui peut se produire. Cela peut se produire quand on ordonnance la première tâche d'un graphe. Cette figure montre un exemple où un graphe à 4 noeuds doit être ordonnancé sur une topologie en étoile à trois processeurs. Suivant qu'on ordonnance la première tâche sur le processeur 1 ou non, on obtient, en suivant la même heuristique

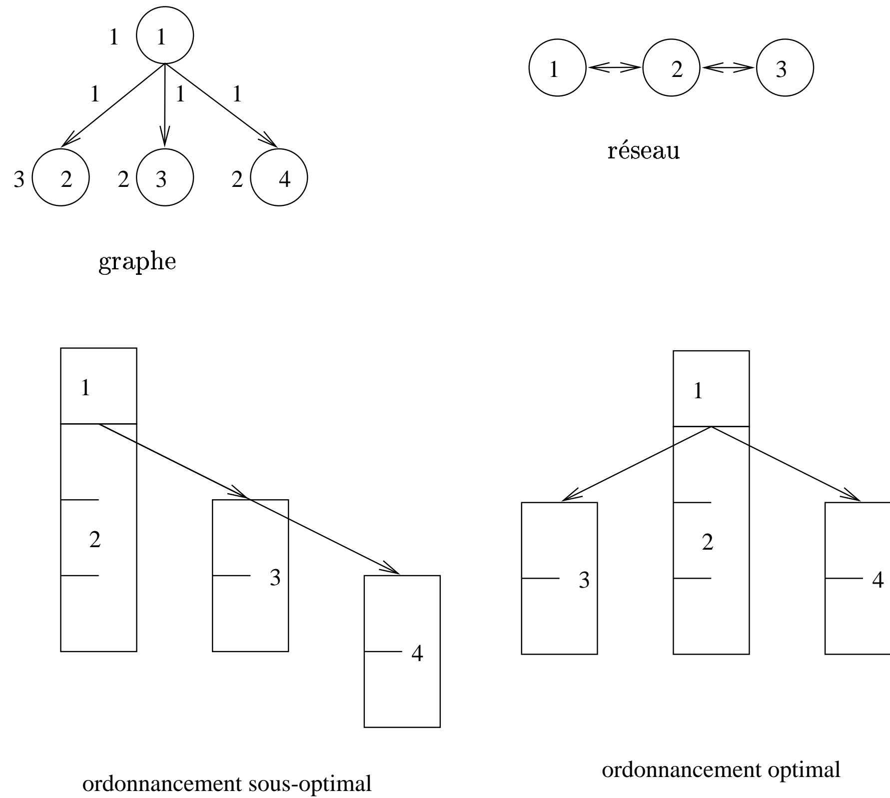


Figure 4.8: Importance du choix du processeur.

où l'on place les tâches suivant l'ordre de leur indice, une durée d'exécution de 4 ou 5 unités de temps.

Cette amélioration de l'algorithme d'Ahmad concerne plus particulièrement les topologies non totalement interconnectées.

L'algorithme de Frode code, en plus, le choix du processeur de manière implicite.

Cet inconvénient est résolu en modifiant l'allocation implicite des processeurs. On ajoute au chromosome une partie nommée "*choix du processeur*". Il s'agit d'une chaîne d'entiers de taille N (N est le nombre total de tâches). Chaque entier cp de cette chaîne est tel que $1 \leq x \leq m$ (m est le nombre de processeurs). Cette chaîne est utilisée quand il y a un choix à faire entre deux (ou plus) processeurs qui permettent d'obtenir la même date minimale d'exécution de la tâche que l'on est en train d'ordonnancer. Cette chaîne est manipulée par l'algorithme génétique de manière classique. Par exemple, si la tâche T_i peut être ordonnancée au plus tôt sur n'importe quel processeur de l'ensemble $e = \{p_1, \ldots, p_s\}$, alors l'algorithme de Frode va choisir ce processeur en fonction du chromosome contrairement à celui d'Ahmad qui effectue ce choix aléatoirement. En effet, c'est le processeur $p_{cp_i modulo(s)}$ qui sera choisi. cp_i est l'entier de la chaîne "*choix du processeur*" associé à la tâche T_i. Ainsi, on ajoute au chromosome une chaîne de même longueur où chaque élément est un entier positif. Cette chaîne subit les opérateurs de croisement et de mutation de manière classique. Si au cours de l'ordonnancement une tâche peut être ordonnancée sur deux processeurs (P_2 et P_5 par exemple, sur une machine à 8 processeurs) avec la même date minimale d'introduction, alors

l'heuristique va regarder dans cette chaîne la valeur associée à cette tâche (7 par exemple). Pour ce faire, elle renumérote les deux processeurs (P_2 devient P'_0 et P_5 P'_1). Elle choisie alors le processeur $P'_{7modulo(2)}$ car on a le choix entre deux processeurs. Le choix du processeur est donc fait de manière déterministe par l'heuristique. Nous appellerons "Frode" cet algorithme mais nous ne l'utiliserons qu'uniquement pour des réseaux totalement connectés.

4.8 L'algorithme IS

L'algorithme IS (pour "Indirect Search") est un algorithme génétique hybride à représentation indirecte que j'ai conçu. Contrairement aux algorithmes précédents, il n'est pas destiné à une topologie hétérogène.

L'algorithme IS utilise la représentation utilisée par Ahmad, c'est-à-dire une chaîne codant un ordre sur les tâches.

Il utilise la même représentation que celle utilisée par Ahmad et la majorité des algorithmes à représentation indirecte ; c'est-à-dire qu'un chromosome est la liste ordonnée des tâches à ordonnancer. Il est ici décliné en trois versions, suivant le croisement utilisé.

4.8.1 Le croisement à un point

Il s'agit d'un des croisements les plus simples lorsque les chromosomes sont des chaînes. Les chaînes des deux parents sont coupées au même endroit. Les deux bouts gauches sont copiés tels quels pour les enfants. Dans la version la plus simple possible, les deux bouts droits sont échangés afin que chaque enfant ait des caractéristiques de chacun de ses parents. Mais ici, cela n'est pas possible car on doit garantir après le croisement que, dans la chaîne, chaque tâche soit présente une et une seule fois. Pour respecter cette contrainte, au lieu d'échanger les deux bouts droits, chaque bout droit est reconstruit. Lors de la reconstruction, les tâches manquantes y sont placées en respectant l'ordre fixé par l'autre parent.

4.8.2 Le croisement à deux points

Il s'agit d'un croisement inspiré de celui de la figure 3.3, page 24. Chaque chromosome est découpé en 5 parties dont deux sont reconstruites de la même manière que pour le croisement à un point.

4.8.3 Le croisement de Ferland

Le croisement utilisé ici est le croisement de Ferland [FF94]. Ce type de croisement a montré son efficacité pour certains problèmes. Il peut être vu comme une généralisation du croisement à n points. Il est expliqué par la figure 4.9. On tire de manière aléatoire quels éléments seront conservés à la même position. Le chromosome est ensuite complété de la même manière que pour les deux croisements précédents.

Le croisement de Ferland est une généralisation du croisement à n points.

4.8.4 La mutation

Une mutation possible serait d'intervertir l'ordre de deux tâches. Cependant, compte tenu du nombre même de tâches, il faut que cet opérateur

parents

parent **a**	4	8	2	1	5	7	6	3
parent **b**	2	3	8	7	6	4	1	5

choix

choix	1	0	1	1	0	1	0	1

1ère étape

fils **a**	4	-	2	1	-	7	-	3
fils **b**	2	-	8	7	-	4	-	5

Résultats

fils **a**	4	8	2	1	6	7	5	3
fils **b**	2	1	8	7	6	4	3	5

Figure 4.9: Exemple du croisement de Ferland

introduise un changement plus important. Ainsi, la mutation permute deux séquences de tâches de même longueur, longueur choisie aléatoirement entre une tâche et $\frac{1}{25}$ du nombre total de tâches. Le choix de cette longueur me permet d'effectuer une modification significative de la solution tout en gardant nombre de caractéristiques du chromosome mutant.

4.8.5 Le sélection

La sélection est la même que celle utilisée lors des précédents algorithmes génétiques; c'est-à-dire qu'on utilise une roulette biaisée.

4.8.6 L'heuristique HL

HL est un algorithme de liste glouton.

Il s'agit d'une heuristique gloutonne assez simple basée sur un algorithme de liste d'où le nom: HL. A chaque étape, on sélectionne parmi les tâches ordonnançables celle qui est la plus prioritaire. Celle-ci est ordonnancée sur le processeur de plus faible indice qui minimise sa date de début d'exécution. La différence essentielle avec l'heuristique employée par Ahmad est que cette dernière ne met à jour l'ensemble des tâches ordonnançables que lorsque celui-ci est devenu vide tandis que HL met à jour cet ensemble à chaque pas.

graphe		Ahmad	Frode	HL
bellford	m	76 512 200	78 994 250	81 476 300
	l	228 000 650	223 578 850	238 859 100
diamond1	m	134422800	134422800	134 422 800
	l	281 255 850	289 169 200	289 169 200
diamond2	m	231 852 500	238 133 400	149 290 350
	l	424 524 050	422 459 250	274 220 500
diamond3	m	194 562 400	212 661 500	196 423 800
	l	248 498 500	282 471 300	261 193 950
diamond4	m	153 293 850	159 215 400	153 475 400
	l	185 391 100	200 830 400	184 846 450
divconq	m	107 072 860	107 114 140	109 910 000
	l	195 686 370	213 520 410	197 503 140
fft	m	**29 888 250**	30 560 950	31 659 600
	l	118 611 500	126 212 450	121 145 150
gauss	m	240 809 300	271 997 050	263 845 750
	l	353 580 250	367 395 100	378 722 200
itératif	m	**16 733 350**	16 793 350	21 808 700
	l	**47 936 700**	48 363 350	53 378 700
ms-gauss	m	**4 598 559 550**	**4 598 559 550**	4 605 112 750
	l	2 570 505 150	2 795 218 300	2 560 276 750
prolog	m	63 445 800	63 522 250	63 522 250
	l	299 015 150	311 565 450	302 190 950
qcd	m	**1 173 100 600**	**1 173 100 600**	1 176 047 050
	l	**2 038 576 050**	**2 038 576 050**	2 041 522 500
elbow		**6 630**	**6 630**	**6 630**
stanford		**627**	639	653
ssc	5	109	110	105
	6	125	127	125
	7	144	142	141
	8	173	172	160
	9	179	180	171
Ecart moyen en %		22.942048	26.285710	20.792307
Pire écart en %		101.1628	100.0000	**86.0465**

Tableau 4.2: Résultats en 2 heures, représentation indirecte.

4.9 Résultats

Les tableaux 4.1, 4.2 et 4.3 montrent les résultats obtenus par les différents algorithmes, toujours après deux heures d'exécution sur un processeur Pentium Pro cadencé à 200 Mhz. Tout d'abord, le meilleur résultat obtenu par l'algorithme d'Ahmad par rapport à celui de Frode s'explique par le fait que notre jeu de tests soit composé uniquement de graphes qui s'appliquent sur des machines homogènes. Ceci désavantage l'algorithme de Frode qui cherche à explorer un espace plus important pour obtenir de bons résultats sur machines non homogènes. Mon algorithme IS a de meilleurs résultats que l'heuristique HL qu'il utilise ce qui montre l'intérêt des algorithmes génétiques hybrides. Les trois versions de l'algorithme IS sont proches mais la version utilisant le croisement de Ferland est celle qui donne les meilleurs résultats, ce qui con-

graphe		IS-Ferland	IS-1pt	IS-2pt
bellford	m	74 030 150	76 124 250	74 030 150
	l	**193 794 250**	199 146 300	198 758 350
diamond1	m	136 904 850	**131 940 750**	134 422 800
	l	280 867 900	**276 758 950**	280 867 900
diamond2	m	**127 224 450**	128 659 450	136 286 450
	l	**218 954 400**	223 543 500	223 108 900
diamond3	m	180 549 200	181 867 850	180 937 150
	l	237 897 600	240 276 450	235 803 500
diamond4	m	143 975 050	144 156 600	141 831 250
	l	171 767 300	173 020 750	171 041 100
divconq	m	98 892 090	101 749 870	101 972 700
	l	176 787 740	181 430 020	182 486 460
fft	m	**29 888 250**	30 986 900	30 224 600
	l	111 346 900	113 207 850	113 454 600
gauss	m	229 252 850	234 763 500	231 587 700
	l	324 115 750	327 597 350	339 153 800
itératif	m	21 091 200	20 671 700	21 091 200
	l	51 226 200	51 226 200	51 226 200
ms-gauss	m	4 605 112 750	4 605 112 750	4 605 112 750
	l	**2 559 388 750**	**2 559 388 750**	**2 559 388 750**
prolog	m	**60 499 350**	60 652 250	60 575 800
	l	275 825 800	287 535 150	284 588 700
qcd	m	1 176 047 050	1 176 047 050	1 176 047 050
	l	2 041 522 500	2 041 522 500	2 041 522 500
elbow		**6 630**	**6 630**	**6 630**
stanford		629	629	629
ssc	5	100	99	97
	6	124	128	122
	7	135	137	130
	8	168	161	164
	9	172	178	172
Ecart moyen en %		**13.820406**	14.719142	14.097523
Pire écart en %		95.3488	93.4783	90.6977

Tableau 4.3: Résultats en 2 heures, algorithmes IS à représentation indirecte.

firme les résultats obtenus dans [FF94]. Ces très bons résultats obtenus par les algorithmes de type IS sont dus en partie au fait que le surcoût en calcul de la partie génétique est très faible.

L'algorithme IS-Ferland est le plus performant.

Le tableau 4.3 permet de comparer trois croisements différents par le nombre de points de coupure. On observe que plus le nombre de points de coupure est élevé plus le résultat est bon. Le croisement de Ferland est en effet une "généralisation" du croisement à n points, pour n important face à la taille du chromosome.

Les figures 8.9 à 8.24 présentées en annexe représentent l'évolution du meilleur résultat trouvé au cours de deux heures d'exécution pour les quatre algorithmes les plus représentatifs. D'une manière générale, on observe que l'algorithme HL est en général le moins performant et celui IS-Ferland le plus efficace. On observe que pour deux graphes, les algorithmes ne sont pas capables

graphe		heuristique-r	heuristique-l	HL	DSC
bellford	m	87 992 200	101 566 300	109 942 700	80 158 100
	l	229 786 050	274 930 150	263 913 200	197 828 100
diamond1	m	165 371 250	157 925 100	159 243 300	**131 940 750**
	l	340 749 950	344 783 800	332 990 950	360 143 300
diamond2	m	162 137 600	190 484 400	202 491 650	138 451 400
	l	261 840 250	288 410 100	377 203 350	238 374 150
diamond3	m	213 852 550	193 374 600	227 011 050	219 153 450
	l	264 348 700	261 866 650	284 747 400	261 375 950
diamond4	m	151 150 050	160 468 850	164 592 300	168 586 400
	l	176 634 350	184 336 600	201 703 350	235 123 650
divconq	m	188 555 430	102 072 640	122 906 180	110 599 130
	l	384 803 240	309 624 920	246 097 970	180 290 200
fft	m	35 964 600	34 193 250	34 955 550	37 063 250
	l	157 513 650	142 490 950	136 257 450	**102 647 300**
gauss	m	282 559 650	303 643 500	348 875 450	**226 882 900**
	l	374 781 900	466 469 000	494 210 250	**307 395 800**
itératif	m	25 822 850	25 694 200	23 243 700	23 961 200
	l	53 676 700	56 830 400	58 401 200	65 576 200
ms-gauss	m	4 999 774 200	5 036 797 200	4 605 112 750	6 559 159 600
	l	2 988 376 550	3 127 279 250	2 570 104 000	5 073 436 150
prolog	m	63 292 900	66 239 350	75 002 250	69 109 350
	l	261 322 900	261 475 800	357 409 000	272 879 350
qcd	m	1 531 468 350	1 531 468 350	1 176 047 050	1 176 047 050
	l	4 556 754 150	4 556 754 150	2 041 522 500	2 041 522 500
elbow		**6 630**	6 838	6 786	6656
stanford		688	711	700	679
ssc	5	129	132	142	**74**
	6	152	138	148	**77**
	7	166	152	158	**82**
	8	181	173	206	**86**
	9	201	186	213	**89**
Ecart moyen en %		51.104138	46.482512	41.236805	14.842689
Pire écart en %		209.3516	156.7524	139.5349	98.2284

Tableau 4.4: Heuristiques : résultats en une itération

d'améliorer la solution de manière significative au cours de leur exécution (graphes qcd2-b-m et elbow, figures 8.20(a) 8.21(a)). Pour ces deux graphes, on pourrait supposer que leur structure implique que les heuristiques utilisées conduisent toujours à un résultat similaire.

Concernant les graphes qcd2-b-l, ms-gauss-m et ms-gauss-l (figures 8.20(b), 8.18(a) et 8.18(b)), l'algorithme IS-Ferland et l'heuristique HL utilisé par ce dernier ne sont pas capables d'améliorer le résultat au cours du temps. Cela semble provenir de la structure même du graphe comme expliqué précédemment qui conduit inévitablement l'heuristique à trouver une solution quasi-optimale. C'est notamment envisageable quand on regarde l'accélération obtenue par le graphe qcd2-b-l, qui est proche du nombre de processeur de la machine parallèle.

Concernant l'évolution au cours du temps, les comportements sont beau-

graphe		tâches	séquentiel	résultat	accélération
bellford	m	354	1 010 814 000	71 936 050	14,05
	l	992	2 841 874 000	193 794 250	14,66
diamond1	m	258	740 460 000	131 940 750	5,61
	l	1026	2 944 620 000	276 758 950	10,40
diamond2	m	486	697 410 000	127 224 450	5,48
	l	1227	1 760 745 000	218 954 400	8,04
diamond3	m	731	2 097 970 000	176 982 100	11,85
	l	1002	2 875 740 000	228 590 500	12,58
diamond4	m	731	1 048 985 000	132 875 550	7,89
	l	1002	1 437 870 000	164 264 000	8,75
divconq	m	382	1 096 340 000	97 307 880	11,27
	l	766	2 198 420 000	169 043 350	13,01
fft	m	194	278 390 000	29 888 250	9,31
	l	1026	1 472 310 000	102 647 300	14,34
gauss	m	782	2 244 340 000	226 882 900	9,89
	l	1227	3 521 490 000	307 395 800	11,46
itératif	m	262	187 985 000	16 733 350	11,23
	l	938	673 015 000	47 936 700	14,04
ms-gauss	m	768	26 864 635 000	4 598 559 550	5,84
	l	1482	26 596 003 000	2 559 388 750	10,39
prolog	m	214	614 180 000	60 499 350	10,15
	l	1313	3 768 310 000	258 529 350	14,58
qcd	m	326	9 356 200 000	1 173 100 600	7,98
	l	1026	29 446 200 000	2 038 576 050	14,44
elbow		103	23420	6630	3,38
stanford		90	2484	627	3,96
ssc	5	200	200	74	2,70
	6	288	288	77	2,59
	7	392	392	82	4,78
	8	512	512	86	5,95
	9	648	648	89	7,28

Tableau 4.5: Accélérations obtenues

coup plus liés au graphe qu'à l'algorithme utilisé.

On remarque ici que l'approche indirecte est plus performante que l'approche directe car les méthodes de type IS donnent de meilleures solutions que celle utilisant l'algorithme FSG après deux heures de calcul. Cela est dû à la nature du problème et notamment au coût important du croisement dans le cas de la représentation directe qui doit effectuer à chaque fois une fermeture transitive du graphe augmenté G'[8]. La figure 8.25 et le tableau 4.6 montrent le temps d'exécution des trois heuristiques suivant le nombre de tâches du graphe à ordonnancer. Les heuristiques utilisées dans le cas de la représentation directe sont plus lentes car, à chaque étape, on doit, pour choisir la tâche à ordonnancer, déterminer l'ensemble des tâches ordonnançables et déterminer leur date d'introduction au plus tôt.

Le tableau 4.5 montre pour chaque graphe son nombre de tâches, la somme des durées des tâches ainsi que le meilleur ordonnancement obtenu et l'accé-

[8] Même si la fermeture transitive de G est précalculée une seule fois car G' contient G.

graphe		HL	heuristique-l	heuristique-r
bellford	m	35,4	799	820
	l	180	6 447	6 709
diamond1	m	9,78	65	63,5
	l	97,5	737	756
diamond2	m	33	223,5	269,5
	l	151,5	1 139	1495
diamond3	m	66,8	877	839
	l	103	1351	1 492,5
diamond4	m	61,6	580	600,3
	l	108,8	1 603,5	1 098
divconq	m	19,6	458,2	528
	l	60,3	1 704,4	2 013,5
fft	m	6,95	123,7	126,8
	l	109,1	1 995	2 322,5
gauss	m	59,9	620,2	732,5
	l	130,5	1 376,8	1 497,5
itératif	m	8,48	122,7	118,2
	l	73,3	789,6	879,2
ms-gauss	m	65,6	380,1	434,4
	l	198	1 674	1 998,3
prolog	m	11,8	191,8	254,3
	l	428,5	5 849,7	9 973,7
qcd	m	40,6	193,3	191,8
	l	285,8	2 169	2 142,2
elbow		2,6	17,9	19,2
stanford		2,2	14,7	17,4
ssc	5	5,4	105,3	114,9
	6	11,5	225,8	246,7
	7	17,2	390	430,4
	8	27	699,1	761,3
	9	39,4	1 054,8	1 146,1

Tableau 4.6: Temps moyen d'exécution des heuristiques en millisecondes

lération correspondante. Les résultats obtenus après une itération de chaque heuristique sont donnés par le tableau 4.4. Cela permet de montrer l'intéret d'effectuer des choix aléatoires au coeur de l'heuristique.

On remarque que l'heuristique DSC s'avère particulièrement efficace sur les graphes de types ssc comme celui représenté par la figure 8.8, page 101. Ceci peut s'expliquer par plusieurs faits. Tout d'abord, les temps de communications sont importants face aux durées d'exécution des tâches. En effet, si une tâche a besoin d'une unité de temps pour s'exécuter, une communication coûte 26 unités de temps alors que pour les autres graphes testés, ces deux coûts sont en général plutôt voisins. Si on regarde sa structure de près, on remarque que beaucoups de tâches ont deux prédécessurs. Ainsi, pour obtenir une parallélisation efficace, il est préférable de placer ces deux tâches sur un même processeur ce que fait l'heuristique DSC. Par contre, les autres heuristiques, qui sont basées sur une approche gloutonne, ne conduisent pas à cette solution car elles choisissent d'ordonnancer chaque tâche aussi tôt que possible

et choisissent donc de placer ces deux tâches sur deux processeurs différents. Comme les temps de communications sont importants, la tâche suivante est retardée. Ceci est confirmé par l'incapacité de ces heuristiques à profiter de l'augmentation du nombre de processeurs disponibles.

4.10 Conclusion

Le problème de l'ordonnancement et plus particulièrement celui de programmes parallèles sur machine multiprocesseur est un problème bien formalisé. Différents moyens ont été mis en oeuvre pour résoudre ce type de problème en utilisant diverses techniques.

Les résultats présentés ici par les tableaux et les figures permettent de tirer certains enseignements. Par exemple, si on compare les trois heuristiques aléatoires proposées, on constate que celle nommée HL se révèle plus performante et, surtout, beaucoup plus rapide sur les graphes de taille importante. Cette caractéristique rend l'algorithme IS performant. Il nous semble donc que développer une heuristique coûteuse en calculs (même si elle est performante dès la première itération) n'est pas une voie prometteuse dès lors que cette heuristique est couplée à un algorithme génétique ou utilisée de manière répétitive en y intégrant de l'aléatoire. Il faut au contraire trouver des méthodes simples donc rapides afin que l'algorithme génétique puisse parcourir de manière intense l'espace des solutions.

La comparaison entre les algorithmes à représentation directe (FSG1 et 2) et ceux à représentation indirecte (Ahmad, Frode et IS) ne permet pas de tirer des conclusions définitives même si la représentation indirecte donne de meilleurs résultats. Mais cela provient en partie des efficacités relatives des heuristiques utilisées. Concernant l'algorithme génétique proprement dit (représentation indirecte), le croisement se révèle d'autant plus efficace qu'il introduit des changements importants (croisement de Ferland).

L'analyse des courbes obtenues par les quatre algorithmes les plus représentatifs montre bien que le comportement de chaque algorithme dépend fortement du problème considéré (comparaison entre les figures 8.16 et 8.17 par exemple).

Les mesures effectuées ici ont mis en évidence l'intérêt des algorithmes génétiques. En effet, les algorithmes génétiques hybrides mis en oeuvre ont été plus performants que les heuristiques sur lesquelles ils étaient construits. Si l'algorithme FSG est moins performant que ceux à représentation indirecte, cela provient de l'heuristique qui semble moins performante mais aussi du coût de la coupure nécessaire lors du croisement qui pénalise fortement cet algorithme.

Cependant, comme le montre l'analyse de l'algorithme de HAR effectuée dans [CFR96], un algorithme génétique mal adapté peut fournir des résultats quelconques. De même, les heuristiques ne sont guère performantes et elles sont trop sujettes à des détails pour être performantes de manière générale.

La simplicité et la rapidité d'exécution de l'heuristique sont donc primordiales pour l'efficacité des algorithmes génétiques hybrides. Par contre, les deux approches sont chacunes justifiables, surtout que leurs résultats sont très voisins. Dans ce cas précis, l'approche directe est pénalisée par le coût de sa mise

en oeuvre (coût de l'heuristique et des précalculs nécessaires au croisement). En revanche, pour d'autres problèmes, les performances de cette approche pourraient être meilleures si les coûts des opérateurs étaient similaires à ceux de l'approche indirecte. L'efficacité des algorithmes hybrides permet de penser que la coopération d'algorithmes de natures différentes (les algorithmes d'équipes) est une voie de recherche à explorer. Un algorithme d'équipe est un ensemble d'algorithmes qui fonctionnent de manière isolée. Contrairement aux algorithmes hybrides, ils s'échangent seulement des informations concernant le problème mais ces informations ne sont pas nécessaires au fonctionnement de chaque algorithme.

CHAPITRE 5

Placement

5.1 Introduction

L'électronique a connu, depuis sa création, un succès fulgurant dans de nombreux domaines où elle est devenue indispensable. Notamment, les **circuits imprimés** et les **circuits intégrés** ont permis de développer de nombreux appareils toujours plus petits, plus performants et plus économes en énergie électrique grâce à une intégration toujours plus poussée des composants.

On cherche à concevoir des circuits électroniques plus petits pour des raisons de performances. On cherche à minimiser la longueur des connexions.

Lors de la conception d'un circuit électronique, on doit placer des composants (ou cellules) sur un circuit imprimé [SSR91, SM91]. On doit, bien sûr, les placer sur une plaquette la plus petite possible mais aussi, par soucis d'encombrement et de performances, les connexions entre composants doivent être les plus courtes possibles. En effet, les connexions entre composants limitent souvent les performances de ceux-ci : les conducteurs n'étant pas parfaits, les connexions sont sensibles à l'effet Joule (chaleur dissipée, mauvais rendement,...) ainsi qu'aux perturbations électromagnétiques. De plus, les connexions prennent de la place sur la plaquette.

5.2 Les problèmes de placement

Il existe nombre de domaines où le placement intervient : découpe de tissus, organisation de bâtiments, ...

Il existe trois types de placements de composants électroniques.

Ici, nous nous intéresserons au placement de composants électroniques. Ce problème du placement étant un problème très concret, il prend différentes formes suivant les modélisations faites et les hypothèses faites. En effet, suivant le processus au cours duquel il intervient, certaines conditions ou hypothèses peuvent être raisonnables ou non. Il existe trois grandes catégories de problèmes : le placement de portes logiques, celui de cellules standards et enfin le problème de placement de cellules diverses qui nous concerne [Swa93].

5.2.1 Le placement de portes logiques

Dans ce type de placement, on dispose d'une matrice bi-dimensionnelle composée de cellules identiques comme présenté à la figure 5.1. Chaque cellule comprend un nombre fixé de transistors et ces cellules sont séparées par des **canaux de communication**. Chaque cellule peut être utilisée pour implémenter

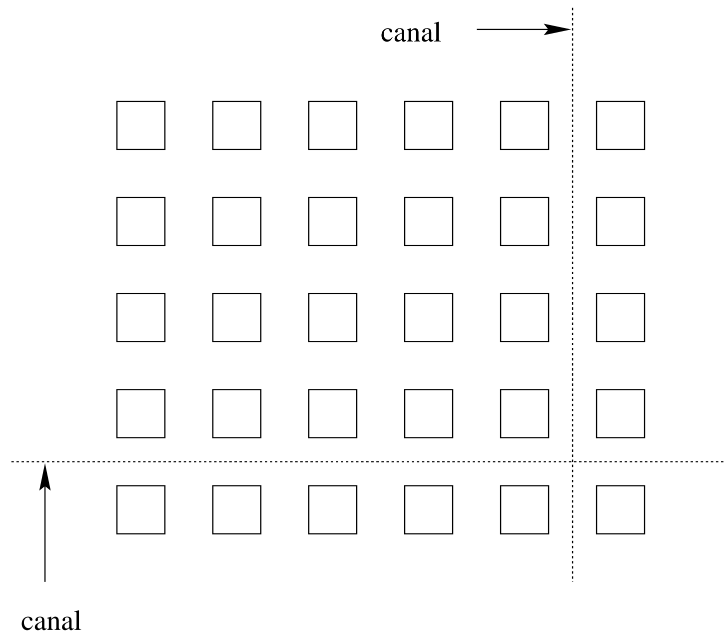


Figure 5.1: Le placement de portes logiques

Le placement de portes logiques utilisent des cellules de taille identique.

une porte logique (ET, OU, ...). Pour cela, il suffit de spécifier les connexions entre les transistors qui composent la cellule. Les liaisons entre cellules se font par l'intermédiaire des canaux de communications qui sont des zones réservées aux connexions. Dans ce type de placement, la mesure de la qualité est fonction des longueurs de connexions et trouver un placement revient à déterminer pour chaque cellule sa position sur la grille.

5.2.2 Le placement de cellules standards

Dans le placement de cellules standards, toutes les cellules ont une dimension identique.

Dans ce type de conception de circuits, le créateur construit son circuit en sélectionnant des modules parmi une bibliothèque de modules qui implémentent des fonctions prédéfinies. Si, dans le placement de portes logiques, toutes les cellules ont la même dimension, dans le placement de cellules standards les cellules ont toutes la même hauteur mais des largeurs différentes. Les cellules sont arrangées en lignes comme à la figure 5.2. Chaque cellule a ses connecteurs qui sont disposés à son sommet ou à son pied. Les connexions se font par des fils qui sont à l'intérieur des canaux de communications se trouvant entre ou autour des lignes de cellules.

5.2.3 Le placement de cellules diverses

Les cellules diverses sont rectangulaires.

Les méthodes de placement proposées précédemment ont l'avantage de réduire le temps de conception. Cependant, l'utilisation de la place n'est pas forcément satisfaisante, notamment lorsque la complexité des circuits VLSI est devenue importante du fait des critères sur la dimension des cellules. Dans le placement de cellules diverses, les dimensions de cellules sont diverses. Cependant, on considère ici que les cellules sont de formes rectangulaires comme

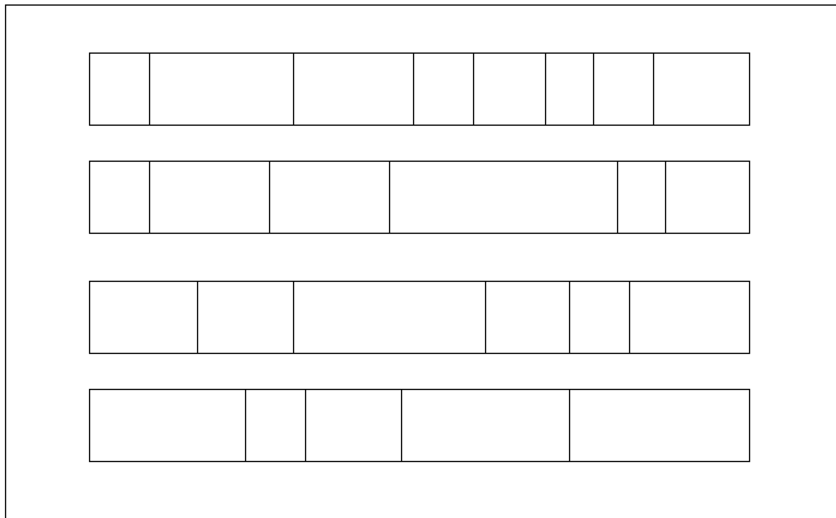


Figure 5.2: Le placement de cellules standards

c'est le cas à la figure 5.3.

5.3 Les travaux déjà effectués

Des outils ont été développés pour résoudre ce problème comme TimberWolf [Swa93] qui est basé sur une approche à base de recuit simulé. L'outil TimberWolf (*maintenant devenu commercial*) est un outil générique qui permet de résoudre les différents problèmes de placement. Il effectue, de plus, le routage des connexions. Il existe aussi des algorithmes génétiques comme celui présenté par [HA92] pour le cas des cellules standards. Cependant, peu d'approches ont été proposées pour le placement de cellules diverses, exceptée celle de TimberWolf.

5.4 Notre modélisation

S'agissant d'un problème concret, la modélisation est une étape primordiale. En effet, il faut que les hypothèses faites soient à la fois réalistes et qu'en même temps elles simplifient considérablement ce problème ardu. J'ai donc choisi la modélisation suivante qui me semble un bon compromis entre la complexité du problème réel et la simplification nécessaire aux calculs:

5.4.1 La plaquette

Une plaquette est composée de rectangles. Elle peut être de dimension infinie.

Il s'agit du support où les composants seront placés et qui assurera les connexions électriques. Dans notre cas, on supposera que tous les composants sont placés du même côté de la plaquette. C'est le cas pour beaucoup de circuits imprimés et il est facile de généraliser l'algorithme génétique, si besoin est, pour tenir compte du cas où l'on pourrait placer les composants des deux côtés.

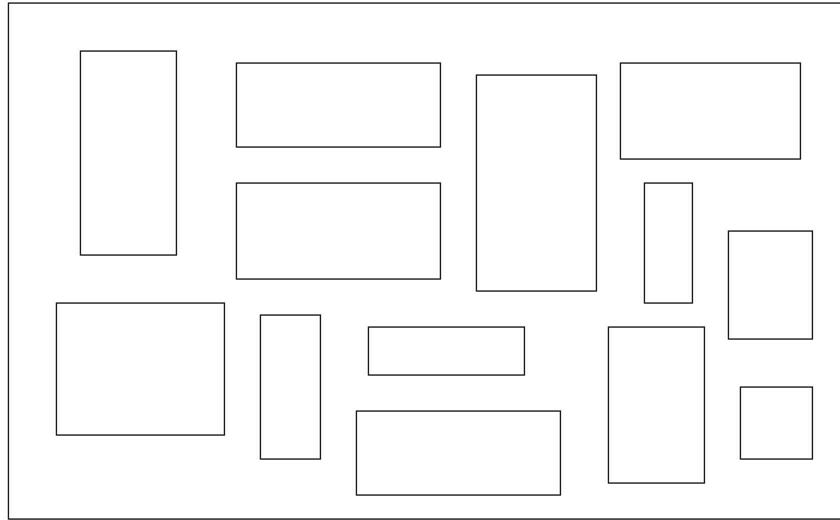


Figure 5.3: Le placement de cellules diverses

On suppose que la plaquette a une forme rectangulaire. Néanmoins, il a été prévu que la forme de la plaquette puisse être plus compliquée et donc, j'ai choisi de considérer que cette forme résultait d'un assemblage de rectangles. Ainsi, la figure 5.4 montre une plaquette P composée de deux rectangles R_1 et R_2:

$$R_1 = \{(x_0, x_1), (y_0, y_1)\},$$

$$R_2 = \{(x_0, x_2), (y_1, y_2)\}$$

On considérera que les dimensions de la plaquette sont des **données** du problème. Cependant, il est prévu qu'une ou plusieurs dimensions puissent être infinies. Par exemple, les valeurs de y_2 ou de x_2 peuvent être de valeur infinie.

5.4.2 Les composants

Un composant est de forme rectangulaire.

Un composant est considéré comme un rectangle. Ce choix est judicieux à plusieurs titres : tout d'abord, la plupart des composants électroniques sont rectangulaires ou de formes proches (puces, résistances,. . .) ou alors de très petites tailles (condensateurs,. . .). Ensuite, le choix de se restreindre aux formes rectangulaires permet de n'envisager que les 4 rotations possibles de chaque composant. Un composant C_c est donc défini par:

1. **sa taille**: sx_c et sy_c
2. **ses connecteurs**: n_c son nombre de connecteurs et $\{(x_i, y_i), 0 < i < n_c\}$, la position de chaque connecteur sur le composant.

Les connecteurs

Un composant est équipé de connecteurs ponctuels.

Un composant est relié aux autres composants grâce à des connecteurs disposés à sa base qui vont être soudés sur le circuit imprimé. A chaque con-

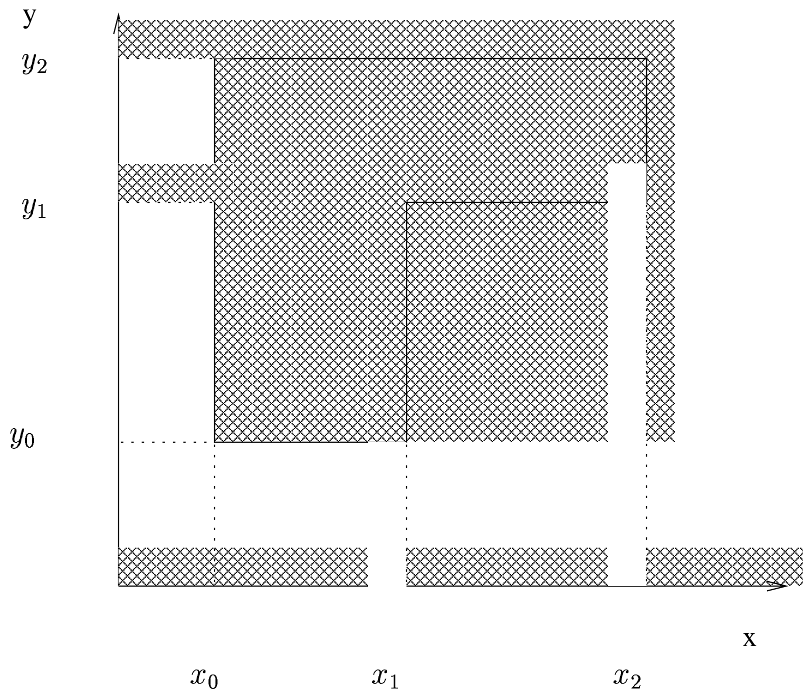


Figure 5.4: Une plaquette

necteur est associé un emplacement géographique sur le composant. Ceci implique donc qu'un composant, qui est considéré comme rectangulaire, perde ses axes de symétrie sauf cas très particuliers que nous négligerons. La figure 5.5 montre un composant avec la disposition de ses connecteurs.

5.4.3 Les connexions

Une connexion relie des connecteurs de différents composants.

La particularité du placement en électronique est que les composants interagissent entre eux via des connexions électriques. Ces connexions relient deux ou plusieurs connecteurs entre eux. Une connexion Cx est donc définie par la liste des couples (composant, connecteur) qu'elle utilise.

$$Cx = \{(C_i, n_i), i = 1, \ldots, m\}$$

5.4.4 La longueur des connexions

Une connexion est un lien électrique réalisé à la surface de la plaquette entre deux ou plusieurs connecteurs d'un ou plusieurs composants. Ainsi, chaque connexion est représentée par un ensemble de couples (*composant, connecteur*). Pour des raisons de performances du circuit, on souhaite, bien sûr, minimiser la longueur des connexions.

La longueur d'une connexion est le demi-périmètre du rectangle englobant.

En général, les liens sont physiquement réalisés sur la plaquette par des lignes parallèles à un bord de la plaquette rectangulaire. Ainsi, on prendra comme mesure de distance la distance dite de Manhattan [CM91].

Dans le cas d'une connexion entre plus de deux connecteurs, on prendra comme mesure de distance le **demi-périmètre** du rectangle englobant tous

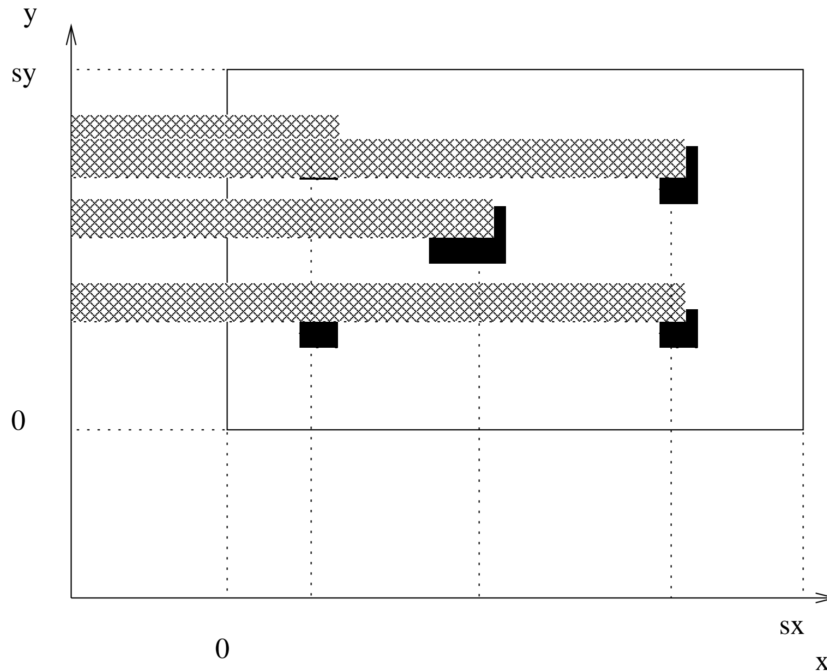


Figure 5.5: Un composant

les connecteurs concernés. Autrement dit, on utilise la distance de Manhattan entre les deux points les plus extrêmes pour chacune des deux dimensions. La figure 5.6 illustre ce choix. Les points représentent les connecteurs et les pointillés un choix de connexion possible.

Un connecteur n'est relié que par l'intermédiaire d'une seule connexion. En effet, si un connecteur était présent dans deux connexions différentes, cela entraînerait que tous les connecteurs de deux liaisons soient reliés au même potentiel via ce connecteur donc ils seraient interconnectés entre eux.

Il faut noter ici que le routage ne sera pas effectué par les méthodes que nous proposerons. En effet, cela compliquerait beaucoup le problème surtout que l'ajout d'une cellule à une solution provisoire pourrait se révéler impossible (ou peu performant) s'il ne restait plus assez de place pour faire passer les connexions nécessaires. On néglige donc la place occupée par les liaisons sur la plaquette. En effet, cette place ne peut se déterminer qu'une fois le routage effectué. De plus, cette place est, en général, assez réduite par rapport à celle des composants. Cependant, on peut facilement augmenter artificiellement la taille de chaque composant afin de laisser de la place libre entre les composants pour les connexions. La politique suivie pour les connexions est de faire leur routage une fois le placement effectué. Au besoin, on pourra déplacer légèrement certaines cellules.

5.4.5 Une heuristique

L'heuristique est un algorithme de liste glouton.

L'heuristique que je propose ici est un algorithme de liste qui construit pas-à-pas la solution à partir d'une liste de priorités. A chaque étape, un

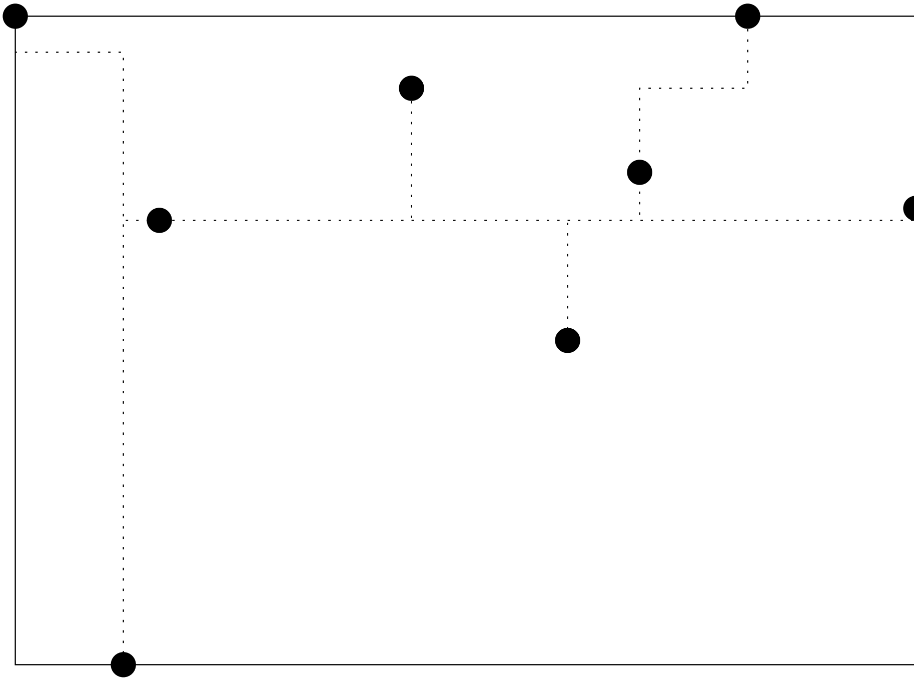


Figure 5.6: La longueur des connexions

composant est choisi et placé de façon à minimiser les connexions existantes après son placement.

Pour ce faire, on maintient à jour la liste des rectangles libres comme expliqué à la figure 5.7. Au départ, la plaquette P est, par hypothèse, une forme composée de rectangles (deux dans l'exemple : $\{((0,0),(x_3,sy)\}$ et $\{(0,y_2),(sx,sy)\}$). Chaque fois qu'un composant est placé, on détermine l'ensemble de rectangles qui sont recouverts en partie par ce composant. Ces rectangles sont supprimés de la liste des rectangles libres. Pour chacun des rectangles supprimés, on décompose la surface restante en rectangles représentant la surface libre qui sont ajoutés à la liste des rectangles libres. Ainsi, après le placement du composant hachuré, on dispose de la liste des rectangles libres suivante :

$$\{(0,y_3),(sx,sy)\};\{(x_2,y_2),(sx,sy)\};\{(0,0),(x_3,y_1)\};$$
$$\{(x_2,0),(x_3,sy)\};\{(0,0),(x_1,sy)\}$$

En représentant ainsi l'espace libre, il est facile de trouver le meilleur placement pour le composant à placer.

Pour placer un élément, on parcourt la liste des rectangles libres pour trouver la meilleure solution possible. Ce parcours est répété quatre fois suivant les quatre rotations possibles du composant. Pour chaque rectangle, la meilleure solution est calculée. En effet, compte-tenu des modélisations faites, il est facile de calculer directement le meilleur placement dans un rectangle libre.

En effet, il y a **indépendance** entre le placement suivant l'axe des x et celui des y. Cette indépendance provient du fait que les positions valides suivant un axe ne dépendent pas de la position suivant l'autre axe. En effet, dans le rectangle $(x_1,y_1),(x_2,y_2)$, x peut prendre n'importe quelle valeur telle

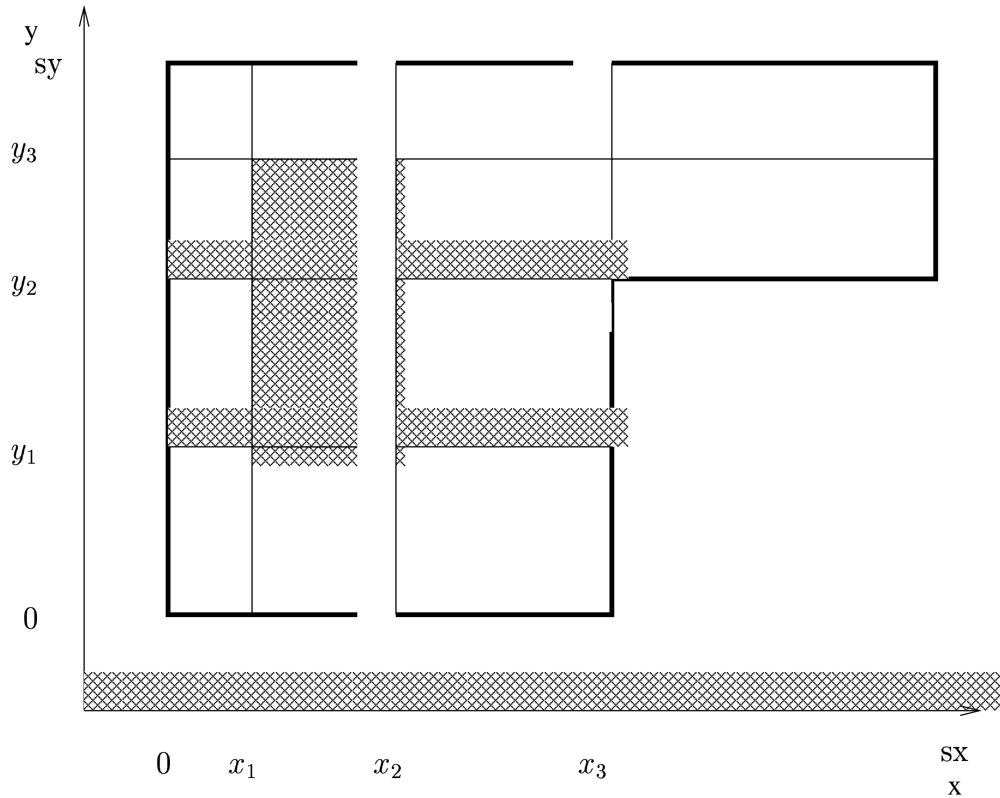


Figure 5.7: Espace libre

que $x_1 \leq x \leq x_2$ et y telle que $y_1 \leq y \leq y_2$. Comme la distance est celle de *Manhattan*, le calcul de la longueur d'une connexion est le demi-périmètre du rectangle englobant tous les connecteurs utilisés par cette connexion.

Soit L la longueur totale des connexions déjà effectuées. Lorsqu'on place un nouveau composant, on cherche à minimiser ΔL qui est l'accroissement de cette longueur. Or, $\Delta L = \sum_{connexions} \Delta L_c$ où L_c est la longueur de la connexion c. Comme il y a indépendance, $\Delta L = \sum_{connexions}(\Delta Lx_c + \Delta Ly_c)$. Lx est la distance de la connexion suivant l'axe des x. $\Delta Lx_c = min(|x - x_{min}|, |x - x_{max}|)$ où x est l'emplacement du connecteur concerné par cette connexion, x_{min} et x_{max} les deux points définissant les coordonnées suivant x du rectangle englobant cette connexion. ΔLx_c est donc affine par morceaux, donc ΔL aussi[1]. Ceci simplifie beaucoup le calcul de la position optimale du composant dans un rectangle donné. En effet, on peut précalculer les points d'inflexion.

Ainsi, pour trouver le meilleur emplacement du composant dans un rectangle libre, il suffit de calculer l'augmentation de la longueur des connexions pour les quatre cas extrêmes (les deux extrémités de chaque dimension) et de comparer cette augmentation avec celle des points d'inflexion correspondant à l'intérieur du rectangle.

On a expliqué précédemment que la plaquette pouvait, ou non, avoir une

[1] Cela reste valide si une connexion utilise plus d'un connecteur sur un composant.

dimension infinie. Si la plaquette est de dimension finie, rien ne garantit que l'heuristique pourra placer tous les composants dessus (soit aucune solution existe, soit les choix de l'heuristique mènent à une impossibilité). En cas d'impossibilité, l'heuristique modifie de manière aléatoire l'ordre sur les tâches avant de recommencer au début son placement. Ceci peut paraître simpliste mais il faut bien voir qu'en général, on cherche à minimiser la longueur des connexions[2] et que l'on dispose d'une surface relativement grande.

Cette heuristique gère de manière la plus fidèle possible l'espace libre au cours de la construction de la solution en maintenant la liste des rectangles libres. Cependant, cette fidélité à la réalité a un coût: cette liste de rectangle libre peut croître de façon exponentielle puisqu'à chaque étape, chaque rectangle libre peut être partagé en deux ou plus autres rectangles libres. Cela rend l'heuristique bien adaptée aux problèmes de petites tailles où l'on modélise fidèlement la réalité. Par contre, il serait obligatoire de modifier la politique du placement du composant pour des problèmes de plus grande taille. Cela peut se faire suivant deux voies: soit on essaye de réduire le nombre de rectangles dits libres en aggrégeants des rectangles voisins (au prix d'une certaine approximation de la réalité), soit on ne garde pas du tout en mémoire l'espace libre sur la plaquette. Dans ce dernier cas, on cherchera un espace libre pour placer le composant traité au dernier moment.

5.5 Un algorithme génétique à représentation indirecte

On vient de présenter un algorithme de liste. Cependant, au lieu de générer aléatoirement cette liste de priorités, on peut utiliser, comme au chapitre précédent, un algorithme génétique ce que j'ai fait. Cet algorithme est nommé AGI pour "Algorithme Génétique Indirect".

L'algorithme génétique utilisé ici pour générer un ordre total sur les cellules à placer est simulaire à celui présenté à la section 4.8, page 53. La seule différence est en fait la fonction de santé qui utilisé l'algorithme de liste précédent pour construire et estimer la qualité de la solution construite. Comme c'est le cas pour la plupart des algorithmes génétiques à représentation indirecte, un chromosome est une liste ordonnée (une "chaîne") des différentes cellules du problème. J'utilise pour effectuer la **sélection** une roulette biaisée (section 3.3.2, page 23).

Le croisement

Comem c'est la cas pour la plupart des algorithmes génétiques à représentation indirecte, un chromosome est une chaîne qui représente la liste ordonnée des éléments traitée par l'algorithme de liste. Ici, le chromosome est donc une liste de toutes les cellules à placer.

[2] Donc indirectement la taille du circuit.

Le croisement

Compte-tenu des résultats du chapitre précédent et ceux obtenus par [FF94], j'ai choisi et implémenté le croisement présenté par [FF94] et résumé dans cet ouvrage à la page 53 et par la figure 4.9, page 54.

La mutation

La plupart des mutations purement génétiques échangent des éléments ou des séquences d'éléments. Compte-tenu de la faible longueur de chaque chromosome (≤ 100) pour le jeu de tests utilisés, j'ai retenu d'implémenter une mutation simple à effectuer. Elle consiste à choisir aléatoirement deux éléments dans la chaîne du chromosome concerné et à les permutter.

5.6 Un algorithme génétique à représentation directe

J'ai aussi développé un algorithme à représentation directe pour résoudre ce problème. Il sera, par la suite, appelé AGD pour "Algorithme Génétique Direct".

5.6.1 La représentation

La représentation n'a que peu d'importance vu que les opérateurs se basent sur la solution et non sur le codage de celle-ci.

Un chromosome doit représenter une solution. Ainsi, on doit coder pour chaque composant sa position suivant l'axe des x et des y ainsi que sa rotation. Le codage par chaîne comme pour l'algorithme FSG, décrit page 41, n'est pas "proche" de la solution qui utilise les deux dimensions du plan. Cependant, il n'y a pas de structure de données pratique et simple qui permettrait de tenir compte de la répartition planaire des composants. Mais, comme on veut utiliser comme opérateurs des heuristiques et non pas des opérateurs purement génétiques qui utiliseraient la structure du chromosome, le codage a peu ou pas d'importance du moment que les opérateurs travaillent directement sur la solution.

Le codage choisi est donc un tableau de taille n (n est le nombre de composants) où chaque élément code la position et la localisation du composant correspondant. Chaque composant est représenté par un entier qui indique le nombre de rotations qu'il a subies et deux réels pour localiser la position de l'angle du composant.

5.6.2 Initialisation

L'initialisation se fait en utilisant l'heuristique. Un ordre total entre les composants est choisi aléatoirement et l'heuristique est utilisée sur cet ordre total pour déterminer une solution initiale.

5.6.3 Croisement

L'opérateur de croisement a pour but de générer à partir de deux individus parents(ou deux solutions), deux individus fils qui présentent des ca-

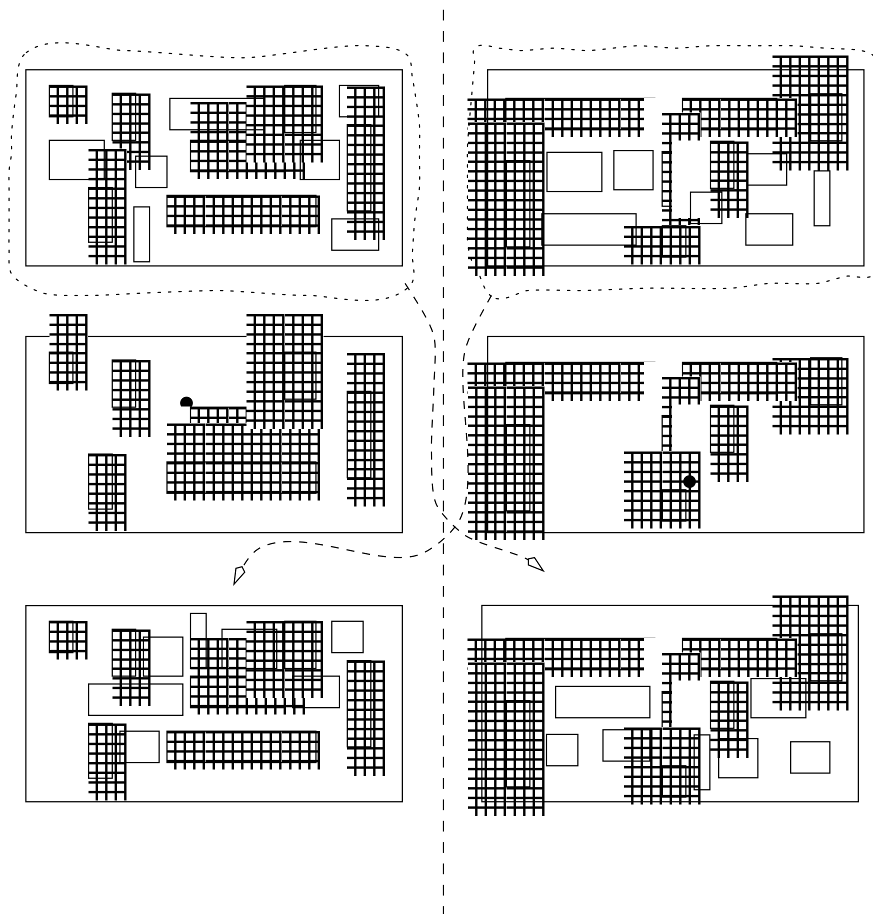


Figure 5.8: Le croisement

ractérisques de **chacun** de leurs parents. De plus, comme il s'agit d'un algorithme hybride, il est nécessaire qu'au cours du croisement on fasse appel à une heuristique.

Ce croisement est expliqué par la figure 5.8. La première étape du croisement consiste à déterminer de manière aléatoire la moitié des composants. Ceux-ci sont hachurés sur la figure.

Ensuite, deux chromosomes fils sont créés, chacun étant la copie de l'un des deux parents, copies restreintes aux composants choisis. On détermine alors le barycentre des centres de ces composants pour chaque nouveau chromosome. Ce barycentre est représenté par un point noir sur la figure. Ainsi, l'individu fils "ressemble" à un de ses parents puisqu'il en est sa copie partielle.

La dernière étape doit permettre de fournir une solution complète, utiliser une heuristique et tenir compte des informations contenues par l'autre parent. J'ai choisi d'utiliser l'heuristique de liste décrite précédemment. Pour ce faire, on a besoin d'un ordre sur les composants à placer. Cet ordre est obtenu en triant pour chaque chromosome parent les composants (seulement ceux non choisis) en fonction de leur distance par rapport au barycentre calculé. Chaque individu fils est alors complété en plaçant les cellules manquantes grâce à

l'algorithme de liste; algorithme qui prend comme ordre sur les cellules l'ordre calculé à partir de l'autre parent (les cellules les plus proche du barycentre d'abord). On obtient ainsi les deux individus qui sont représentés au bas de la figure 5.8.

Ainsi, chaque individu fils répond bien aux critères souhaités. En effet, il dépend de chacun de ses parents et il est aussi sensible à l'algorithme de liste.

5.6.4 Mutation

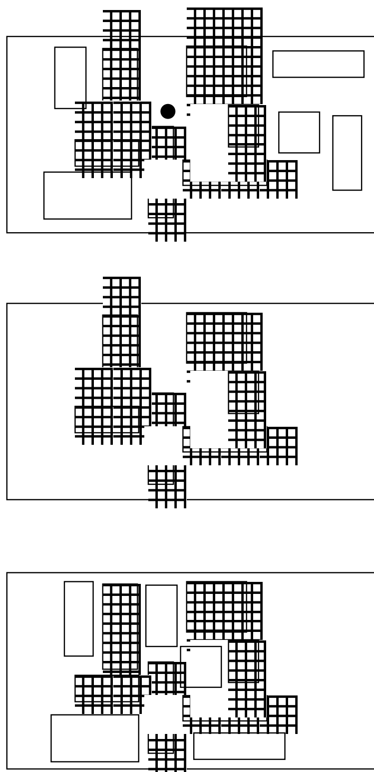


Figure 5.9: La mutation

La mutation utilise aussi l'heuristique ci-dessus. La mutation a pour but, outre d'utiliser les connaissances de l'heuristique, d'introduire un changement, une pertubation, dans le chromosome donc dans la solution. La mutation présentée ici est proche du croisement décrit précédemment.

La mutation est résumée par la figure 5.9. Le principe de cette mutation est de chercher à optimiser la solution représentée par le chromosome. L'idée consiste à conserver sans changement une zone de la plaquette pour reconstruire une solution à partir de cette zone en utilisant l'algorithme de liste pour optimiser la solution produite.

Pour ce faire, la première étape est de choisir aléatoirement un point sur la plaquette comme le point noir de la figure 5.9. Ensuite, on effectue le tri des composants avec comme critère la distance entre leur centre et ce point.

La deuxième étape consiste à choisir un entier x ($0 \leq$x$\leq N$, N étant le nombre total de composants ou cellules du problème.) Dans l'exemple de la

figure, x vaut 5. Les x composants les plus proche du point choisi restent immobiles (Ils sont hachurés sur la figure).

La dernière étape consiste à utiliser l'algorithme de liste pour compléter la solution partielle. L'algorithme de liste place les cellules de manière gloutonne. Les cellules sont placées suivant l'ordre précalculé; c'est-à-dire que les plus proches sont placées d'abord.

circuit	temps en millisecondes	nb de composants
1	9,7	10
1-b	7,4	10
2	81,8	25
2-b	109,9	25
3	80	25
4	137	27
5	958	50
5-b	4 929	50
6	787	50
6-b	7 045	50
7	1 655	50
8	20 341	80
9	22 449	80
10	12 299	100
11	18 805	100

Tableau 5.1: Temps moyen de l'heuristique

5.7 Résultats

Le jeu de tests a été généré de manière aléatoire faute de jeux de tests reconnus et adéquats. Les caractériques de ce jeu de tests sont résumées par le tableau 5.2. Pour les deux jeux de tests 1 et 1-b, la solution optimale que pouvait générer l'heuristique a été calculée. En effet, compte-tenu du faible nombre de pièces (10), l'heuristique de liste peut donner 10! résultats différents. Pour le jeu 1, cette solution a un coût de 73861 et pour 1-b un coût de 74474. L'heuristique et l'algorithme AGI atteignent cet optimum tandis que AGD atteint des valeurs inférieures (respectivement 71531 et 71484). En effet, AGD étant un algorithme à représentation directe, on peut obtenir des solutions qui ne sont pas le résultat d'une application directe de l'heuristique : une solution peut être obtenue par modifications successives. Ainsi, l'espace des solutions atteignables est plus vaste (au sens large) que celui des algorithmes à représentation indirecte.

L'algorithme à représentation indirecte est le plus performant.

Le tableau 5.3 montre les résultats obtenus avec chacun des deux algorithmes génétiques ainsi que par l'heuristique avec, comme entrée, un ordre aléatoire sur les composants. Tous ces algorithmes ont été exécutés durant deux heures sur un PC à processeur Pentium Pro à 200 Mhz, la dernière colonne représentant le résultat obtenu par une exécution de l'heuristique.

Contrairement à l'algorithme génétique à représentation directe du chapitre précédent, celui développé pour ce problème a un croisement dont le

circuit	nb total de connecteurs	nb de connexions	nb de connecteurs utilisés	surface des composants	surface autorisée
1	520	16	66	56 092 672	-
1b	520	16	66	56 092 672	111 220 000
2	212	10	128	43 901 230	-
2-b	212	10	128	43 901 230	87 796 900
3	325	10	127	3 236 080	-
4	451	99	394	32 007 933	-
5	527	25	258	30 126 327	-
5-b	527	25	258	30 126 327	60 248 644
6	527	25	258	55 556 227	-
6-b	527	25	258	55 556 227	111 112 681
7	573	40	517	6 819 991	-
8	959	50	716	111 593 911	-
9	959	50	716	54 507 011	-
10	1134	50	556	84 396 436	-
11	1134	99	556	121 270 236	-

Tableau 5.2: Le jeu de tests

coût est proche de celui d'une exécution de l'heuristique. En effet, le fait de déterminer un ensemble de cellules, calculer son barycentre et ordonner les cellules suivant leur distance à un point représente peu de calculs face à ceux générés par l'heuristique.

Ces trois approches ont un défaut commun qui est dû à l'heuristique. En effet, lors du placement de chaque composant, la liste des places libres est mise à jour. Pour chaque rectangle libre qui intersecte avec le composant placé, on peut générer jusqu'à quatre nouveaux rectangles libres. Ainsi, le coût du placement d'un composant augmente très rapidement et donc, on est obligé de se limiter à une centaine de cellules. Le tableau 5.1 donne pour chaque jeu de test le temps moyen d'exécution de l'heuristique.

Les figures 8.26 à 8.33 présentées en annexe représentent l'évolution du résultat au cours des deux heures de calculs pour les trois méthodes proposées. On remarque que l'algorithme AGD à représentation directe se révèle le plus performant et confirme le résultat du tableau 5.3. Si on observe les figures concernant les problèmes de taille importante, (circuits 8, 9, 10 et 11), on remarque que les algorithmes génétiques nécessitent un temps long pour obtenir le premier résultat et fournissent peu d'améliorations. Ceci est dû à la lenteur de l'heuristique. L'algorithme génétique a donc besoin de beaucoup de calculs pour passer d'une génération à la suivante.

La fonction de santé utilisée ici ne tient pas compte de la surface utilisée. Quand on observe l'évolution de la surface du rectangle englobant les composants, on constate que cette surface ne diminue pas forcément avec la longueur des connexions. Ceci s'explique par le fait que l'heuristique employée ne cherche pas à placer les cellules dans un rectangle. Elle ne cherche même pas à optimiser la surface utilisée mais seulement à optimiser la longueur des connexions.

circuit	AGD	AGI	heuristique	
	2 heures			une itération
1	**71 531**	74 474	74 474	131 790
1-b	**71 484**	73 861	73 861	93 788
2	**69 658**	70 931	74 931	84 445
2-b	**67 882**	75 312	80 773	96 656
3	**15 953**	16 215	17 801	21 161
4	**286 230**	303 897	305 002	393 459
5	**121 421**	121 440	135 589	170 036
5-b	**123 652**	144 590	152 608	203 263
6	**186 255**	190 695	206 294	241 638
6-b	**189 945**	222 450	229 834	276 163
7	**111 093**	114 301	124 578	130 258
8	766 439	**758 226**	758 734	810 099
9	**510 052**	521 337	524 881	579 131
10	**557 770**	570 542	577 691	606 845
11	704 783	705 024	**699 984**	736 879
Ecart moyen en %	**0,12**	4,84	9,18	32,03
Pire écart en %	**1,08**	17,11	23,42	84,24

Tableau 5.3: Résultats

5.8 Conclusion

Dans ce problème, on distingue bien la différence entre les deux méthodes : la représentation indirecte est "basée" sur **une** heuristique tandis que la représentation directe est moins "dépendante" d'une heuristique et peut en outre utiliser plusieurs heuristiques. Cette dépendance est montrée par les jeux de tests 1 et 1-b où la représentation indirecte (algorithme AGD) arrive à générer une solution non atteignable par l'heuristique, donc par un algorithme à représentation indirecte.

L'heuristique doit être modifiée pour les problèmes de tailles plus importantes. Le routage des connexions permettrait d'être plus près de la réalité.

Contrairement au problème de l'ordonnancement, la représentation directe n'étant pas pénalisée par de coûteux calculs, elle se révèle plus performante que celle indirecte, prisonnière de l'heuristique qu'elle utilise. En outre, si les problèmes de place étaient plus importants, on pourrait, le cas échéant, utiliser une autre heuristique qui minimise la place occupée; le choix pouvant se faire de manière dynamique. Mais ce problème confirme l'efficacité des méthodes hybrides.

Les algorithmes présentés ne tiennent pas compte du problème ardu du routage des connexions. La prise en compte de ce problème peut se faire suivant plusieurs axes : soit une fois les composants placés on s'inquiète du routage en modifiant, si besoin est, la solution, soit le routage se fait au cours de la construction de la solution. Dans les deux cas, il peut être judicieux de placer les composants en laissant de la place aux futures connexions.

Pour rendre l'heuristique adaptée aux circuits disposant de plus de composants, il faut rendre la gestion des espaces libres plus performante. Par exemple, on pourrait, de temps en temps, diminuer le nombre de rectangles libres en supprimant ceux inintéressants, ou ceux trop peu différents de ceux existant déjà.

Une autre solution consisterait à ne pas garder en mémoire les espaces libres de cette façon. Dans ce cas, on pourrait garder dans une structure adaptée la position de toutes les cellules déjà placées. Lors du placement d'une nouvelle cellule, on pourrait commencer par déterminer sa position optimale, tester si l'on peut placer ce composant dans cette position et sinon, regarder la plus proche région où l'on puisse le placer.

Il y a aussi une amélioration qui pourrait augmenter l'efficacité de l'algorithme à représentation directe. En effet, lors du croisement ou de la mutation, on choisit de manière complètement aléatoire les cellules qui seront déplacées. Or, on peut très bien se servir d'une heuristique pour déterminer cet ensemble. Par exemple, on pourrait chercher les composants dont les connecteurs sont proches de la frontière du rectangle englobant tous les connecteurs d'une connexion donnée.

CHAPITRE 6 Réseaux cellulaires

6.1 Description du problème

Depuis l'invention de la radio, les communications **sans fil** ont connu un essor important. Dans l'ordre chronologique, cela a commencé avec la radio au début du siècle, puis cela s'est poursuivi avec la télévision, la radio en bande FM, les satellites de diffusion directe et, depuis quelques années, avec les réseaux cellulaires, notamment ceux de téléphonie comme le standard GSM.

A l'exception des réseaux cellulaires, tous ces médias sont des médias de type **diffusion** : les communications se font essentiellement dans le sens *média* $\longrightarrow$ *particulier* où un **même** message est diffusé à tout le monde.

La contrainte essentielle de ces réseaux est donc une contrainte de **couverture** : il faut que le maximum de gens puissent recevoir de manière correcte le signal radio, et ceci avec le moins d'émetteurs possible. Comme les ondes radio ne traversent pas les obstacles naturels, les professionnels ont choisi pour leurs émetteurs des emplacements les plus élevés possibles : montagnes, collines, tour Eiffel, sommets de bâtiments, etc... Le problème de l'allocation de fréquences n'est pas primordial pour ce genre de réseau, notamment par le faible nombre de canaux utilisés. En outre, le choix des emplacements des émetteurs permet facilement d'obtenir un grand nombre de fréquences disponibles simultanément sur un grand territoire (un pays par exemple).

Les réseaux cellulaires diffusent des messages différents à chaque mobile. En plus de la contrainte de couverture, ils doivent donc disposer de suffisamment de fréquences disponibles.

Depuis quelques années, les progrès techniques ont permis de développer de petits émetteurs-récepteurs autonomes. Ainsi, le téléphone sans fil, notamment avec le standard **GSM**, a connu un succès éclatant. Dans un tel système, de petits terminaux autonomes et légers sont en liaison radio avec des émetteurs fixes et puissants (BTS) qui permettent à l'utilisateur d'être relié au réseau terrestre fixe. Dans ce cas, les communications se font dans les deux sens et sont différentes pour chaque mobile. Cela correspond aux réseaux **cellulaires** où le territoire est décomposé en cellules, c'est-à-dire en zones dépendantes d'un émetteur. Pour assurer le bon fonctionnement d'un tel système, il faut satisfaire deux conditions :

- Le terminal mobile doit être dans la zone de couverture d'une BTS.
- Chaque BTS doit pouvoir fournir à chaque utilisateur une fréquence radio en cas de besoin.

Si le premier point est commun aux réseaux de diffusion, le second point

provient du fait que chaque mobile en fonctionnement nécessite une fréquence[1] qui lui soit propre. En effet, dans ce type de réseau, les communications sont propres à chaque mobile. Cette fréquence ne peut pas être réutilisée par un émetteur proche à cause des interférences que cela produirait.

6.2 Modélisation

Comme il s'agit d'un problème concret, la modélisation n'est pas triviale si l'on veut qu'elle soit fidèle [RC98]. En effet, il faut, comme en physique, modéliser le monde réel pour pouvoir le représenter. Dans notre cas, outre la modélisation physique inhérente aux caractéristiques des ondes radio, il faut modéliser le problème combinatoire en cherchant le meilleur compromis entre la fidélité de la modélisation et la facilité pour trouver une *bonne* solution dans ce modèle.

6.2.1 Le terrain

Le terrain est modélisé par ses différentes altitudes.

Pour pouvoir effectuer des simulations, il est nécessaire de connaître la topologie du terrain. En effet, le relief modifie beaucoup la portée des émetteurs car le signal radio de ces réseaux ne peut pas traverser des obstacles importants (collines, ...). Pour ce faire, on a besoin de cartes qui modélisent le terrain. Ainsi, pour chaque point, on connaît son altitude grâce à un modèle numérique de terrain (DTM).

6.2.2 La couverture

A chaque émetteur est associée sa zone de couverture.

Chaque BTS est caractérisée par ses caractéristiques techniques (puissance d'émission, direction d'émission...) ainsi que sa localisation. Grâce à sa localisation, ses caractéristiques techniques ainsi que le relief, on peut par simulation obtenir la zone de couverture théorique d'une BTS : connaissant le relief, on peut déterminer si une BTS "voit" un point du terrain. Dans ce dernier cas, sachant la puissance d'émission et la distance, on peut déterminer si le signal reçu est suffisamment puissant pour être exploitable. Sinon, on considère que le point n'est pas couvert par la BTS [Cha99].

6.2.3 La demande

La demande en fréquences n'est pas uniforme sur tout le territoire.

Un réseau de téléphonie cellulaire ne doit pas seulement offrir une bonne couverture du terrain comme un réseau de diffusion (radio, télévision) mais aussi assez de fréquences pour que chaque mobile puisse disposer d'une fréquence libre en cas de besoin. Comme cette demande en fréquences n'est pas uniforme sur le territoire (villes, campagne...), il est nécessaire de connaître sa répartition sur le terrain. Ainsi, de la même façon qu'on dispose d'une carte du relief, on dispose d'une carte représentant la *demande* où à chaque point est associée une valeur, valeur qui correspond au nombre de mobiles communiquant en même temps (en moyenne). Bien sûr, cette valeur est une image de la réalité car ce nombre varie au cours du temps. Cependant, compte

[1] Ou une fraction de fréquences dans les futurs systèmes à découpage temporel.

tenu du grand nombre de terminaux en circulation, on choisit une valeur proche du maximum réellement atteint. Ce critère est pris en compte dans nombres de travaux [FM97, IL97, SP96].

6.2.4 Les cellules

La zone de couverture, appelée ZoC (Zone Of Coverage) ou encore cellule, est propre à chaque BTS. Il s'agit de l'ensemble des points de la carte où le signal reçu est suffisamment puissant pour être exploité (cf figure 6.1).

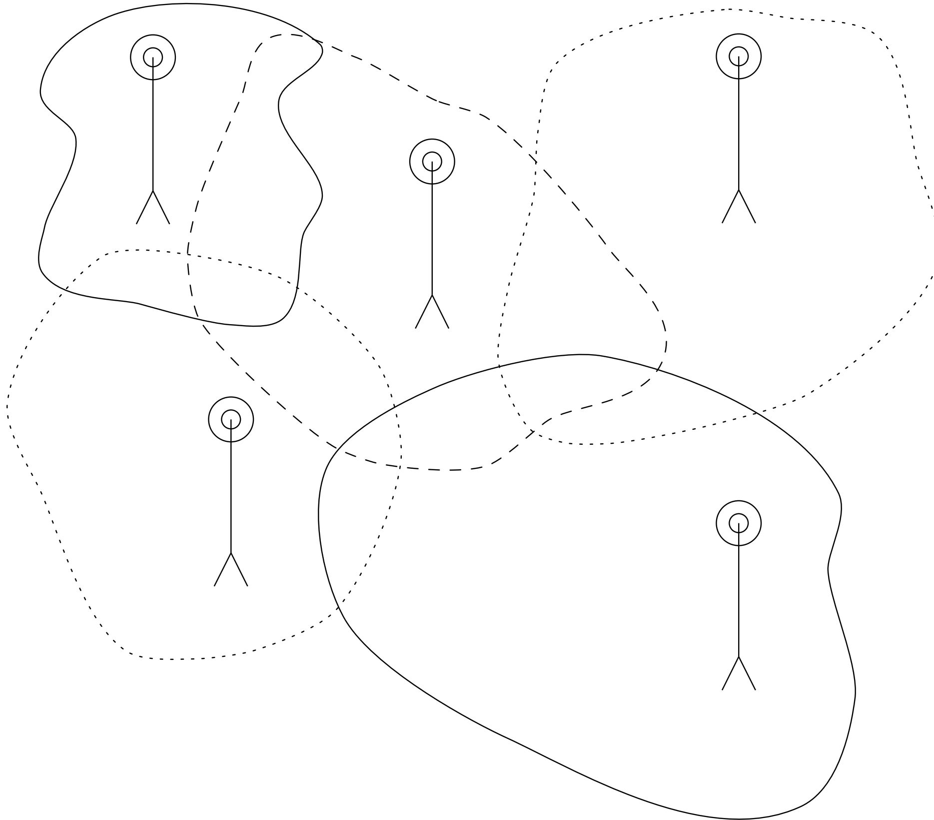


Figure 6.1: Exemple de cellules

Définition 6.2.1
Une **cellule** *est la surface couverte par une BTS.*

Une cellule est la surface couverte par une BTS.

Comme on dispose de la demande, on peut facilement déduire le nombre de personnes qui sont dans la cellule d'une BTS donnée. Bien sûr, l'intersection de deux cellules peut ne pas être vide. Autrement dit, certaines personnes peuvent utiliser indifféremment plusieurs BTS. Malgré cela, le nombre de fréquences dont doit disposer une BTS afin de satisfaire pleinement la demande est seulement fonction de la demande de sa cellule. On ne tient donc pas compte du fait que certains mobiles présents dans la cellule pourraient avoir le choix de communiquer avec plusieurs BTS. Cela s'explique par la simplicité voulue pour

caractériser la qualité d'une solution. De même, compte-tenu que le recouvrement entre les cellules est en général assez uniforme, cette approximation ne rend pas la modélisation plus imprécise.

6.2.5 La zone d'interférences

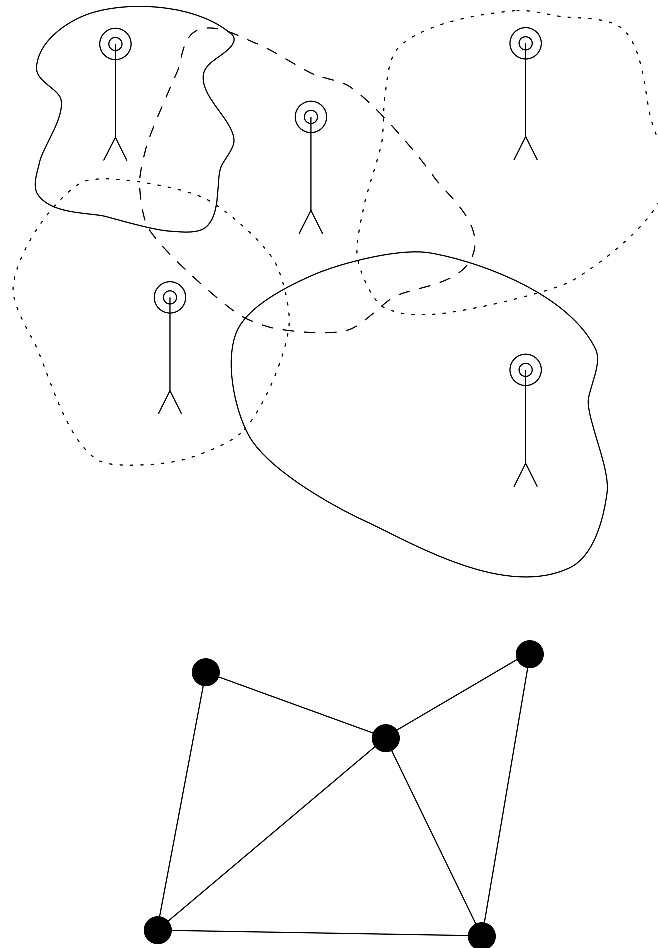


Figure 6.2: Zone d'interférences et graphe d'interférences

La zone d'interférences d'un émetteur est la zone où le signal émis par cet émetteur rend impossible la réception d'un signal de même fréquence d'un autre émetteur.

Définition 6.2.2
La zone d'interférence d'une BTS (ZoI: Zone of Interference) est le terrain où le signal émis par cette BTS est assez fort pour rendre inexploitable tout signal de même fréquence émis par une autre BTS.

Ainsi, la zone d'interférences englobe la zone de couverture car un signal assez faible pour ne pas être exploitable peut quand même perturber la réception de signaux beaucoup plus puissants. On déduit de l'ensemble des zones d'interférences le graphe d'interférences (GI) dont les sommets représentent les BTS possibles. Une arête indique qu'il y a **recouvrement** [2] entre les deux zones d'interférences correspondantes comme le montre la figure 6.2. Une arête

[2] En fait, un certain seuil de recouvrement.

signifie que les deux émetteurs correspondants ne peuvent pas utiliser une même fréquence. De la même façon, on parlera du graphe de couverture pour indiquer le graphe qui est basé sur les cellules de couverture.

6.2.6 Les intercellules

Comme on cherche à obtenir une couverture la plus parfaite possible, on ne peut éviter que deux ou plusieurs cellules se chevauchent. Ceci implique que l'utilisateur peut avoir la possibilité d'utiliser plusieurs émetteurs différents en certains endroits. La notion même de *cellule* ne rend pas compte de cette réalité car un même point géographique peut faire partie de plusieurs cellules et, deux points de la même cellule peuvent avoir un comportement différent par rapport à une autre cellule.

Les intercellules sont le résultat de la décomposition des cellules en zones disjointes.

Pour ce faire, on introduit la notion **d'intercellule**. Une intercellule est une zone géographique où tous les points sont couverts par le même ensemble de cellules. Ainsi, dans le cas où deux cellules A et B s'intersectent, on décompose ces deux cellules en trois intercellules : $A \cap B, A \setminus B$ et $B \setminus A$. L'union de toutes les intercellules correspond donc à la surface couverte par toutes les BTS, et l'intersection de deux intercellules différentes est toujours vide.

Ce concept d'intercellules peut s'appliquer aussi bien aux zones de couverture qu'aux zones d'interférences. Cependant, ici, seule l'application aux zones de couverture (cellules) nous intéresse.

6.3 Le problème

Ce problème se pose depuis des années. En fait, depuis que la radio, surtout FM, et la télévision existent, se pose le problème de couvrir un territoire avec un nombre le plus faible possible d'émetteurs et une attribution de fréquences qui évite les interférences entre émetteurs voisins. Avec le développement de la téléphonie mobile cette contrainte sur les fréquences est devenue critique.

Les réseaux de diffusion

La seule contrainte des réseaux de diffusion est celle de la couverture.

Il s'agit des réseaux où les communications ne sont que dans un seul sens : d'un émetteur vers un récepteur comme la radio, la télévision. Le même signal est diffusé à tous les mobiles. Le but de ces réseaux est donc d'offrir sur une couverture la plus grande possible (en terme de nombre de personnes couvertes) un signal exploitable. La seule contrainte donc pour un émetteur est que ses fréquences d'émissions ne perturbent pas les signaux des émetteurs voisins. Ainsi, les techniques utilisées depuis toujours consistent à placer les émetteurs sur des maximums locaux en termes d'altitude donc en terme de couverture (tour Eiffel, Fourvière,...), l'attribution des fréquences se faisant à la main.

6.4 Etat de l'art

La plupart des travaux génèrent une solution en combinant deux algorithmes différents : le premier construit un réseau de BTS et le second alloue

les fréquences pour ce réseau [Tab97]. Cependant, ces techniques présentent des inconvénients : tout d'abord, il n'y a pas coopération entre les deux phases de la construction de la solution et aucun compromis n'est réalisé. Dans notre cas, on doit choisir les BTS parmi un ensemble de BTS possibles. Cet ensemble de localisations possibles est déterminé par un outil qui effectue un maillage du territoire et extrait de chaque zone un maximum local pour l'altitude.

Les travaux précédents décomposent le problème en deux.

6.4.1 Extraction du réseau

Lors de cette étape, on cherche à extraire de l'ensemble des BTS un réseau dont la couverture soit maximale tout en minimisant les interférences possibles. Pour ce faire, on utilise la notion de *stable maximum* [CJK+97].

Définition 6.4.1
Le stable maximum d'un graphe G est le sous-graphe G' de G ayant le nombre le plus élevé de sommets qui vérifie la propriété suivante : quel que soit le couple de sommets de G', ceux-ci ne sont pas voisins dans G.

On utilise le concept de stable pour déterminer la localisation des émetteurs.

Les sommets du stable maximum sont les sommets ou BTS retenus pour faire partie de la solution. Il ne reste alors plus qu'à effectuer l'allocation de fréquence pour avoir la solution complète du problème. L'intérêt de sélectionner les sommets en utilisant un stable maximum est de choisir un ensemble de sommets minimisant les interférences car si deux sommets couvrant la même zone de territoire (à partir d'une certaine surface), seulement un seul pourra être retenu dans la solution. En outre, cela permet de favoriser le ratio entre la couverture et le nombre d'émetteurs utilisés. Il faut préciser que cela concerne la zone de couverture et que donc, l'allocation de fréquence ne sera pas triviale car il pourra exister des interférences entre des BTS non voisines dans le graphe de couverture mais voisine dans le graphe d'interférence. Cependant, le fait de sélectionner un stable garantie le fait que si deux émetteurs ont deux zones de couverture qui se chevauchent de manière importante, seul l'un des deux sera au plus retenu.

Cette méthode empêche donc de sélectionner deux BTS dont les zones de couverture se recouvrent de manière importante, au détriment de la couverture.

Complexité 6.4.1
La recherche du stable maximum est un problème NP-complet dans le cas général [Joh85].

Comme on ne peut pas trouver de stables maximums dans le cas général, deux choix sont possibles : soit on se restreint à un cas particulier où le problème est de complexité polynomiale, soit on recherche seulement une bonne solution.

La triangulation

Dans le cas d'un graphe général, le problème est trop difficile, mais dans le cas d'un graphe à degré borné, cette recherche est de complexité polynomiale.

Si le graphe d'interférences est, par exemple, un graphe triangulé (chaque BTS est le sommet d'un hexagone et a donc 6 voisins), le degré du graphe

est 6 et donc, il existe des algorithmes polynomiaux pour trouver le stable maximum. Cependant, ce type de méthode n'est pas utilisé à cause de la contrainte qui n'est pas vérifiée en réalité.

Les stables maximaux

Comme trouver le stable maximum est un problème trop complexe dans le cas général, on va se contenter de générer un stable maximal car ce problème a une complexité polynomiale.

Définition 6.4.2
Un stable maximal est un stable qui, par ajout de n'importe quel sommet, perd sa propriété de stable.

Générer un stable maximal est un problème polynomial. Différentes heuristiques existent :

Les heuristiques

Plusieurs méthodes ont été proposées pour trouver des stables maximaux [Ber73, GM90]. Notamment, différentes heuristiques gloutonnes ont été testées [CJK+97]. Ces heuristiques fonctionnent toutes sur le même principe : Les sommets sont ajoutés un par un dans le stable jusqu'à obtenir un stable maximal, c'est-à-dire qu'aucun sommet ne peut être ajouté au stable courant. Parmi les critères testés pour sélectionner le sommet, on peut citer :

1. sommet de plus faible degré.
2. sommet de plus fort degré.
3. couverture la plus forte, sommet de plus petit degré si égalité.
4. degré le plus faible, couverture maximale.
5. sélection aléatoire du sommet.

Ces méthodes semblent très différentes mais chacune suit une certaine logique. Ainsi, la méthode 1 cherche, d'abord, à sélectionner les sommets qui génèrent le moins d'interférences, donc ceux qui ont peu de voisins dans le graphe d'où le critère. En revanche, on peut supposer qu'une BTS qui produit des interférences avec beaucoup de ses voisines est une BTS puissante qui a une grande zone de couverture ; zone de couverture qui peut englober des zones de couverture d'autres BTS moins performantes. Dans ce cas, comme on cherche à maximiser la couverture en minimisant le nombre de BTS, il semble logique de choisir cette BTS puissante d'où la méthode 2. Les résultats montrent que la méthode 1 est plus performante que 2. Cela peut vouloir dire que la logique suivie par la méthode 1 est plus proche de la réalité que celle de la méthode 2.

Comme on cherche à obtenir une couverture maximale, ce paramètre a été introduit pour sélectionner le sommet dans les méthodes 3 et 4. Les expérimentations ont montré que le critère le plus performant était le critère 3 [CJK+97].

Un algorithme génétique

Un algorithme génétique a aussi été testé pour extraire des stables maximaux [CGK+97]. La représentation choisie est la chaîne de booléens de longueur égale au nombre total de BTS. Cette chaîne indique pour chaque BTS si elle fait partie du stable représenté ou non. Le croisement est un croisement classique à un point. Pour que le résultat obtenu après un croisement soit bien un stable, un opérateur de reconstruction est utilisé basé sur l'algorithme de Fujisuma et Nakajima (1992) pour extraire un stable d'un graphe biparti. La mutation consiste à insérer dans le stable un sommet non présent et à supprimer tous ses voisins qui seraient dans le stable afin de respecter les contraintes. La fonction objectif ou santé retenue est $\frac{couverture^{\alpha}}{\#BTS}$, où $\#BTS$ est le nombre d'émetteurs utilisés par la solution. La couverture est la surface de terrain qui est couverte par au moins un émetteur. Les résultats expérimentaux ont montré que $\alpha = 2$ semblait le meilleur choix d'après les avis des spécialiste du domaine.

6.4.2 L'allocation de fréquences

Chaque émetteur couvre une certaine surface qui correspond donc, suivant la densité de population, à une certaine *demande* en nombre de fréquences. Si on fait correspondre à chaque fréquence une couleur, allouer les fréquences revient à effectuer un **multi-coloriage**[3] sous contraintes (nombre de couleurs fixé)[RC97]. On peut cependant facilement se ramener au cas du **coloriage** simplement en remplaçant chaque sommet qui doit être colorié par n couleurs, $n > 1$, par une clique (graphe complet) à n sommets.

L'algorithme de Petford-Welsh

Il s'agit d'une algorithme itératif randomisé [PW89]. Il prend comme entrée un coloriage initial. Ce coloriage est généralement obtenu grâce à un algorithme glouton. L'algorithme cherche à optimiser ce coloriage; c'est-à-dire à diminuer le nombre de couleurs nécessaires. L'algorithme supprime alors une couleur disponible et va essayer de re-colorier le graphe sans cette couleur en réutilisant uen autre couleur. Pour chaque sommet, on calcule la probabilité d'utiliser cette couleur, probilité liée au voisinage du sommet.

Suivant cette probabilité et une température donnée, on recolorie ce sommet avec cette couleur ou non. Ensuite, on détecte les sommets mal coloriés et on réitère ce procédé afin d'essayer d'obtenir un coloriage satisfaisant les contraintes. Si, au bout d'un certain nombres d'itération, on n'a pu obtenir un coloriage satisfaisaint, l'algorithme stoppe. Par contre, si on a obtenu un coloriage correct, l'algorithme essaye de colorier avec un nombre de couleur inférieur.

Compte-tenu de la complexité du problème, l'algorithme peut échouer son coloriage alors qu'une solution existe. Cet algorithme fournit donc, si initialisé avec un nombre de couleurs autorisées suffisamment élevé, une borne supérieure du nombre de couleurs nécessaires.

[3] Un émetteur peut demander plusieurs fréquences donc le sommet correspondant doit avoir plusieurs couleurs.

6.5 L'algorithme génétique en une passe

Les approches précédentes qui décomposent le problème en deux (recherche d'un ensemble de BTS et allocation de fréquences) ont, comme principal inconvénient, de ne pas rechercher un *compromis* entre les deux problèmes. Pour pouvoir réaliser ce compromis, on peut suivre deux voies : une voie consisterait, une fois l'allocation en fréquences faite, à revenir à la phase de la sélection des BTS pour éventuellement modifier cette phase afin d'obtenir un meilleur résultat. Cette méthode de retour arrière est difficile à mettre en oeuvre. En effet, il faut être capable de caractériser les faiblesses de la solution proposée mais aussi d'utiliser ces faiblesses afin de générer une nouvelle solution plus adéquate.

L'autre méthode est celle que j'ai retenue. Au lieu de décomposer le problème en deux, on garde le problème entier et la solution est construite en tenant compte directement des deux critères.

Pour obtenir un compromis entre la couverture et les fréquences disponibles, on utilise une méthode qui construit de manière directe la solution.

6.5.1 La santé

Les travaux jusqu'alors effectués n'ont pas défini de but précis ou plus exactement, le but était clair (meilleure couverture...), mais comme les algorithmes précédents ne génèrent qu'une solution, le travail ne portait pas sur comment quantifier la qualité de la solution trouvée. De même, en l'absence de comparaison entre articles, peu de travaux ont porté sur la modélisation du problème. Or, pour le fonctionnement même d'un algorithme génétique, il est primordial de disposer d'une fonction de santé qui soit la plus réaliste possible. De plus, cela s'avère nécessaire pour comparer des solutions trouvées par différentes méthodes.

On a besoin de quantifier de manière précise la qualité d'une solution.

Dans notre cas, la fonction de santé est basée sur les désagréments potentiels des utilisateurs, c'est-à-dire sur l'impossibilité de fournir un service demandé. Ce service, c'est la possibilité pour le mobile à obtenir une connexion avec une BTS. Cette impossibilité peut provenir de deux causes :

1. mobile en dehors de toute zone de couverture de BTS.

2. toutes les fréquences sont déjà réservées ou utilisées.

On va mesurer le nombre d'*insatisfaits* que la solution génère. La fonction de santé f, c'est-à-dire le nombre d'insatisfaits, est donc décomposée en une somme sur chacune des intercellules.

La qualité d'une solution revient à estimer le nombre de communications non établies.

Si l'ensemble des BTS couvrant une intercellule est vide, alors la fonction de santé restreinte à cette intercellule est égale à la population concernée. Sinon, on calcule pour chaque BTS le pourcentage p_f de fréquences obtenues.

$$p_f = \frac{\text{nombre de fréquences obtenues}}{\text{nombre de fréquences demandées}}$$

La fonction de santé relative à l'intercellule IC_i est :

$$f(IC_i) = (1 - Max_{BTS_{IC_i}}(p_f)) * pop(IC_i)$$

$Max_{BTS_{IC_i}}(p_f)$ est la fonction qui calcule la valeur maximale de p_f pour l'ensemble des BTS couvrant l'intercellule IC_i. $pop(IC_i)$ est la population (ou la demande) de l'intercellule IC_i.

La fonction de santé de la solution est donc

$$s = \sum_{i=1}^{n} f(IC_i) + co\hat{u}t_{BTS} * \#BTS$$

On a supposé que la demande en fréquences d'une BTS était reliée directement à la population de sa cellule. Cela semble logique à première vue. Cependant, à partir du moment où il y a un recouvrement entre les cellules, une même personne peut utiliser indifféremment une des BTS couvrantes et même, puisque le standard GSM prévoit et implémente cette possibilité, de changer de BTS au cours même d'une communication. Ainsi, il serait logique, si deux BTS couvrent exactement le même terrain, que chacune n'ait besoin, en fait, que de la moitié des fréquences normalement demandées.

Cependant, le calcul des fréquences tel qu'expliqué précédemment reste une bonne approximation de la réalité. En effet, le pourcentage de recouvrement des cellules reste dans des proportions raisonnables par rapport à la modélisation faite. Ensuite, cette fonction de santé ainsi définie reste facilement calculable. En effet, si on voulait être plus proche de la réalité, cela impliquerait qu'en rajoutant une BTS à une solution, une ou plusieurs BTS auraient besoin de moins de fréquences et donc, ces fréquences seraient éventuellement disponibles pour d'autres BTS. De proche en proche, la solution pourrait être largement modifiée. Mais surtout, le calcul de la santé serait dépendant de la façon dont on construit la solution (l'ordre d'ajout des BTS).

6.5.2 L'heuristique

Il s'agit d'une heuristique qui construit pas-à-pas la solution en prenant les BTS dans un ordre donné. Il s'agit de nouveau d'un algorithme de liste.

L'algorithme génétique est basé sur un algorithme de liste. Il s'agit d'un algorithme à représentation indirecte.

Pour chaque BTS, elle alloue, si possible, autant de fréquences que nécessaire. Les fréquences sont ordonnées suivant un ordre fixe (celui de leur numéro, de leur longueur d'onde,...). Cela se fait en parcourant dans l'ordre toutes les fréquences et en allouant à la BTS la fréquence si celle-ci n'est pas déjà allouée à une BTS voisine dans le graphe d'interférences.

Ce processus continue jusqu'à ce que l'addition d'une BTS à la solution courante produise une augmentation de la valeur de santé de la solution. Dans ce cas, la dernière BTS traitée est retirée de la solution afin de garder la meilleure solution obtenue.

6.5.3 Le croisement

Le croisement utilisé ici est celui développé par [FF94] et appelé croisement uniforme ordonné. Ce choix semble judicieux vu la structure de l'heuristique. Il a été expliqué en détail à la section 4.8.3, page 53 et par la figure 4.9.

6.5.4 La mutation

Il s'agit d'une mutation classique où deux éléments sont permutés. Comme l'ensemble des localisations possibles n'est pas très grand (une centaine en général), il n'est pas nécessaire de mettre en oeuvre un dispositif comme celui utilisé dans le cas de l'ordonnancement (section 4.8.4).

6.5.5 La sélection

Comme pour les algorithmes précédemment décrits, on utilise une roulette biaisée. Cela permet de garder une population assez hétérogène. Un autre avantage de cette sélection est d'être plus facilement modifiable que celle par tournois car elle ne se fait qu'une fois par génération.

6.6 Résultats

Problème	heuristique		Génétique
	1 itération	40 000 itérations	
uniforme	106 943	56 519	32 326,5
ville	21 118,7	6 703,9	903,4

Tableau 6.1: Résultats: réseaux cellulaires

Le jeu de tests considéré est obtenu à partir de terrains réels sur lesquels on a réparti la demande pour être proche de la réalité. Le terrain est un rectangle d'environ une centaine de kilomètres de côté. Chacun des tests comprend 100 BTS et les gens du métier retiennent aux alentours d'une trentaine de BTS. Nos paramètres sont donc ajustés pour retenir ce pourcentage de BTS. Enfin, la demande sous-jacente n'est jamais nulle pour deux raisons : la première est d'ordre technique car l'heuristique s'arrêterait de construire la solution dès qu'un émetteur couvrirait une zone où la demande est partout nulle (on s'arrête dès que la fonction de santé décroît). L'autre raison vient du fait qu'en général toute zone est plus ou moins peuplée (promeneurs,...) et que le fait de voir sur la carte qu'une zone est couverte a un effet publicitaire positif non négligeable.

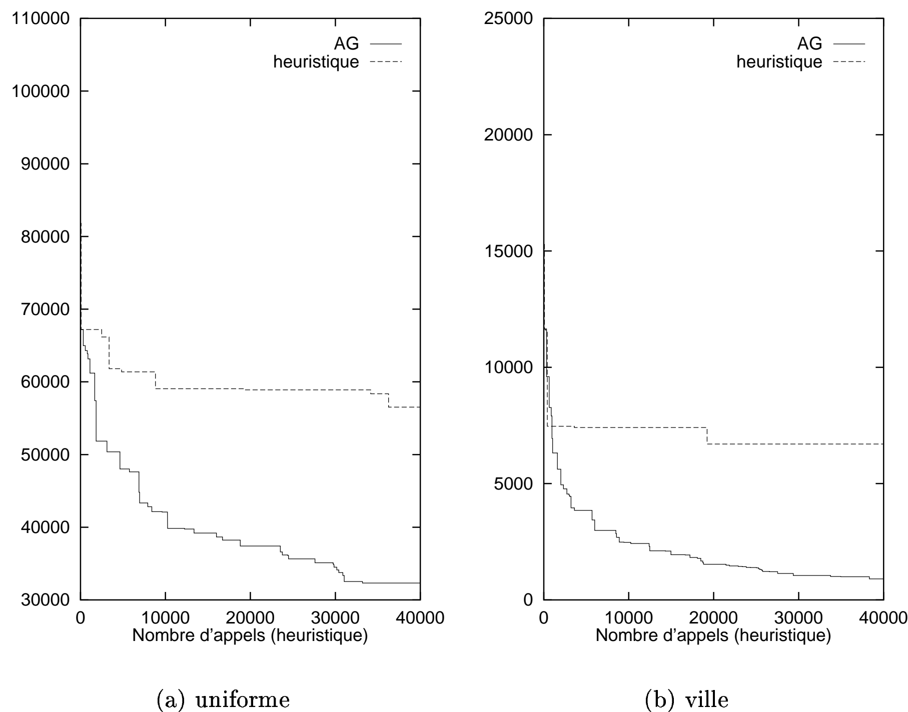
Dans notre cas, on dispose de deux tests différents. Le premier, appelé “uniforme”, représente un territoire où la demande est uniformément répartie. Le second est nommé “ville” car la demande est répartie de manière non-uniforme sur le territoire. L'algorithme génétique est réglé pour effectuer 40 000 appels à l'heuristique.

Le tableau 6.1 indique, pour ces deux jeux de tests, la valeur de santé de la solution obtenue par l'heuristique après une itération, 40 000 itérations et par le génétique après 40 000 appels à l'heuristique. Comme pour les autres problèmes, on voit l'intérêt d'utiliser une heuristique aléatoire car la différence entre les deux résultats de l'heuristique pour chaque test est considérable. De même, ces résultats confirment l'intérêt de l'approche génétique hybride. La figure 6.3 donne pour chacun des deux tests l'évolution de la solution courante au cours de l'exécution de chacune des deux méthodes.

6.7 Conclusion

Tous les travaux présentés ici cherchent à extraire un ensemble de BTS parmi un choix limité de localisations possibles. En plus, dans notre cas, toutes les BTS ont les mêmes caractéristiques physiques (puissance d'émission, directionnalité de l'antenne, ...). S'il est facile d'avoir parmi les BTS des émetteurs de caractéristiques différentes, il serait intéressant de pouvoir tester et proposer à la fois de nouvelles locations pour les BTS ainsi que de nouvelles caractéristiques techniques. Par exemple, si on n'arrive pas à satisfaire la demande sur une zone restreinte, c'est que les BTS proposées sont trop puissantes et interfèrent avec trop d'émetteurs voisins, ce qui limite les fréquences disponibles. On pourrait, dans ce cas, proposer une nouvelle BTS avec une location et des caractéristiques ayant une couverture réduite pour satisfaire la forte demande de cette zone pour remplacer une BTS trop puissante. C'est ce qui est actuellement fait manuellement dans les zones très peuplées (centre-villes, manifestations, congrès,...) où l'on place une (ou plusieurs) BTS de très faible puissance qui couvre une petite surperficie. On décompose ainsi le territoire en micro-cellules afin de pouvoir réutiliser les mêmes fréquences dans des zones proches.

L'algorithme génétique présenté donne de bons résultats. Cependant, il ne propose pas de nouveaux emplacements pour les émetteurs ainsi que de nouvelles caractéristiques techniques.



(a) uniforme (b) ville

Figure 6.3: Résultats: réseaux cellulaires

On a supposé pour l'instant que la demande était **statique**, c'est-à-dire constante au cours du temps. Ceci n'est évidemment qu'une supposition, qu'une approximation. En fait, au cours de la journée, cette répartition évolue avec les déplacement quotidien des utilisateurs. S'il n'est guère envisageable

de modifier la location ou les caractéristiques des BTS[4], on peut parfaitement envisager de modifier l'allocation de fréquences au cours du temps. En effet, on pourrait imaginer qu'un algorithme génétique fonctionne en permanence afin de faire évoluer la solution en fonction des variations de la demande. Cependant, cette modification permettrait-elle d'augmenter de manière significative les performances du réseau cellulaire ?

[4]Il existe cependant des projets où la puissance d'émission mais aussi la direction dans laquelle pointe l'antenne peuvent être changées de manière dynamique par logiciel.

CHAPITRE 7 Conclusion

7.1 Résumé

7.1.1 Les problèmes d'optimisation combinatoire

Les trois problèmes étudiés montrent bien la diversité des problèmes d'optimisation combinatoire. Certains problèmes sont déjà formalisés tandis que d'autres sont trop particuliers pour avoir une modélisation universelle. Dans ce dernier cas, il est obligatoire, avant de commencer à résoudre le problème, d'effectuer une modélisation. Cette phase est complexe car on doit effectuer un compromis entre deux buts contradictoires : la modélisation doit être aussi fidèle que possible de la réalité (c'est-à-dire que tous les paramètres doivent être pris en compte) et on doit éviter de faire des hypothèses restrictives. En même temps, cette modélisation doit déboucher sur un problème le plus "simple" possible à résoudre. Cependant, le problème n'est en général pas assez simplifié pour être de complexité polynomiale. Il faut donc utiliser des techniques pour obtenir une solution approchée. Parmi ces techniques, les algorithmes génétiques donnent des résultats intéressants.

7.1.2 Les algorithmes génétiques

Les algorithmes génétiques ont fait l'objet de beaucoup de travaux de recherche. En effet, à première vue, ils disposent d'atouts intéressants pour les problèmes d'optimisation combinatoire :

- Leur parallélisation est, à priori, facile.
- Ils sont assez robustes.
- Ils sont faciles à mettre en oeuvre car on n'a pas besoin de connaissances précises du problème.
- Le résultat est constamment affiné au cours du temps.

Cependant, ces algorithmes sont-ils efficaces par rapport à d'autres algorithmes de nature différente ? Les expérimentations présentées dans les chapitres précédents le montrent. Cependant, pour obtenir une bonne efficacité, il est nécessaire de coupler l'algorithme génétique avec une heuristique. Pour ce faire, deux méthodes existent suivant qu'un chromosome code directement une solution ou non. Ces deux approches sont donc appelées respectivement représentation directe et indirecte. La comparaison faite entre les

deux méthodes sur le problème de l'ordonnancement et celui du placement montre que le résultat d'une approche dépend essentiellement du problème et de l'implémentation effectuée. D'ailleurs, il est hasardeux de vouloir tirer des conclusions générales à partir d'expérimentations portant sur un unique problème, comme par exemple, pour la taille de la population nécessaire.

7.1.3 L'ordonnancement

Dans le problème de l'ordonnancement statique de tâche sur machine parallèle homogène, les coûts de communications sont pris en compte. Pour ce problème, plusieurs algorithmes génétiques hybrides ont déjà été proposés. Ces algorithmes utilisent l'un ou l'autre des deux types de représentations précédemment expliquées. Compte-tenu du coût du croisement choisi dans l'approche directe, les algorithmes à représentation indirecte se montrent plus performants.

7.1.4 Le placement

Les problèmes de placement sont très divers comme les problèmes d'ordonnancement. Le problème étudié ici est celui des cellules diverses ; c'est-à-dire que l'on suppose seulement que les cellules aient une forme rectangulaire et que la plaquette où les cellules seront placées soit le résultat d'un assemblage de rectangles. Le critère d'optimisation n'est pas la place occupée mais la longueur des connexions. Dans ce cas, l'approche directe se révèle plus performante. Elle arrive même, dans un cas précis, à obtenir une solution meilleure que la meilleure des solutions atteignables par l'algorithme génétique à représentation indirecte ou par l'algorithme de liste. Par contre, la gestion des emplacements libres employée par l'heuristique décrite n'est pas adaptée aux problèmes avec un nombre élevé de composants.

7.1.5 Les réseaux cellulaires

Ce problème consiste à dimensionner un réseau cellulaire (téléphonie mobile par exemple). Les travaux précédents reposaient sur une décomposition du problème en deux sous-problèmes : tout d'abord, l'ensemble des émetteurs était déterminé puis, ensuite, l'allocation de fréquences était faite. L'inconvénient de cette méthode est de ne pas chercher un compromis global. La méthode présentée ici résout le problème tout entier en optimisant non pas un critère particulier mais la qualité de la solution du point de vue de l'utilisateur. Pour ce faire, il a donc fallu mettre au point une fonction de santé facilement calculable pour estimer de manière précise la qualité d'une solution. Une fois cette fonction construite, une heuristique gloutonne et un algorithme génétique à représentation indirecte permettent de générer efficacement de bonnes solutions.

7.2 Travaux futurs

Lorsqu'on cherche à résoudre un problème réel comme celui du placement de composants électroniques ou la construction d'un réseau cellulaire,

on fait, au cours de la modélisation, un compromis entre la fidélité de la modélisation faite par rapport à la réalité et l'efficacité des algorithmes suivant cette modélisation.

Ainsi, pour améliorer un algorithme et trouver une solution réellement meilleure, soit on garde le même modèle soit on change le modèle pour le rendre plus complexe mais plus proche de la réalité. Ces deux voies sont à explorer pour augmenter l'efficacité des algorithmes. Par exemple, cela consisterait à tenir compte de la place prise par les connexions dans le problème du placement. Ceci impliquerait donc de considérer le problème du routage.

7.2.1 Le parallélisme

Les algorithmes présentés jusqu'ici sont des algorithmes séquentiels. Or, ces algorithmes ont un fort potentiel de parallélisation possible. Cette parallélisation peut être menée dans deux voies différentes : soit on garde le même algorithme et on obtiendra le même résultat[1] mais plus rapidement soit on modifie l'algorithme dans l'esprit des algorithmes évolutionnaires. Cette dernière voie consiste à appliquer la technique des îles, décrite au paragraphe 3.6.2, page 30. Les travaux menés montrent que cette dernière solution est certainement la meilleure. Néanmoins, il nous semble intéressant d'expérimenter les algorithmes précédents sur des machines parallèles ou des réseaux de stations.

7.2.2 Les algorithmes d'équipes

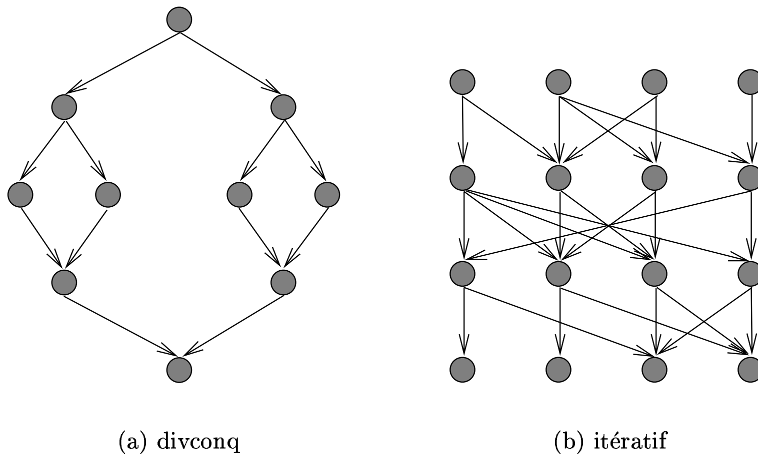
Le parallélisme, notamment la technique des îles, consiste à faire coopérer des algorithmes génétiques identiques. Pour plus d'efficacité et de robustesse, on peut vouloir faire coopérer des algorithmes différents. Il s'agit dans ce cas d'algorithmes d'équipes. La différence essentielle entre un algorithme hybride et un algorithme d'équipe est au niveau des échanges entre les méthodes. Dans le cas d'un algorithme d'équipe, les échanges sont au niveau de la solution du problème (meilleure solution trouvée, borne sur l'optimum,. . .) et sont supposés non bloquants. Par essence, ces algorithmes sont dédiés au parallélisme. Par contre, les algorithmes hybrides sont de nature séquentielle, et les échanges entre méthodes sont à priori bloquants. Cette approche est intéressante car elle permet de tirer profit des caractéristiques des avantages de chaque algorithme. Par exemple, on peut coupler un algorithme génétique avec un algorithme de séparation-évaluation [Cor97]. Cela permettrait d'obtenir la solution avec les performances du génétique et d'évaluer sa qualité voire de démontrer son optimalité avec l'algorithme de séparation-évaluation.

[1] Nous négligeons ici le problème du générateur de nombres aléatoires.

CHAPITRE 8

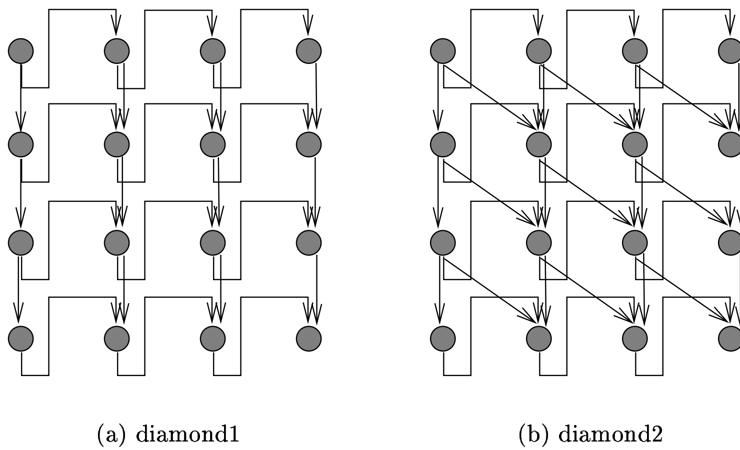
Annexes

8.1 Structure des graphes utilisés



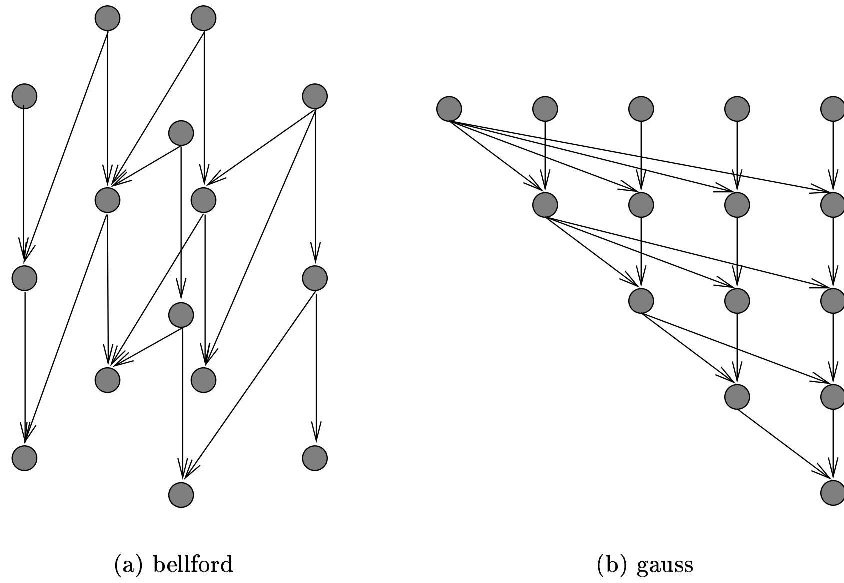
(a) divconq (b) itératif

Figure 8.1: Structure de divconq et itératif



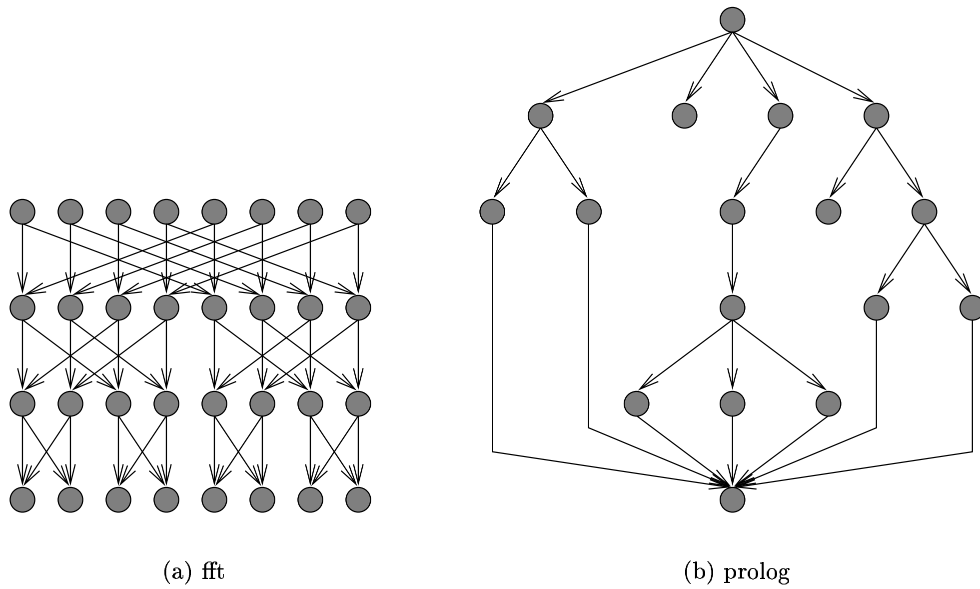
(a) diamond1 (b) diamond2

Figure 8.2: Structure de diamond1 et diamond2



(a) bellford (b) gauss

Figure 8.3: Structure de bellford et gauss



(a) fft (b) prolog

Figure 8.4: Structure de fft et prolog

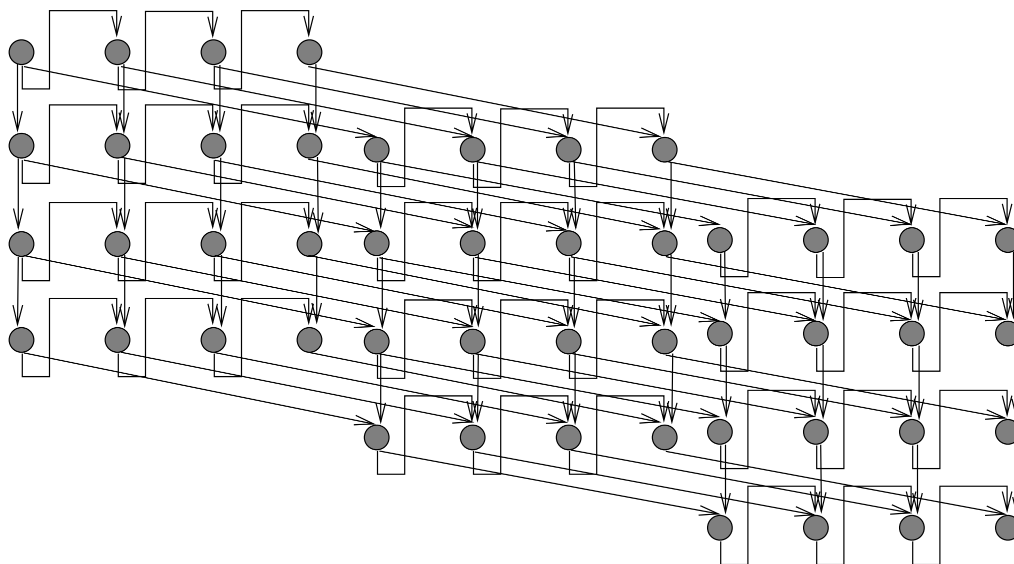


Figure 8.5: Structure de diamond3

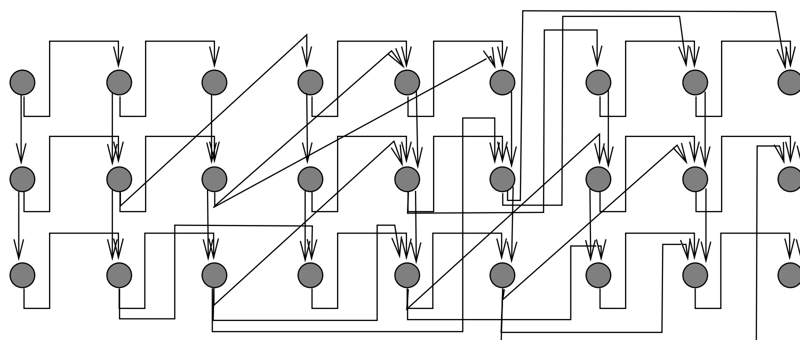


Figure 8.6: Structure de diamond4

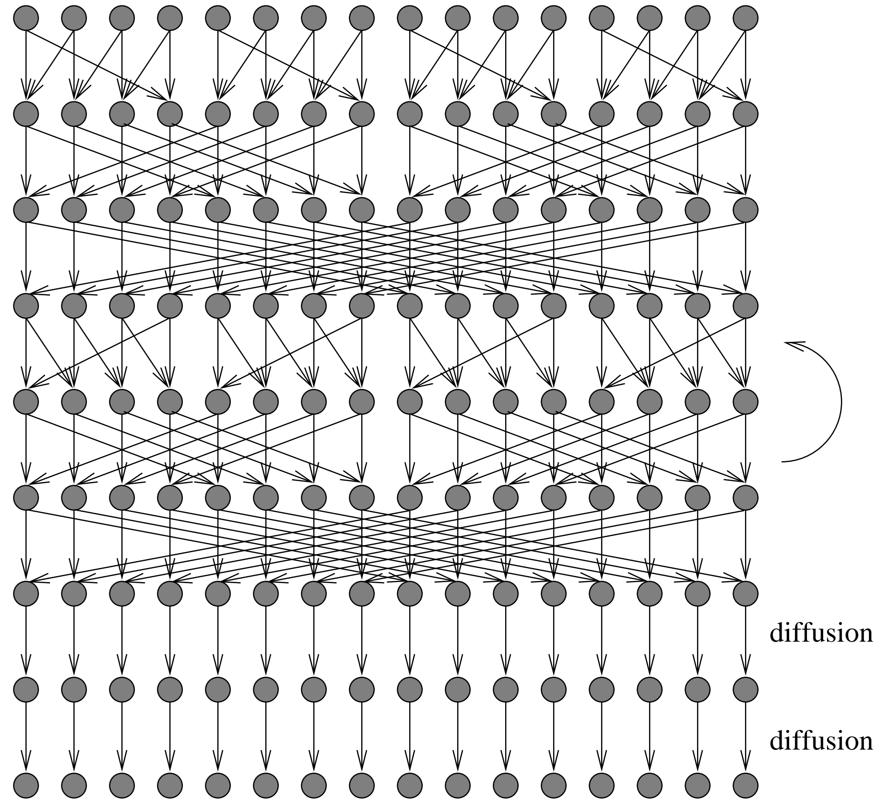


Figure 8.7: Structure de qcd

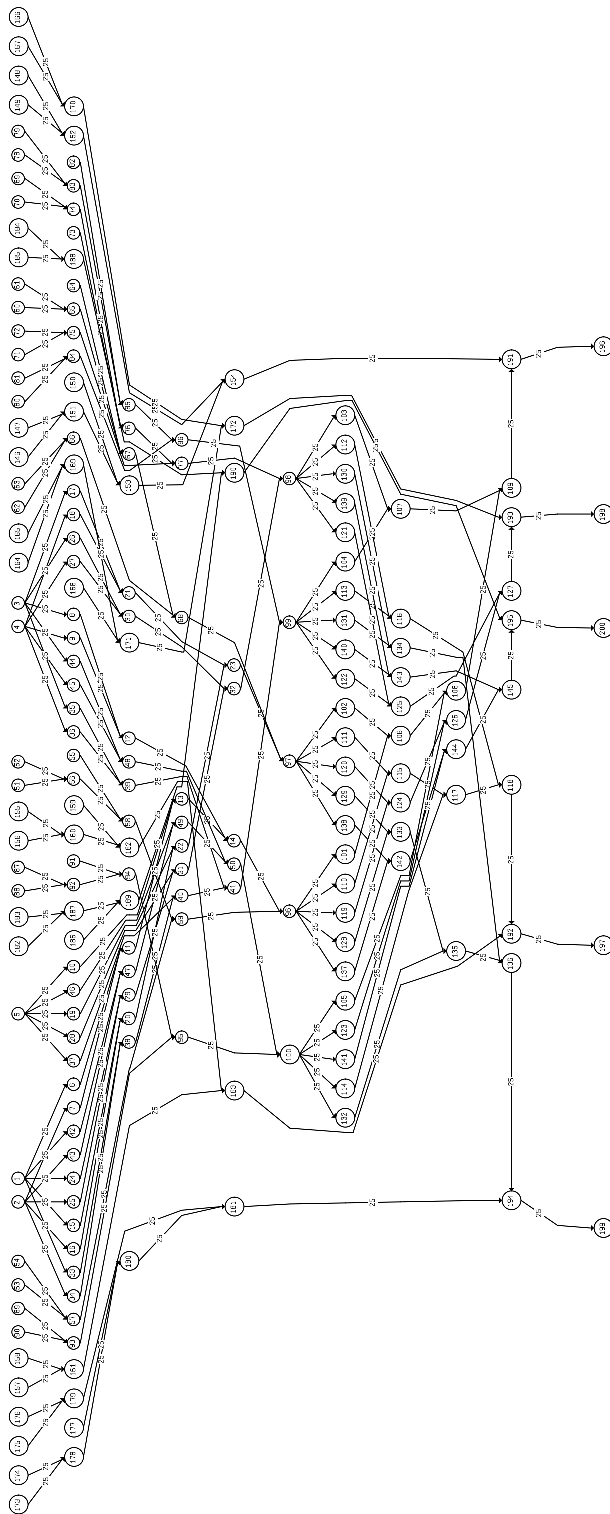
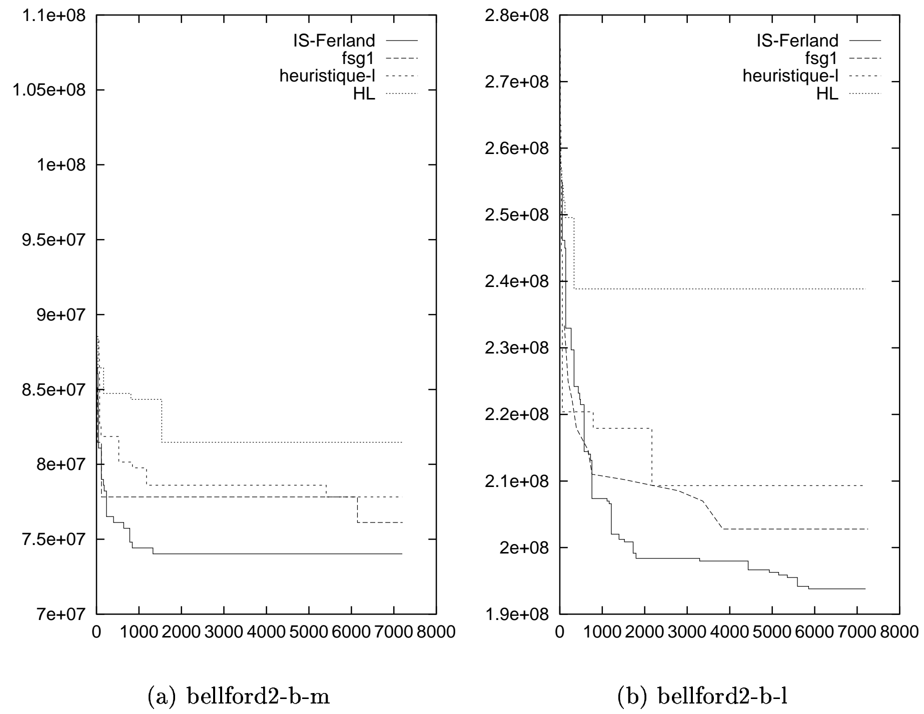


Figure 8.8: Graphe ssc5

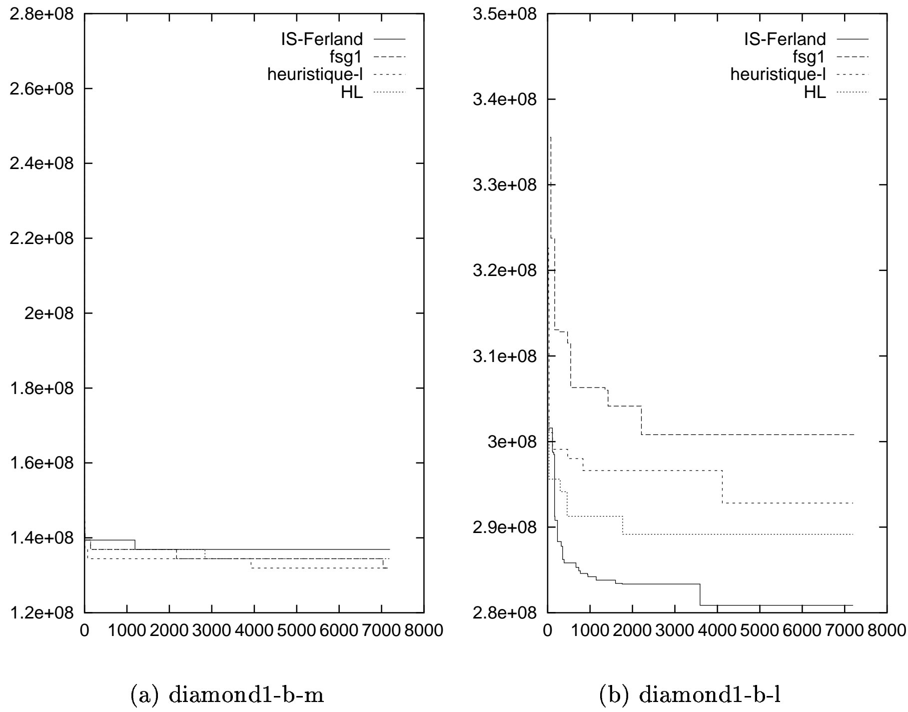
8.2 Résultats pour le problème de l'ordonnancement



(a) bellford2-b-m

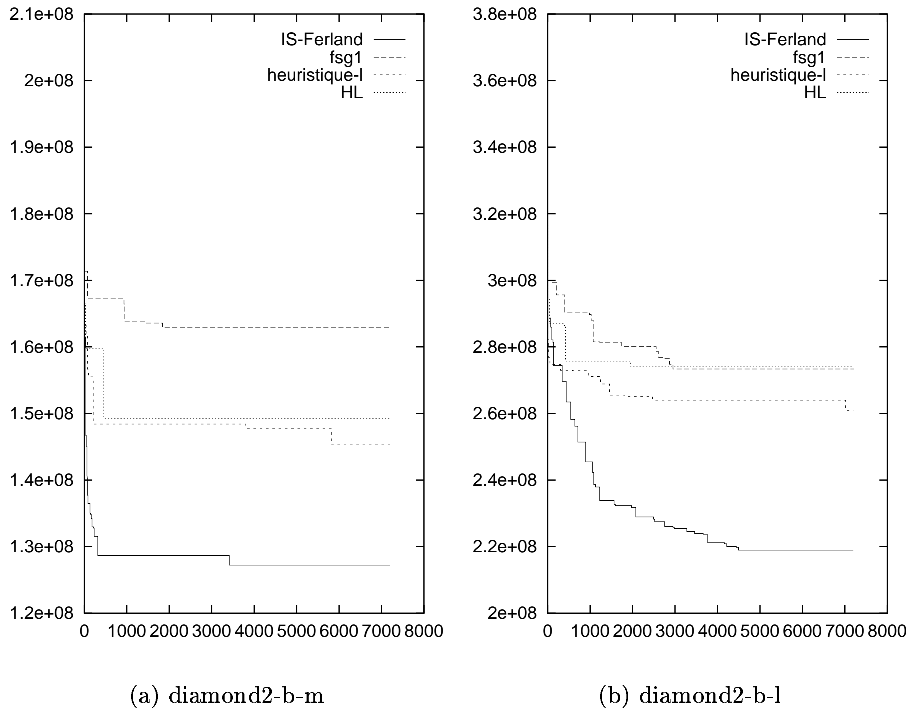
(b) bellford2-b-l

Figure 8.9: Résultats: bellford2-b



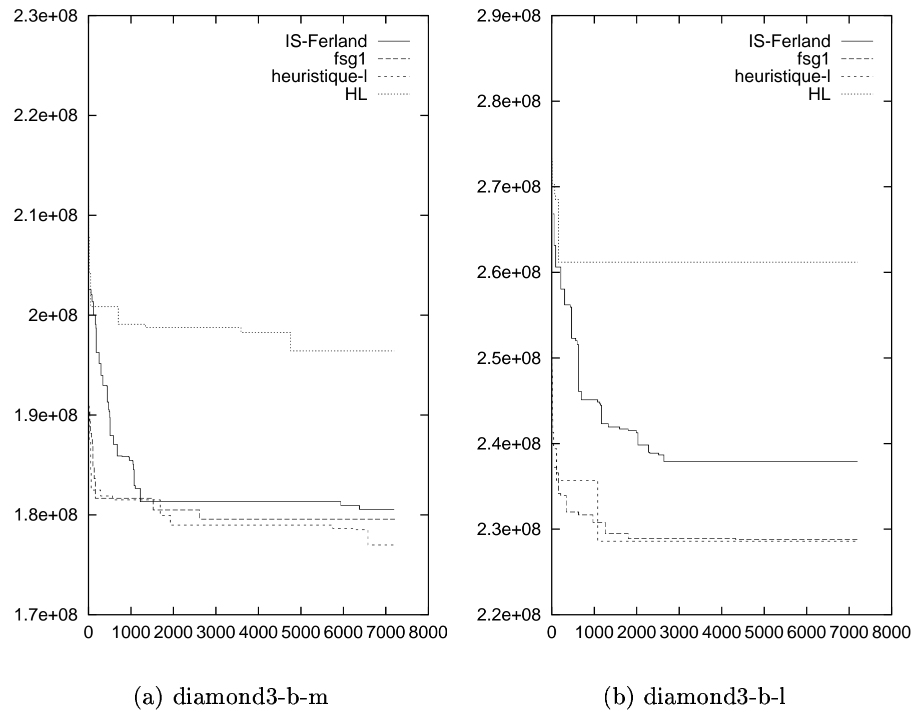
(a) diamond1-b-m (b) diamond1-b-l

Figure 8.10: Résultats: diamond1-b



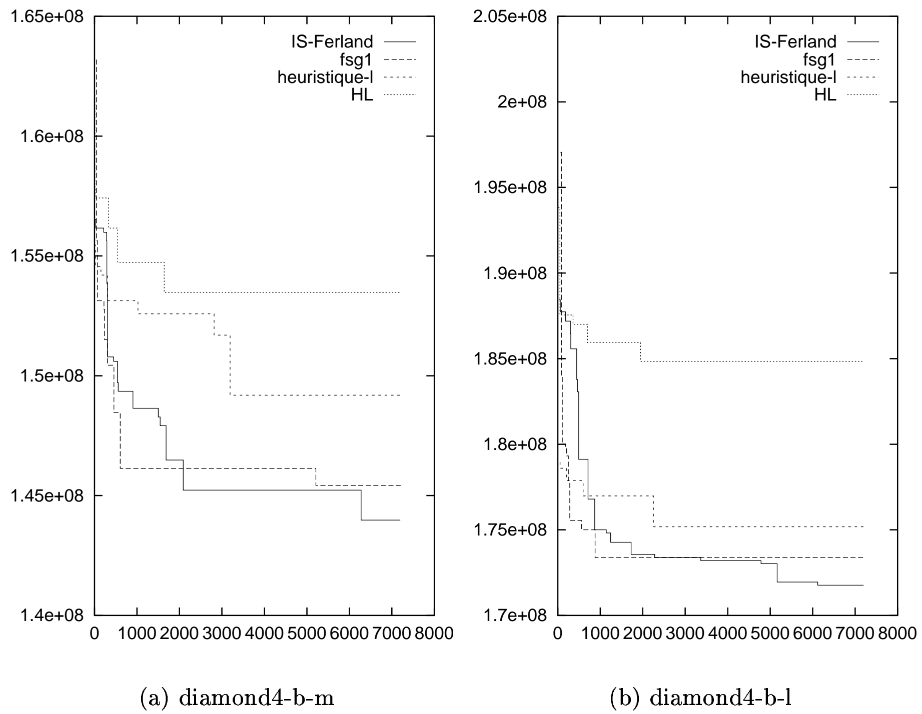
(a) diamond2-b-m (b) diamond2-b-l

Figure 8.11: Résultats: diamond2-b



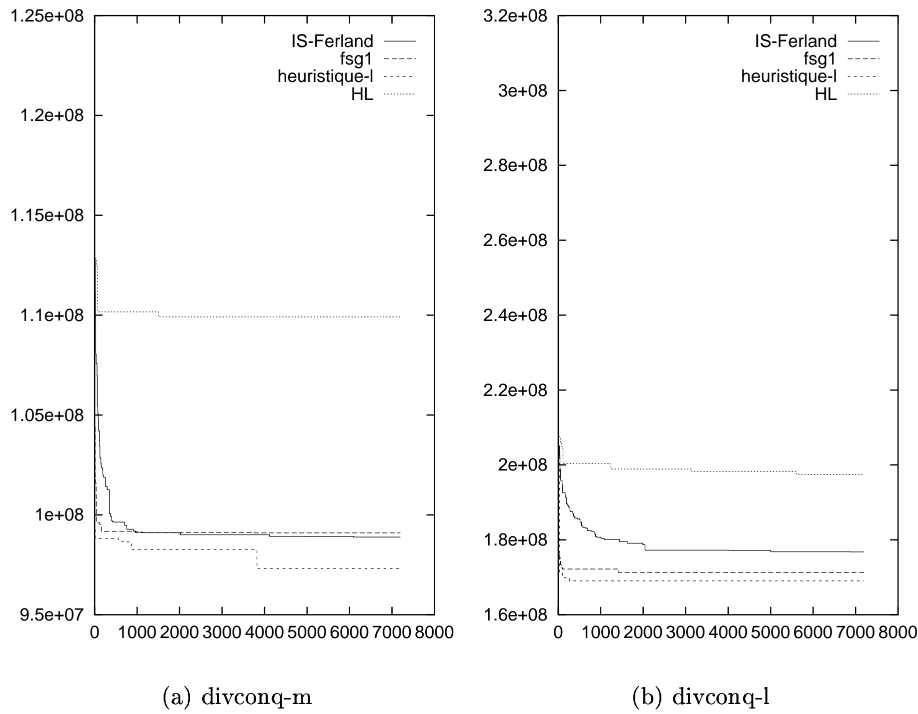
(a) diamond3-b-m (b) diamond3-b-l

Figure 8.12: Résultats: diamond3-b



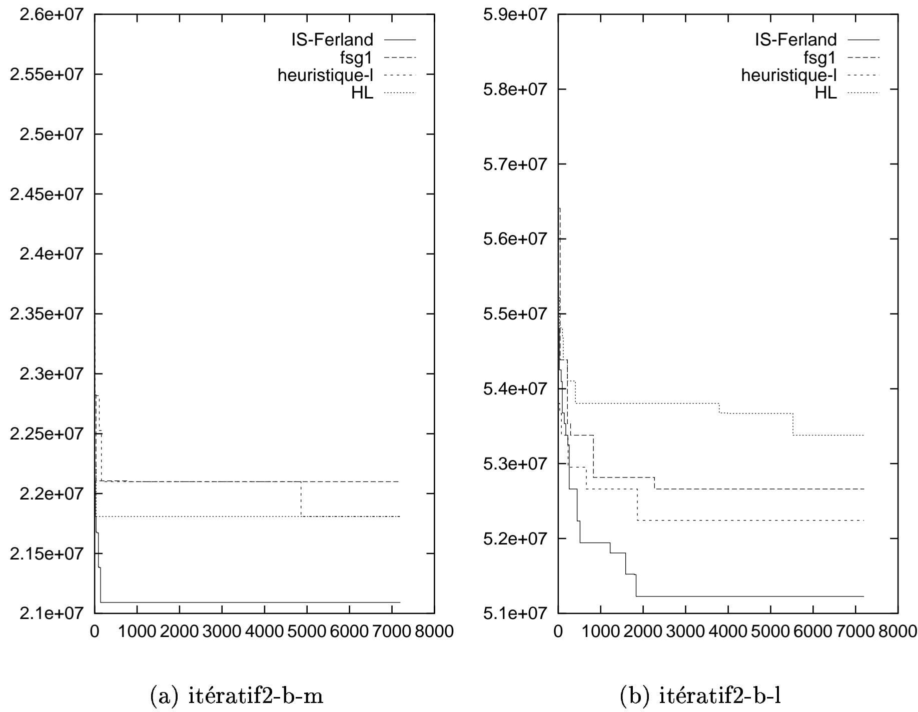
(a) diamond4-b-m (b) diamond4-b-l

Figure 8.13: Résultats: diamond4-b



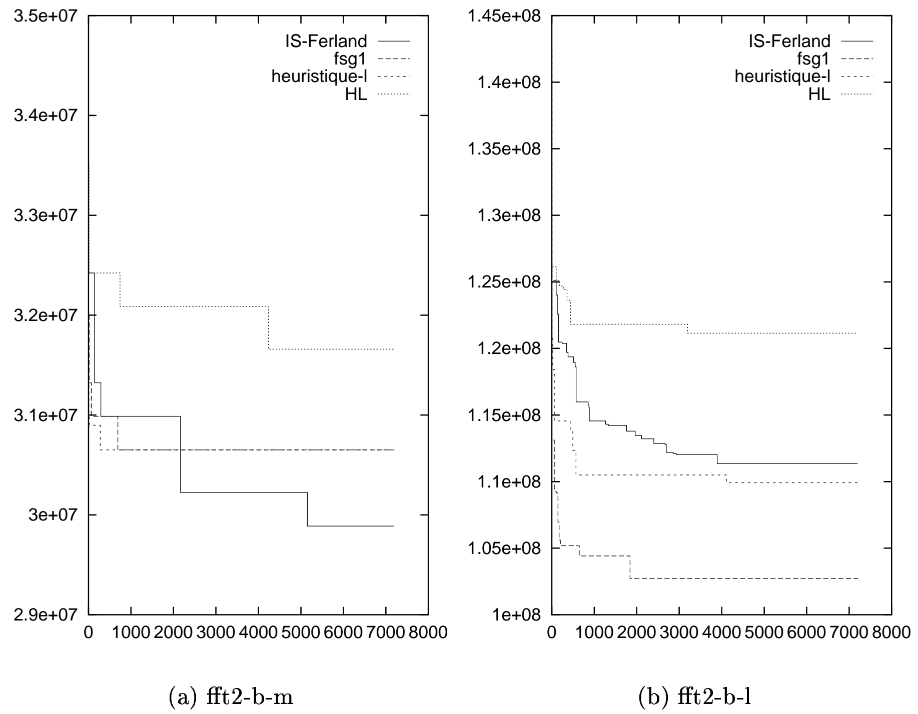
(a) divconq-m (b) divconq-l

Figure 8.14: Résultats: divconq



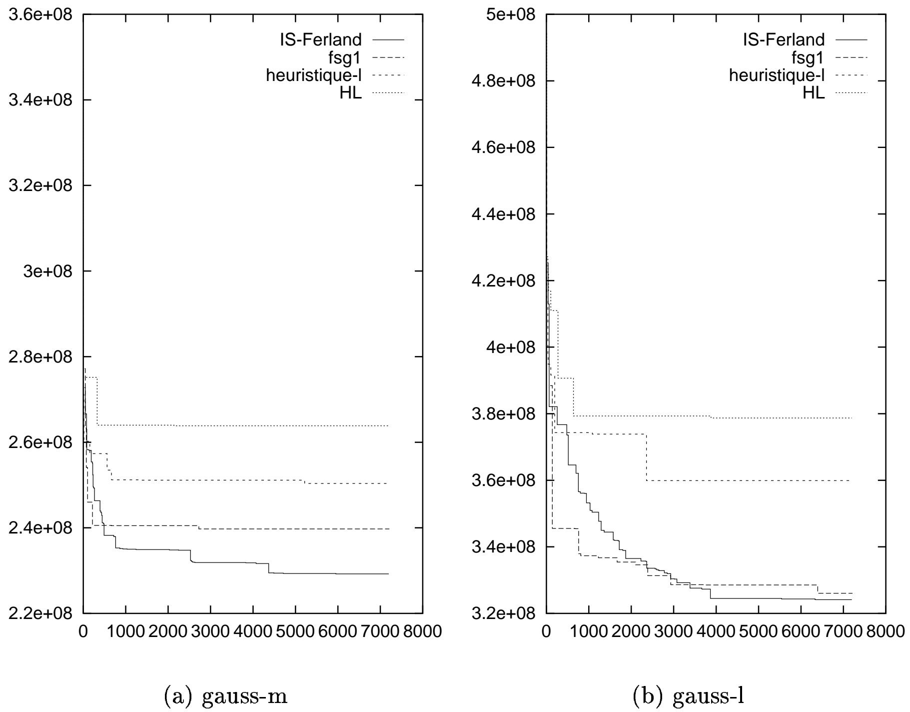
(a) itératif2-b-m (b) itératif2-b-l

Figure 8.15: Résultats: itératif2-b



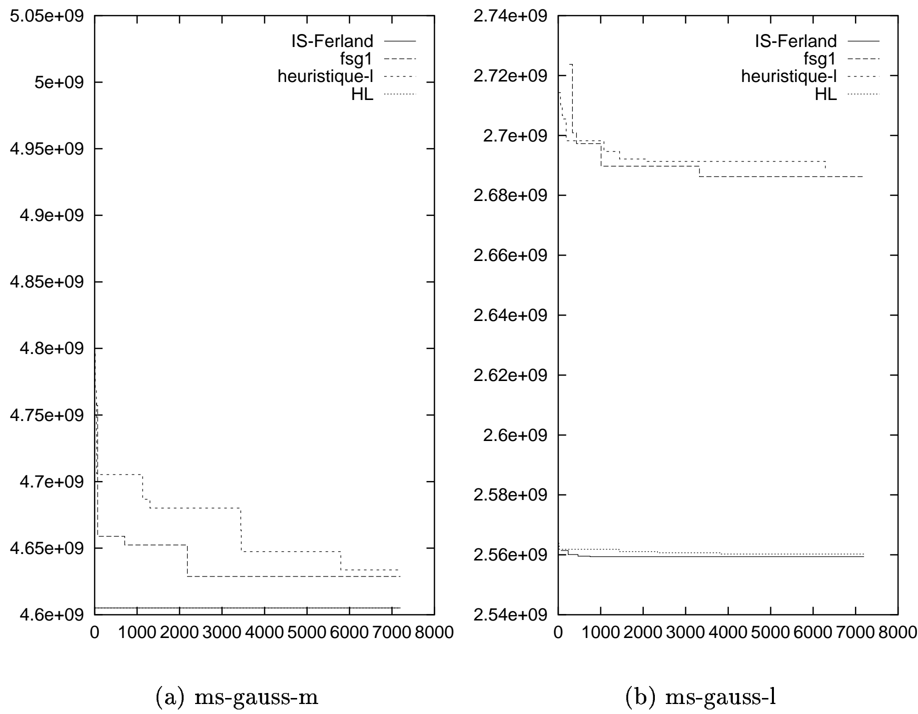
(a) fft2-b-m (b) fft2-b-l

Figure 8.16: Résultats: fft2-b



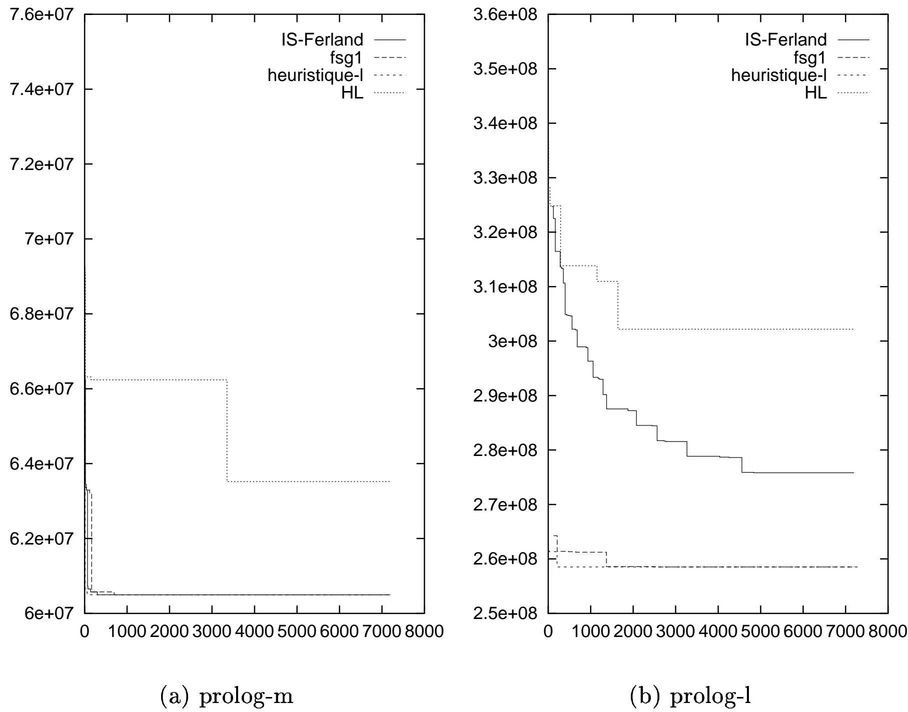
(a) gauss-m (b) gauss-l

Figure 8.17: Résultats: gauss-b



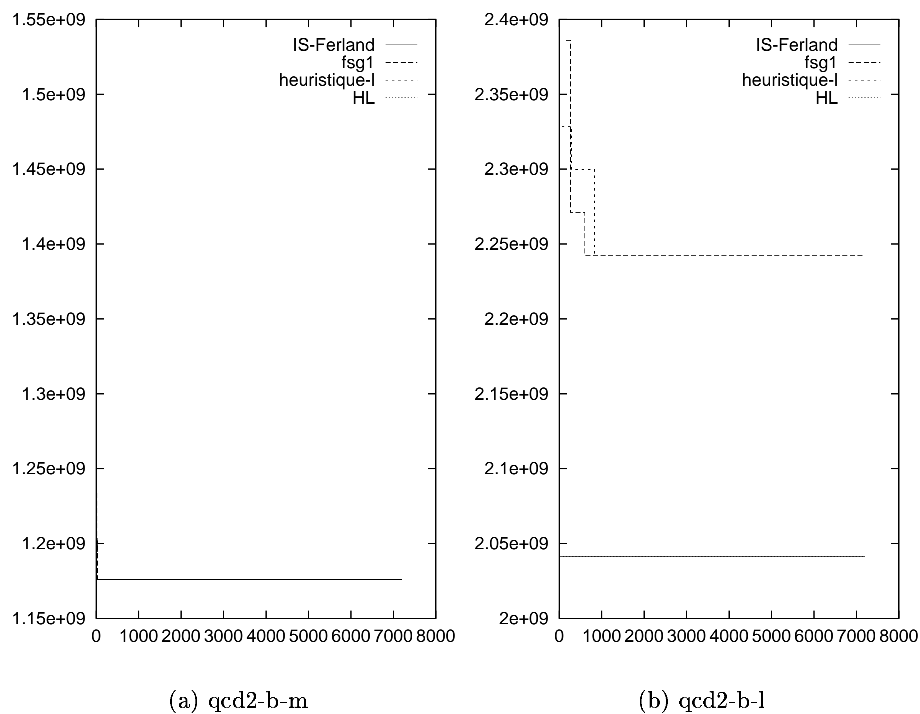
(a) ms-gauss-m (b) ms-gauss-l

Figure 8.18: Résultats: ms-gauss



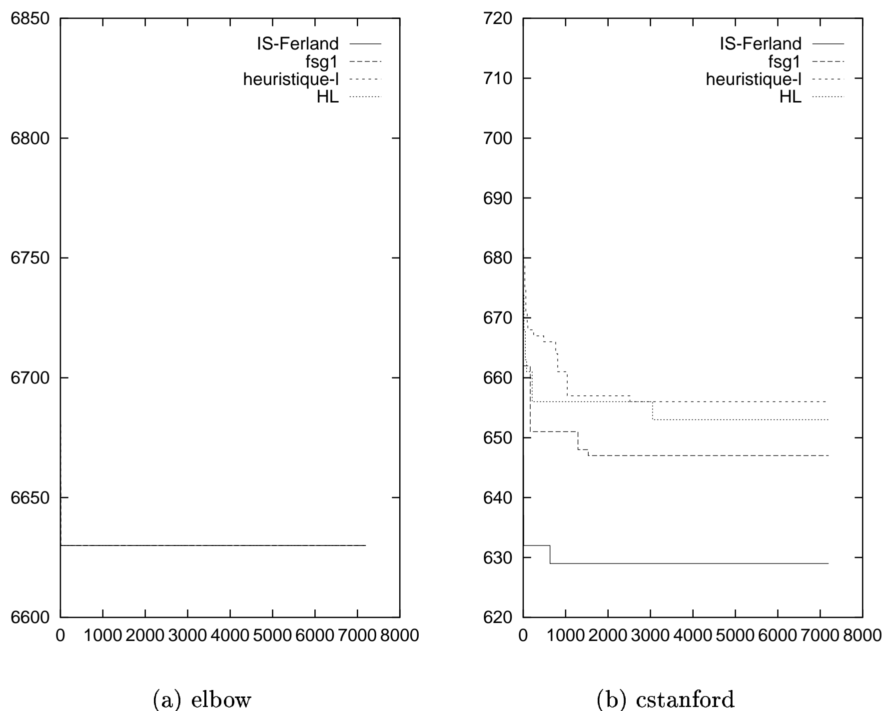
(a) prolog-m (b) prolog-l

Figure 8.19: Résultats: prolog



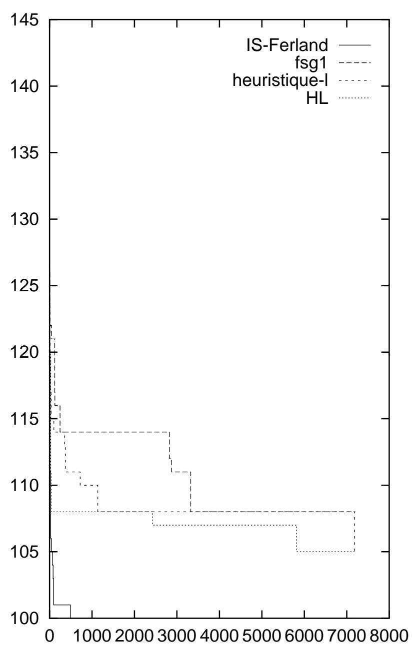
(a) qcd2-b-m (b) qcd2-b-l

Figure 8.20: Résultats: qcd2-b

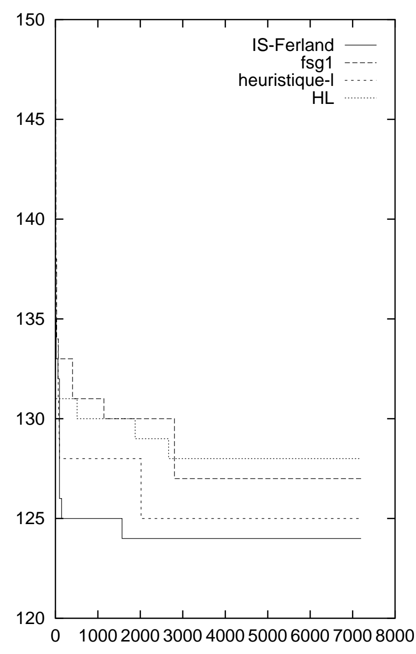


(a) elbow (b) cstanford

Figure 8.21: Résultats: elbow et cstanford

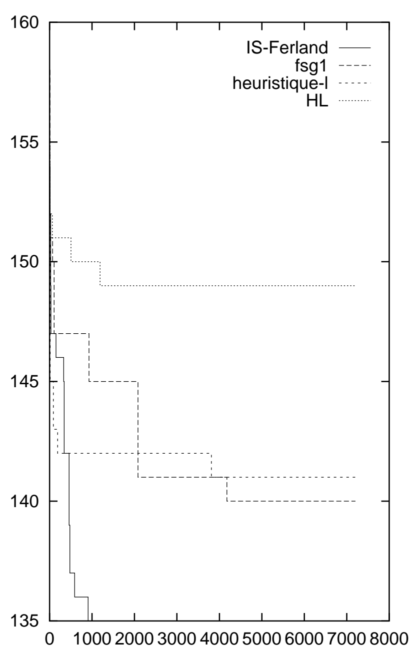


(a) ssc5

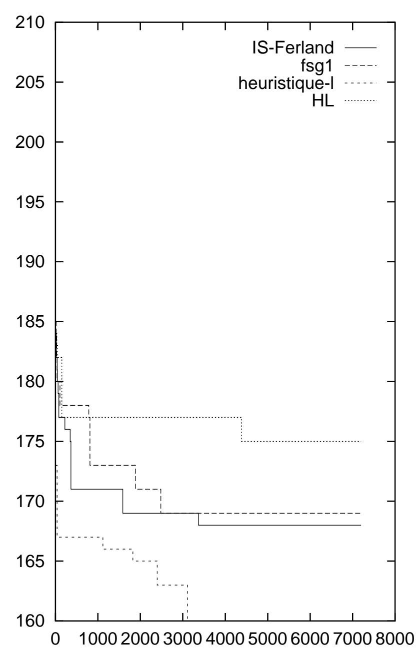


(b) ssc6

Figure 8.22: Résultats: ssc5 et 6



(a) ssc7



(b) ssc8

Figure 8.23: Résultats: ssc7 et 8

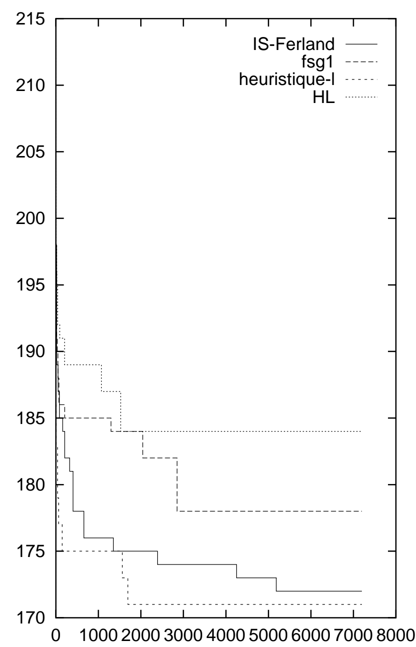


Figure 8.24: Résultat: ssc9

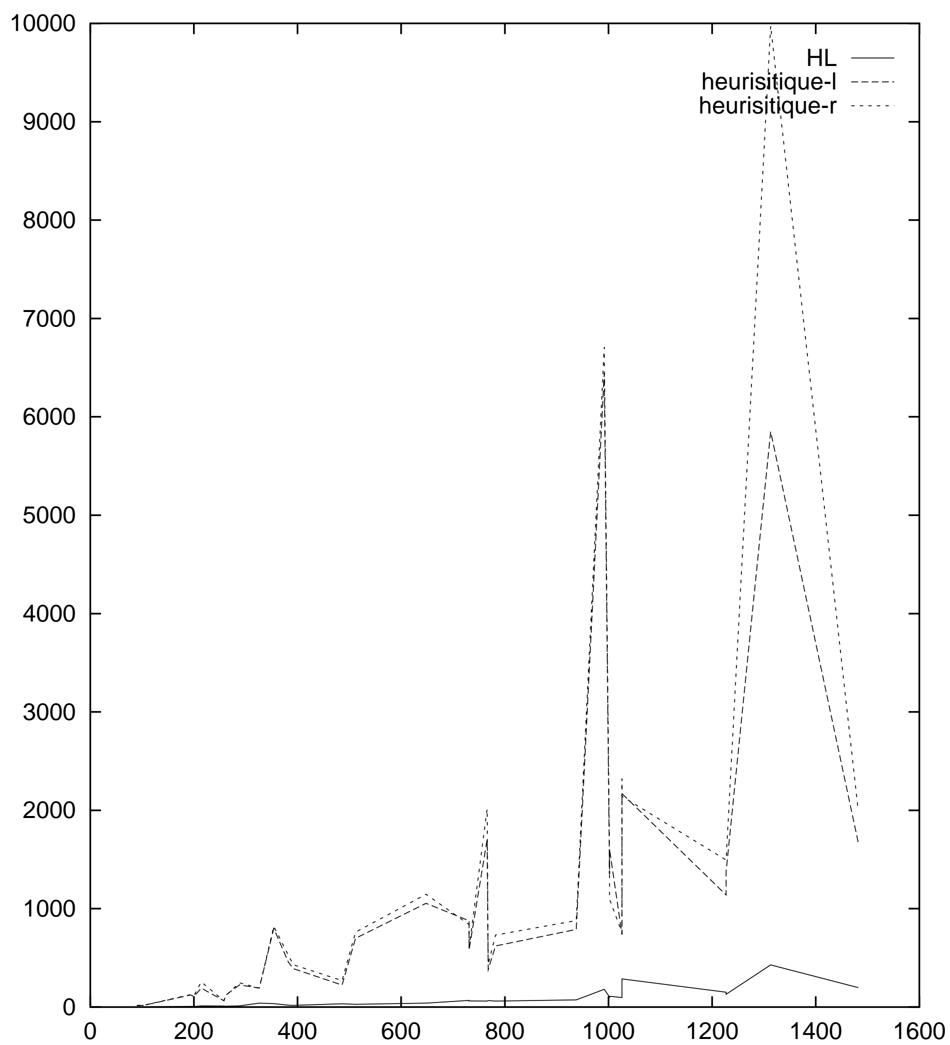
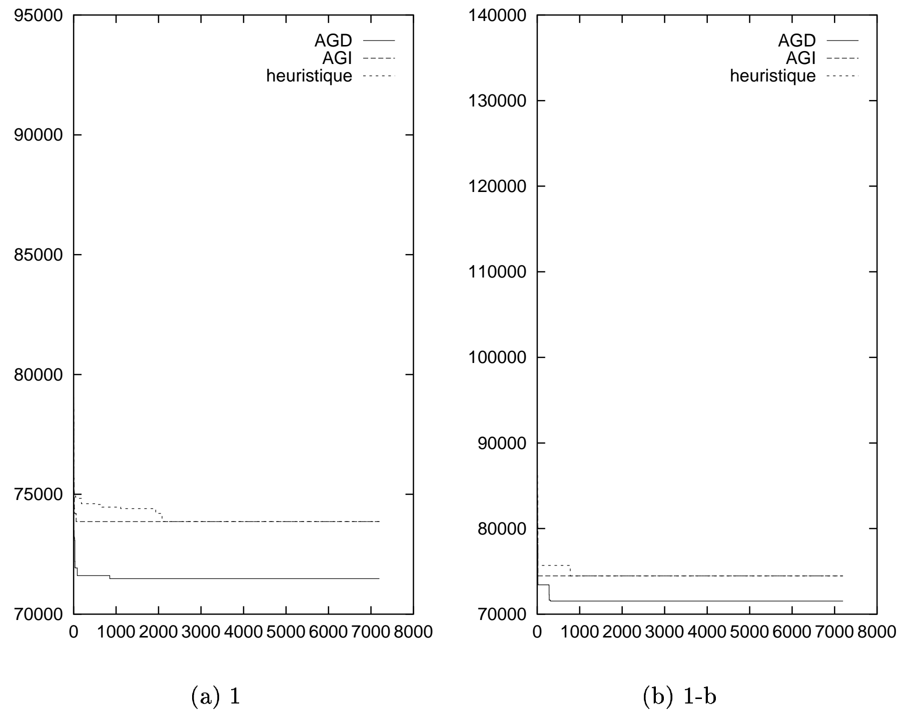


Figure 8.25: Temps des heuristiques suivant le nombre de tâches

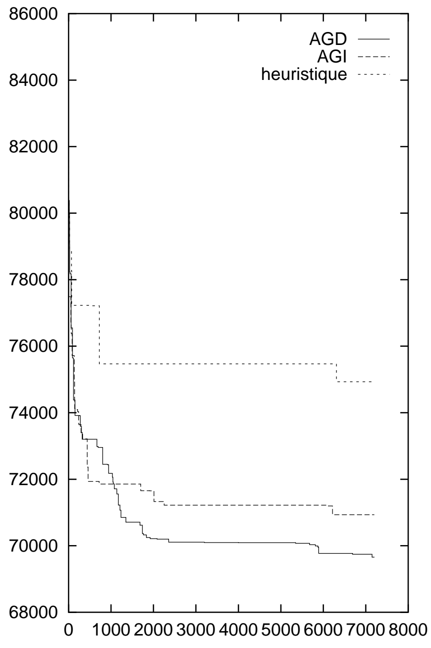
8.3 Résultats pour le problème du placement



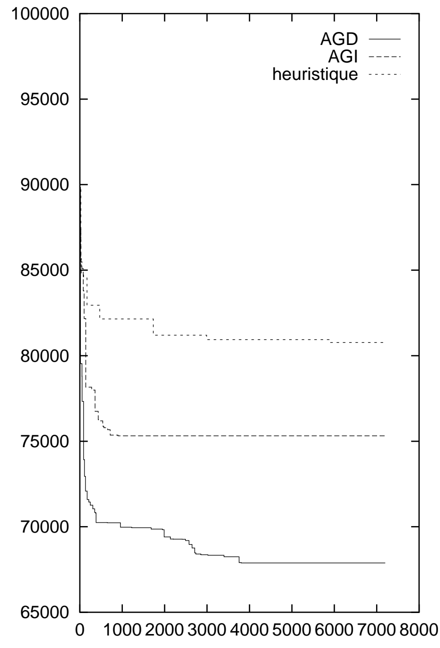
(a) 1

(b) 1-b

Figure 8.26: Résultats: 1

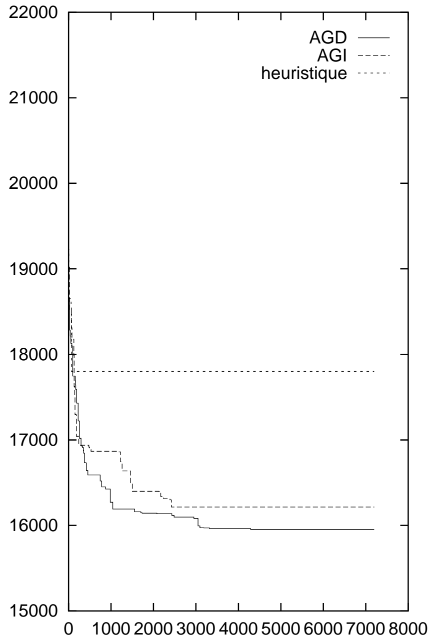


(a) 2

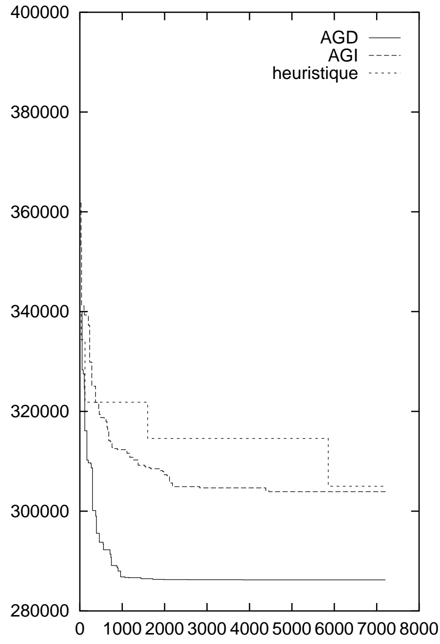


(b) 2-b

Figure 8.27: Résultats: 2

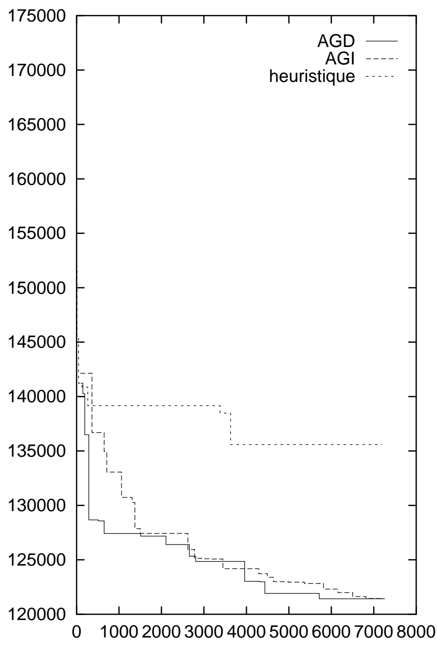


(a) 3

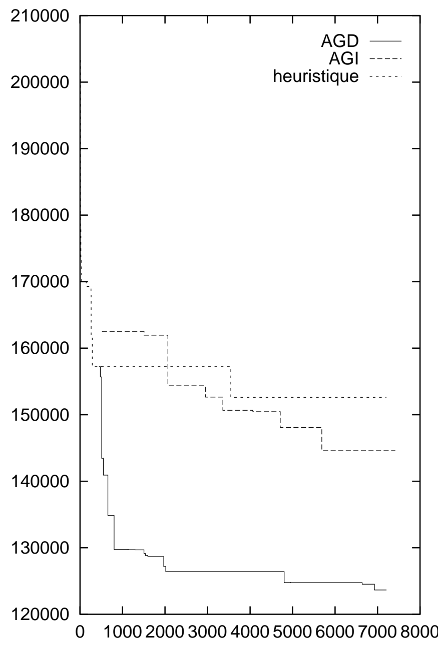


(b) 4

Figure 8.28: Résultats: 3 et 4

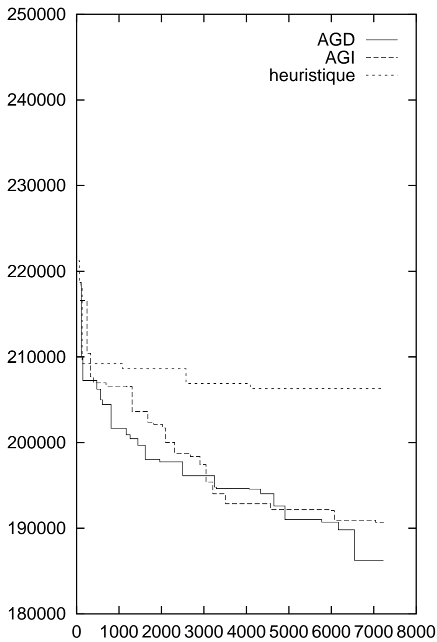


(a) 5

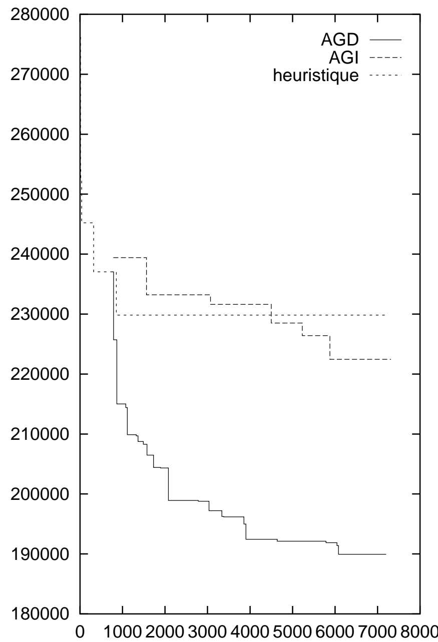


(b) 5-b

Figure 8.29: Résultats: 5

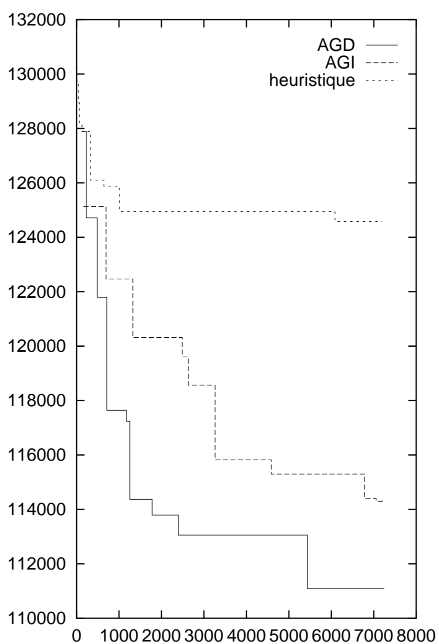


(a) 6

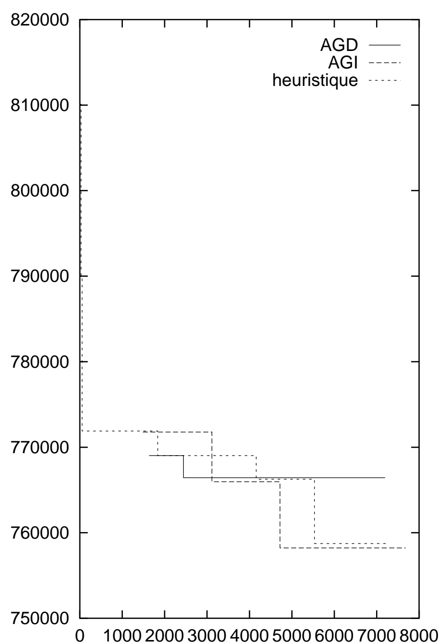


(b) 6-b

Figure 8.30: Résultats: 6

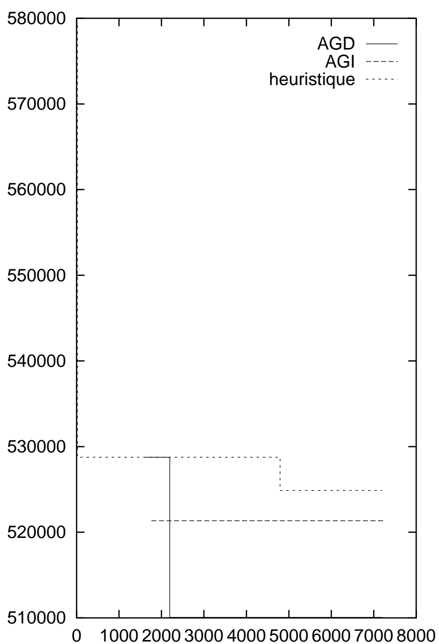


(a) 7

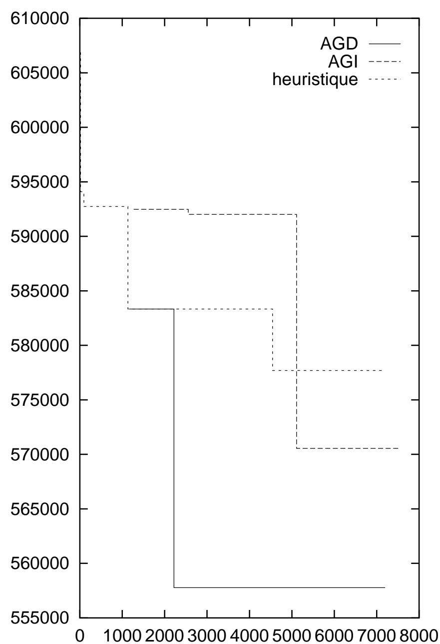


(b) 8

Figure 8.31: Résultats: 7 et 8

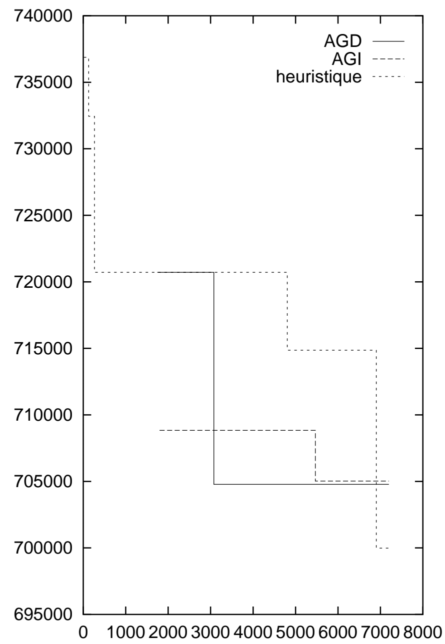


(a) 9



(b) 10

Figure 8.32: Résultats: 9 et 10



(a) 11

Figure 8.33: Résultat: 11

Bibliographie

[AD96] Ahmad (I.) et Dhodhi (M. K.). – Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, n° 22, 1996, pp. 395–406.

[Aze92] Azencottt (R.). – *Simulated Annealing: Parallelization techniques.* – Wiley-Interscience, 1992.

[BBC92] Beauquier (D.), Berstel (J.) et Chrétienne (P.). – *Eléments d'algorithmique.* – Masson, 1992.

[BCD+96] Benaïchouche (M.), Cung (V.-D.), Dowaji (S.), Cun (B. Le), Mautor (T.) et Roucairol (C.). – Building a parallel branch and bound library. *Dans : Solving Combinatorial Optimization Problems in Parallel*, éd. par Ferreira (A.) et Pardalos (P.), pp. 201–231. – Springer-Verlag, 1996.

[Ber73] Berge (C.). – *Graphes et hypergraphes.* – Bordas, 1973.

[Bou94] Bouvry (P.). – *Placement de tâches sur ordinateurs parallèles à mémoire distribuée.* – Thèse de Doctorat, LMC-IMAG, octobre 1994.

[Cau93] Caux (C.). – *Analyse et spécification de systèmes de production pour l'évaluation des performances et la recherche d'ordonnancement.* – Thèse de Doctorat, Université Blaise Pascal, 1993.

[CC88] Carlier (J.) et Chrétienne (P.). – *Problèmes d'ordonnancement.* – MASSON, 1988.

[CCLL95] Chrétienne (P.), Coffman (E. G. Jr), Lenstra (J. K.) et Liu (Z.). – *Scheduling Theory and its Applications.* – John Wiley and Sons, 1995.

[CDM92] Colorni (A.), Dorigo (M.) et Maniezzo (V.). – An investigation of some properties of an ant algorithm. *Dans : Proceedings of the Parallel Problem Solving from Nature Conference (PPSN 92).* pp. 509–520. – Elsevier Publishing, 1992.

[CF96] Corrêa (R.) et Ferreira (A.). – Parallel best-first branch-and-bound in discrete optimization: A framework. *Dans : Solving Combinatorial Optimization Problems in Parallel*, éd. par Ferreira (A.) et Pardalos (P.), pp. 171–200. – Springer-Verlag, 1996.

[CFR96] Corrêa (R.), Ferreira (A.) et Rebreyend (P.). – Integrating list heuristic into genetic algorithms for multiprocessor scheduling. *Dans : Eighth IEEE Symposium on Parallel and Distributed Processing*. pp. 462–469. – New-Orleans, USA, octobre 1996. A paraitre: Transaction on Parallel and Distributed System, version complète disponible à: http: //www.nce.ufrj.br/aep/reltec.

[CGK+97] Calégari (P.), Guidec (F.), Kuonen (P.), Chamaret (B.), Ubéda (S.), Josselin (S.), Wagner (D.) et Pizarosso (M.). – Radio network planning with combinatorial optimisation algorithms. – Projet Européen STORMS, colloque ACTS, Espagne, janvier 1997.

[Cha99] Chamaret (B.). – *Outils de planification pour les réseaux cellulaires*. – Thèse de Doctorat, Université de Saint-Etienne, janvier 1999.

[CJK+97] Chamaret (B.), Josselin (S.), Kuonen (P.), Pizarroso (M.), Salas-Manzanedo (B.), Ubéda (S.) et Wagner (D.). – Radio network optimization with stable set search. *Dans : 47th Vehicular Technology Conference, VTC'97, IEEE VTS*. – mai 1997.

[CJUW97] Chamaret (B.), Josselin (S), Ubéda (S.) et Wagner (D.). – Adaptive bts location in cellular network. *Dans : Parallel and Distributed Computing and Networks, PDCN'97, IASTED, Singapour*, pp. 287–291. – août 1997.

[CM91] Chassery (J.M.) et Montavert (A.). – *Géométrie discrète en analyse d'image*. – Paris, Editions Hermès, 1991.

[Cor97] Corrêa (R.). – *Recherche Arborescente Parallèle: de la Formulation Algorithmique aux Applications*. – Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1997.

[CP95] Cantú-Paz (E.). – *A Summary of Research on Parallel Genetic Algorithm*. – Rapport technique n° 95007, Illinois Genetic Algorithms Laboratory, juillet 1995.

[CRP96] Caux (C.), Rebreyend (P.) et Pierreval (H.). – No-wait jobshop scheduling using simulated annealing. *Dans : Proceedings of the 12th International Conference on CAD/CAM Robotics and Factories of the Future, 1996*. Middlesex University, pp. 633–638. – août 1996.

[CT91] Clausen (J.) et Traff (J.). – Implementation of parallel branch-and-bound algorithmes - experiences with the graph partitioning problem. *Annals of Operations Research*, n° 33, 1991, pp. 331–349.

[CT94] Clausen (J.) et Traff (J.). – Do inherently sequential branch-and-bound algorithms exist ? *Parallel Processing Letters*, vol. 4, n° 1 and 2, 3-13, 1994.

[CUZ96] Chamaret (B.), Ubéda (S.) et Zerovnik (J.). – A randomized algorithm for graph coloring applied to channel allocation in mobile telephone networks. *Dans : Proceeding of the 6th International Conference on Operational Research, KOI'96, Croatie.*, éd. par Hunjak (T.), Martic (Lj.) et Neralic (L.), pp. 25–30. – 1996.

[DAS95] Dhodhi (M. K.), Ahmad (I.) et Storer (R.). – Shemus: synthesis of heterogeneous multiprocessor systems. *Microprocessors and Microsystems*, vol. 19, n° 6, 1995, pp. 311–319.

[DG97] Dorigo (M.) et Gambardella (L. M.). – Ant colonies for the traveling salesman problem. *BioSystems*, 1997. – sous presse, disponible comme rapport technique, TR/IRIDIA/1996-3, IRIDIA, Université Libre de Bruxelles, ftp://iridia.ulb.ac.be/pub/dorigo/journals/IJ.15-BIOSYS97.ps.gz.

[DMC96] Dorigo (M.), Maniezo (V.) et Colorni (A.). – The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, vol. 26, n° 1, 1996, pp. 1–13.

[DT96] Djordjevic (G. L.) et Tosic (M. B.). – A heuristic for scheduling task graphs with communication delays onto multiprocessors. *Parallel Computing*, n° 22, 1996, pp. 1197–1214.

[Dub96] Duboux (T.). – *Régulation dynamique du partitionnement de données sur machines parallèles à mémoire distribuée.* – Thèse de Doctorat, Ecole Normale Supérieure de Lyon, janvier 1996.

[FF94] Fleurent (C.) et Ferland (J.A.). – *Genetic algorithms and hybrids for graphs colorings.* – Rapport technique n° IRO-922, Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, septembre 1994.

[FM97] Floriani (L.C.P.) et Mateus (G.R.). – Optimization models for effective cell planning design. *Dans : Proceeding of Mobicom'97 conference.* – 1997.

[FP96] Ferreira (A.) et Pardalos (P.). – *Solving Combinatorial Optimization Problems in Parallel.* – LNCS No 1054, 1996, *Springer-Verlag.*

[FR95] Feo (T.A) et Resende (M.G.C). – Greedy randomized adaptive search procedures. *Journal of Global Optimization*, n° 6, 1995, pp. 109–133.

[Gég96] Gégout (C.). – *Techniques Evolutionnaires pour l'Apprentissage des Réseaux de Neurones à Coefficients Réels.* – Thèse de Doctorat, Ecole Normale Supérieure de Lyon, décembre 1996.

[GGRG85] Grefenstette (J. J.), Gopal (R.), Rosmaita (B. J.) et Gucht (D. V.). – Genetic algorithms for the traveling salesman problem. *Dans : Proceedings of the 1st International Conference on*

Genetic Algorithms and their Applications, éd. par Grefenstette (John J.). pp. 160–168. – Pittsburgh, PA, juillet 1985.

[GJ79] Garey (M. R.) et Johnson (D. S.). – *Computer and intractability A Guide to the theory of NP-Completeness.* – Freeman, 1979.

[Glo89] Glover (F.). – Tabu search–part I. *ORSA Journal on Computing*, vol. 1, n° 3, 1989, pp. 190–206.

[Glo90] Glover (F.). – Tabu search–part II. *ORSA Journal on Computing*, vol. 2, n° 1, 1990, pp. 4–32.

[GM90] Gondran (M.) et Minoux (M.). – *Graphes et algorithmes.* – Eyrolles, octobre 1990.

[Gol89] Goldberg (D. E.). – *Genetic Algorithms in Search, Optimization, and Machine Learning.* – Addison Wesley, 1989.

[HA92] Hedge (S. U.) et Ahsmore (B.). – A feasibility study of genetic placement. *Technical Journal*, novembre-décembre 1992, pp. 72–82.

[HAR94] Hou (E. S.H.), Ansari (N.) et Ren (H.). – A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributes Systems*, vol. 5, n° 2, février 1994, pp. 113–120.

[HB96] Heitkötter (J.) et Beasley (D.). – The hitch-hiker's guide to evolutionary computation. – http: //research.de.uu.net: 8080/encore/, mars 1996.

[Hol92] Holland (J. E.). – *Adaptation in Natural and Artificial Systems.* – MIT Press, 1992.

[IL97] Ibbetson (L.J.) et Lopes (L.B.). – *An automatic bse site placement algorithm.* – Rapport technique, Université de Leeds, Angleterre, 1997.

[Ind96] Indaco (P.). – *PCB Final Placement by Mixed Integer Linear Optimisation.* – Rapport de stage, Faculty of Science, The University of Liverpool, décembre 1996.

[Joh85] Johnson (D.S.). – The np-completeness column : an ongoing guide. *Journal of algorithms*, vol. 6, n° 1, 1985, pp. 434–451.

[KA97] Knopman (J.) et Aude (J. S.). – Parallel simulated annealing: An adaptive approach. *Dans : IPPS: 11th International Parallel Processing Symposium.* – IEEE Computer Society Press, 1997.

[KGV83] Kirkpatrick (S.), Gelatt (C.D.) et Vecchi (M.P.). – Optimization by simulated annealing. *Science*, vol. 220, mai 1983, pp. 671–680.

[Kit94] Kitajima (J.). – *Modèles Quantitatifs d'Algorithmes Parallèles.* – Thèse de Doctorat, LMC-IMAG, octobre 1994.

[KK96] Karypis (G.) et Kumar (V.). – *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* – Rapport technique n° 95–035, Département d'Informatique, Université du Minnesota, 1996. http: //www.cs.umn.edu/ karypis.

[KN84] Kasahara (H.) et Narita (S.). – Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, vol. C-33, n° 11, novembre 1984, pp. 1023–1029.

[KN85] Kasahara (H.) et Narita (S.). – Parallel processing of robot-arm control computation on a multiprocessor system. *IEEE Journal of Robotics and Automation*, vol. RA-1, n° 2, 1985, pp. 104–113.

[Koz92] Koza (J. R.). – *Genetic Programming: On the Programming by Means of Natural Selection.* – MIT press, 1992.

[KP94] Kitajima (J.) et Plateau (B.). – Modelling parallel program behaviour in ALPES. *Information and Software Technology*, vol. 36, n° 7, 1994, pp. 457–464.

[KTP93] Kitajima (J.), Tron (C.) et Plateau (B.). – Alpes: A tool for the performance evalutation of parallel programs. *Dans : Environments and Tools for Parallel Scientific Computing*, éd. par Dongarra (J. J.) et Tourancheau (B.), pp. 213–228. – North-Holland, 1993.

[Lou93] Louis (S. J.). – *Genetic Algorithms as a Computational Tool for Design.* – Thèse de Doctorat, Département d'informatique, Université d'Indiana, août 1993.

[MKTM94] McCreary (C.L.), Khan (A.A.), Thompson (J.J.) et McArdle (M.E.). – A comparison of heuristics for scheduling DAGS on multiprocessors. *Dans : Proceedings of 8th International Parallel Processing Symposium.* pp. 446–451. – IEEE, 1994.

[NF97] Niar (S.) et Freville (A.). – A parallel tabu search algorithm for the 0-1 multidimensional knapsack problem. *Dans : IPPS: 11th International Parallel Processing Symposium.* – IEEE Computer Society Press, 1997.

[OTT91] Onorada (H.), Taniguchi (Y.) et Tamaru (K.). – Branch-and-bound placement for building block layout. *Dans : 28th ACM/IEEE Design Automation Conference, 1991*, pp. 433–439. – 1991.

[PC94a] Park (K.) et Carter (B.). – *On the effectiveness of genetic search in combinatorial optimization.* – Rapport technique n° BU-CS-94-010, Département d'informatique, Université de Boston, 1994.

[PC94b] Perregaard (M.) et Clausen (J.). – *Parallel Branch-and-Bound Methods for the Job Shop Scheduling problem.* – Rapport technique, Université de Copenhague, rapport DIKU 94/35, 1994.

[Pel95] Pellegrini (F.). – *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme.* – Thèse de Doctorat, Ecole doctorale de mathématiques et d'informatiques, Université Bordeaux I, janvier 1995.

[Pel96] Pellegrini (F.). – *Scotch 3.1 User's Guide.* – LaBRI, TALENCE, FRANCE, août 1996.

[PPMR95] Pardalos (P. M.), Pitsoulis (L.), Mavridou (T.) et Resende (M. G.C.). – Parallel search for combinatorial optimization: Genetic algorithmes, simulated annealing, tabu search and grasp. *Dans : Parallel Algorithms For Irregularly Structured Problems*, éd. par Ferreira (A.) et Rolim (J.). pp. 317–331. – Springer-Verlag, 1995, août 1995.

[PR91] Padberg (M.) et Rinaldi (G.). – A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, vol. 33, n° 1, mars 1991, pp. 60–100.

[PW89] Petford (A.D.) et Welsh (D.J.A.). – A randomized 3-coloring algorithm. *Discrete Mathematics*, vol. 74, 1989, pp. 253–261.

[QPPW98] Quagliarella (D.), Périaux (J.), Poloni (C.) et Winter (G.). – *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science.* – Wiley, 1998.

[RB94] Renders (J-M.) et Bersini (H.). – Hybridizing genetic algorithms with hill-climbing methods for global optimization: Two possible ways. *Dans : Proceedings of the First IEEE Conf. On Evolutionary Programming, 1994*, pp. 312–317. – 1994.

[RC97] Renaud (D.) et Caminada (A.). – Evolutionary methods and operators for frequency assignment problem. *Speedup Journal*, vol. 11, n° 2, novembre 1997, pp. 27–32.

[RC98] Reininger (P.) et Caminada (A.). – Model for gsm radio network optimisation. *Dans : DIAL M 98 Workshop, Dallas, US.* – octobre 1998.

[Reb95] Rebreyend (P.). – *Ordonnancement dans un atelier sans temps d'attente: problème du no-wait jobshop.* – Rapport de stage, Ecole Normale Supérieure de Lyon, juin 1995.

[RSM98] Rebreyend (P.), Sandnes (F.E) et Megson (G.M.). – *Static Multiprocessor Task Graph Scheduling in the Genetic Paradigm: A Comparison of Genotype Representations.* – Rapport de recherche n° 98-25, 46 allée d'Italie, F-69364 Lyon Cedex 07, France, LIP-ENS-Lyon, mai 1998.

[SM91] Shahookar (K.) et Mazumder (P.). – VLSI cell placement techniques. *ACM Computing Surveys*, vol. 23, n° 2, juin 1991, pp. 143–220.

[SM96] Sandnes (F.) et Megson (G.). – A hybrid genetic algorithm applied to automatic parallel controller code generation. *Dans : IEEE Proceedings of the 8'th Euromicro Workshop on Real-Time Systems, 1996*, pp. 70–75. – 1996.

[SP96] Sherali (H.D) et Pendyala (C.M.). – Optimisation location of transmitters for micro-cellular radio communication system design. *Journal on Selected Areas in Communicatins*, vol. 14, n° 4, mai 1996, pp. 662–672.

[SSR91] Sutanhavibul (S.), Shragowitw (E.) et Roser (J. B.). – An analytical approach to floorplan design and optimization. *IEEE Transactions on Computer-Aided Design*, vol. 10, n° 6, juin 1991, pp. 761–769.

[Swa93] Swartz (W. P.). – *Automatic Layout of Analog and Digital Mixed Macro/Standard cell Integrated Circuits.* – Thèse de Doctorat, Université de Yale, 1993.

[Tab97] Tabbane (S). – *Réseaux mobiles.* – Hermès, 1997.

[TLM95] Tschöke (S.), Lüling (R.) et Monien (B.). – Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. *Dans : 9th International Parallel Processing Symposium (IPPS 95)*, pp. 182–189. – avril 1995.

[Tur96] Turrini (S.). – *Optimization and Placement with the Genetic Workbench.* – Rapport technique, Western Research Laboratory, décembre 1996. http: //www.research.digital.com/wrl/home.html.

[VL90] Von Laszewski (G.). – A parallel genetic algorithm for the graph partitioning problem. *Transputer Research and Applications*, vol. 4, 1990, pp. 164–172.

[VL91] Von Laszewski (G.). – Intelligent structural operators for the k-way graph partitioning problem. *Dans : 4th International Conference on Genetic Algorithms, 1991*, pp. 45–52. – 1991.

[VL93] Von Laszewski (G.). – *A Collection of Graph Partitioning Algorithms: Simualted Annealing, Simulated Tempering, Kernighan Lin, Two Optimal, Graph Reduction, Bisection.* – Rapport technique, Northeast Parallel Architectures Center at Syracuse University, mai 1993.

[Wal96] Wall (M.). – *GAlib: A C++ Library of Genetic ALgorithm Components.* – Mechanical Engineering Department, Massachusetts Institute of Technology, août 1996. Version 2.4, http: //lancet.mit.edu/ga/ galib-request@mit.edu.

[Wil89] Wilf (H. S.). – *Algorithmes et complexité.* – Masson, 1989.

[WK95] Wang (P.C.) et Korfhage (W.). – Process scheduling using genetic algorithms. *Dans : Symposium on Parallel and Distributed System '95.* pp. 638–643. – Los Alamitos, Ca., USA, octobre 1995.

[WLL88] Wong (D.F.), Leong (H.W.) et Liu (C.L.). – *Simulated Annealing for VLSI Design.* – Kluwer Academic Publishers, 1988.

[YG92] Yang (Tao) et Gerasoulis (Apostolos). – Pyrros: Static task scheduling and code generation for message passing multiprocessors. *Dans : Proceeding of the 6th ACM International Conference on Supercomputing*, pp. 428–437. – juillet 1992.

[YG93] Yang (T.) et Gerasoulis (A.). – List scheduling with and without communication delays. *Parallel Computing*, vol. 19, n° 7, 1993, pp. 1321–1344.

Publications personnelles:

Conférences internationales avec comité de lecture:

[1] Corrêa (R.), Ferreira (A.) et Rebreyend (P.). Integrating List Heuristic into Genetic Algorithm for Multiprocessor Scheduling. *Dans: Eight IEEE Symposium on Parallel and Distributed Processing*, pp. 462–469, Nouvelle-Orléans, USA, octobre 1996.

[2] Chamaret (B.), Rebreyend (P.) et Sandnes (F.). Static Multiprocessor Task Graph Scheduling: A comparison on several hybrids Genetic Algorithms. *Dans: Conference on Parallel and Distributed Computing and Networks.* pp. 210–213, Brisbane, Australie, décembre 1998.

Conférence nationale avec comité de lecture:

[3] Corrêa (R.), Ferreira (A.) et Rebreyend (P.). Un algorithme génétique d'ordonnancement de tâches. *Dans: RenPar'8.* pp. 45–48, mai 1996, Bordeaux.

Rapports de recherche:

[4] Corrêa (R.), Ferreira (A.) et Rebreyend (P.). *Static Multiprocessor Tasks with Genetic Algorithms.* Rapport de recherche NCE-02/96, Núcleo de Computaçãp Electrônica, Universadade Federal Do Rio de Janeiro, Brésil, 1996

[5] Rebreyend (P.), Sandnes (F.) and Megson (G.). *Static Multiprocessor Tasks Graph Scheduling in the Genetic Paradigm: A comparison of Genotype representations.* Rapport de recherche RR 1998-25, LIP, ENS-Lyon, France, mai 1998.

Revue Internationale

[6] Corrêa (R.), Ferreira (A.) et Rebreyend (P.). Scheduling Multiprocesor Tasks with Genetic Algorithms. *IEEE Transactions on Parallel and Distributed Systems.* A paraître.