UNIVERSITÉ DE MONTRÉAL

MÉTAHEURISTIQUES APPLIQUÉES AU PROBLÈME DE COVERING DESIGN

KAMAL FADLAOUI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE
MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2009

© Kamal Fadlaoui, 2009.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTAHEURISTIQUES APPLIQUÉES AU PROBLÈME DE COVERING DESIGN

présenté par : FADLAOUI Kamal.
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. MULLINS John, Ph.D., président.
M. GALINIER Philippe, Doct., membre et directeur de recherche.
M. QUINTERO Alejandro, Doct., membre.

Remerciements

Merci à mon directeur de recherche, Philippe Galinier, pour son grand intérêt pour mes travaux et son suivi régulier.
Merci à ma famille et mes amis pour leur soutien.
Merci à mes compagnons de travail, Wolfgang Amadeus, Ludwig et Piotr Ilitch.

Résumé

Un (v, k, t)-covering design est un ensemble de blocs (sous ensemble à k éléments d'un ensemble de référence $\mathcal{V}$ à v éléments) tel que tout sous-ensemble à t éléments de $\mathcal{V}$ soit contenu dans un des blocs. Considérant v, k et t, le problème de Covering Design consiste à trouver un covering contenant le moins de blocs possible. Pour résoudre le problème, nous avons adapté des métaheuristiques au Covering Design. Nous avons en particulier conçu un algorithme tabou avec diversification et un algorithme mémétique. Afin de rendre ces algorithmes plus rapides, nous les avons munis de nouvelles structures de données totalement incrémentales. Nos algorithmes de bas niveau sont devenus ainsi plus rapides et moins gourmands en espace mémoire. Nos algorithmes ont donc été capables de traiter des jeux de données que les précédentes métaheuristiques développées ne pouvaient pas tester. En matière de vitesse, nos algorithmes sont entre 10 et 100 fois plus rapides et l'accélération est d'autant plus élevée que les jeux de données à traiter sont gros. Nous avons testé nos algorithmes sur plus de 700 jeux de données. Nous avons ainsi réussi à trouver un meilleur covering design pour 77 jeux de données, dont 71 n'ont pas été améliorés par la suite.

Abstract

A (v, k, t)-covering design is a collection of k-subsets (called blocks) of a v-set $\mathcal{V}$ such that every t-subset of $\mathcal{V}$ is contained in at least one block. Given v, k and t, the goal of the Covering Design problem is to find a covering made of a minimum number of blocks. In this paper, we present a new tabu algorithm with a mechanism of diversification and a new memetic algorithm for the solution of the problem. Our algorithms use a new implementation totally incremental designed in order to evaluate efficiently the performance of the neighbors of the current configuration. The new implementation is much less space-consuming than the currently used technique, making it possible to tackle much larger problem instances. It is also significantly faster (between 10 and 100 times faster) and the speeding rate gets higher and higher as the size of the instances raises. We mesured the performance of our tabu algorithm trying more than 700 problem instances. Thanks to the improved data structures, our tabu algorithm was able to improve the upper bound of 77 problem instances and still hold the record for 71 of them.

Table des matières

Remerciements iii

Résumé iv

Abstract v

Table des matières vi

Liste des tableaux ix

Liste des figures x

Liste des annexes xii

Liste des sigles et abréviations xiii

Chapitre 1 INTRODUCTION 1
- 1.1 Définitions 1
- 1.2 Éléments de la problématique 3
- 1.3 Objectifs de recherche 5
- 1.4 Plan du mémoire 6

Chapitre 2 REVUE DE LITTÉRATURE 7
- 2.1 Problèmes apparentés et applications 7
  - 2.1.1 Problèmes apparentés au Covering Design 7
  - 2.1.2 Applications du problème de Covering Design 8
- 2.2 Présentation des heuristiques 9
  - 2.2.1 L'algorithme glouton 10
  - 2.2.2 La recherche locale et l'algorithme de descente 10
  - 2.2.3 L'algorithme de recuit simulé 11
  - 2.2.4 La recherche avec tabous 12
  - 2.2.5 L'algorithme mémétique 12
- 2.3 Les méthodes de résolutions existantes pour le Covering Design . . . 13

2.3.1 Les méthodes mathématiques 13
2.3.2 Les méthodes d'exploration arborescente 14
2.3.3 Les méthodes de construction de Gordon, Kuperberg et Patashnik 14
2.3.4 Les méthodes heuristiques 16

Chapitre 3 TECHNIQUES PROPOSÉES 23
3.1 Algorithmes de haut niveau 23
3.1.1 Algorithme tabou 24
3.1.2 Algorithme mémétique 29
3.1.3 Résolution du problème d'optimisation 34
3.2 Algorithmes de bas niveau 35
3.2.1 Définition des structures de données 35
3.2.2 Formules de mise à jour de γ^+, γ^- et θ 36
3.2.3 Algorithmes de bas niveau 39
3.2.4 Illustration 40
3.2.5 Indiçage des t-subsets 46
3.2.6 Implémentation efficace d'ensembles d'éléments 46
3.2.7 Pseudo-code des algorithmes de mise à jour 48

Chapitre 4 RÉSULTATS EXPÉRIMENTAUX 51
4.1 Améliorations due aux structures de données 51
4.2 Résultats obtenus avec l'algorithme TS-CD 53
4.3 Expérimentations effectuées avec l'algorithme mémétique 55
4.3.1 Comparaison des différents croisements proposés 55
4.3.2 Réglages des paramètres avec le croisement X2 et comparaison avec l'algorithme tabou avec diversification 56
4.4 Analyse du comportement de l'algorithme 57
4.4.1 Profil d'exécution 57
4.4.2 Observation du phénomène de stagnation 58
4.4.3 Identification du phénomène d'hyperstagnation avec chute . . 61
4.4.4 Identification du phénomène d'hyperstagnation avec descente lente 63
4.4.5 Phénomène d'hyperstagnation pure 64
4.4.6 Comportement de la procédure de haut niveau qui résout le problème d'optimisation 65

Chapitre 5 CONCLUSION 67
5.1 Synthèse des travaux 67
5.2 Principales contributions 67
5.3 Limitations et perspectives 68

Références 69

Annexes 72

Liste des tableaux

TABLEAU 3.1 Contributions aux matrices γ et θ selon les différents cas de figures 38
TABLEAU 3.2 Modification des matrices γ et θ selon les différents cas de figures (cas 1 à 4) 39
TABLEAU 3.3 Modification des matrices γ et θ selon les différents cas de figures (cas 5 à 7) 40
TABLEAU 3.4 Situation des blocs par rapport au t-subset T selon les différents cas énoncés 43
TABLEAU 4.1 Comparaison entre deux implémentations : celle de Nurmela (Impl.1) et la nôtre (Impl.2) pour différents jeux de données . 53
TABLEAU 4.2 Nombre de solutions trouvées par chacun des croisement pour 10 exécutions sur différents jeux de données 56
TABLEAU 4.3 Tests avec le croisement X2 pour différents jeux de paramètres 57

Liste des figures

FIGURE 1.1 Exemple pour $v = 12$, $k = 6$ et $t = 2$ 2

FIGURE 3.1 Schéma représentant une période des variations de la liste tabou entre $lgtl0$ et $8lgtl0$ 27

FIGURE 3.2 Situations statiques : Ce schéma représente les éléments par des disques (par exemple l'élément x est représenté par un disque vert). Le t-subset T est représenté par un rectangle bleu. Le bloc B_i est représenté par un ovale vert dans la première situation et bleu dans la seconde. Remarquons qu'ici un élément est contenu dans un ensemble, si le disque qui représente l'élement est à l'intérieur de la figure représentant cet ensemble. Par exemple sur le schéma de gauche, l'élément x est contenu dans T car le disque qui le représente est contenu dans le rectangle bleu, mais n'est pas contenu dans B_i car le disque vert n'est pas à l'intérieur de l'ovale vert. 41

FIGURE 3.3 Cas 1 : Le bloc B_i en situation *Candidat* couvre le t-subset T grâce au mouvement 42

FIGURE 3.4 Cas 2 : Le bloc B_l quitte la situation *Rempart* suite à la couverture du t-subset T par le bloc B_i après le mouvement . . . 43

FIGURE 3.5 Cas 3 : Le bloc B_i en situation *Rempart* avant le mouvement passe en situation *Candidat* en découvrant le t-subset T . . . 44

FIGURE 3.6 Cas 4 : Le bloc B_l passe en situation *Rempart* suite à la découverture du t-subset T par B_i 44

FIGURE 3.7 Cas 5 : Le mouvement fait entrer le bloc B_i en situation *Candidat* par rapport au t-subset T 44

FIGURE 3.8 Cas 6 : Le mouvement fait sortir le bloc B_i de la situation *Candidat* par rapport au t-subset T 45

FIGURE 3.9 Cas 7 : Le mouvement modifie la situation *Candidat* du bloc B_i par rapport au t-subset T qui était candidat avec l'élément x avant le mouvement et qui est candidat avec l'élément y après 45

FIGURE 4.1 Exemple d'un profil typique d'un algorithme de recherche local pour un problème de satisfaction de contraintes 58
FIGURE 4.2 Profil d'exécution de l'algorithme tabou sans diversification . . 59
FIGURE 4.3 Profil d'exécution de l'algorithme tabou avec diversification . . 60
FIGURE 4.4 61
FIGURE 4.5 Hyperstagnation avec chute 62
FIGURE 4.6 Hyperstagnation avec descente lente 63
FIGURE 4.7 Hyperstagnation pure 64
FIGURE 4.8 Mise en évidence du mécanisme qui décrémente b en repartant de la configuration trouvée 66

Liste des annexes

Annexe 1 : Améliorations obtenues avec notre algorithme TS-CD 72

Liste des sigles et abréviations

CPU	Central Processing Unit
GRASP	Greedy Randomized Adaptive Search Procedure
MLTS	Multi Level Tabu Search

Chapitre 1

INTRODUCTION

Peut-on augmenter ses chances de gagner au Lotto grâce aux mathématiques ? La réponse immédiate est non bien sûr, mais il peut être intéressant de regarder le fonctionnement du jeu en détail. Au Québec, le Lotto 6/49 consiste en un tirage de 6 numéros parmi 49 toutes les semaines. On gagne la cagnotte lorsqu'on possède un ticket contenant les 6 bons numéros. On peut aussi gagner des sommes moins importantes si on possède 5, 4 ou 3 numéros gagnants. Il va de soi que pour gagner à coup sûr le gros lot il faut posséder tous les tickets (pour être certain d'avoir les 6 bons numéros). Qu'en est-il pour les lots moins gros ? Combien de tickets à 6 numéros (et lesquels) faut-il pour être sûr d'avoir par exemple 4 numéros gagnants ?

Nous nous intéressons ici au problème de Covering Design qui soulève la même question. En effet, on peut voir un ensemble de tickets, qui garantissent d'avoir un certain nombre de numéros gagnants (nombre que nous définissons à l'avance) sur l'un des tickets quelque soit le tirage effectué, comme un covering design selon une définition étendue. Notre objectif sera de trouver une méthode numérique performante pour trouver des covering designs. Nous chercherons en particulier à trouver des covering designs ayant le moins de blocs (voir la définition juste en dessous) possible.

1.1 Définitions

Dans le problème de Covering Design, on donne trois entiers v, k et t tels que $0 < t < k < v$. Soit $\mathcal{V} = \{1, \ldots, v\}$ un ensemble de référence contenant v éléments. On appelle **blocs** les sous-ensembles de $\mathcal{V}$ contenant k éléments et **t-subsets** les sous-ensembles de $\mathcal{V}$ contenant t éléments. Un (v, k, t)-covering design est un ensemble de blocs tel que tout t-subset est contenu dans au moins un bloc. Étant donnés v, k et t, le problème de Covering Design consiste à trouver un (v, k, t)-covering design contenant un nombre de blocs le plus petit possible.

Illustrons cela par un exemple. Considérons le cas où $v = 12$, $k = 6$ et $t = 2$. Nous pouvons par exemple former les ensembles V_1, V_2, V_3 et V_4 comme sur la figure 1.1, chacun contenant 3 éléments. On obtient une solution au problème en considérant par exemple les blocs formés par chacune des paires de l'ensemble $\{V_1, V_2, V_3, V_4\}$. Soit S la solution considérée, alors $S = \{V_1 \cup V_2, V_3 \cup V_4, V_1 \cup V_3, V_2 \cup V_4, V_1 \cup V_4, V_2 \cup V_3\}$. En effet, si on considère un t-subset T c'est-à-dire une paire d'éléments, alors on a :

- Soit les deux éléments appartiennent au même sous-ensemble V_i et le t-subset T est bien couvert par un bloc qui contient ce V_i (par exemple $(1, 2) \subset V_1 \subset V_1 \cup V_2$).
- Soit les deux éléments appartiennent à des sous-ensembles différents V_i et V_j avec $i \neq j$. Dans ce cas le t-subset T est couvert par le bloc $V_i \cup V_j$ (par exemple $(3, 4) \subset V_1 \cup V_2$ car $3 \in V_1$ et $4 \in V_2$).

Notons que la solution S contient 6 blocs et que c'est le mieux qu'on puisse faire ici : la solution est optimale.

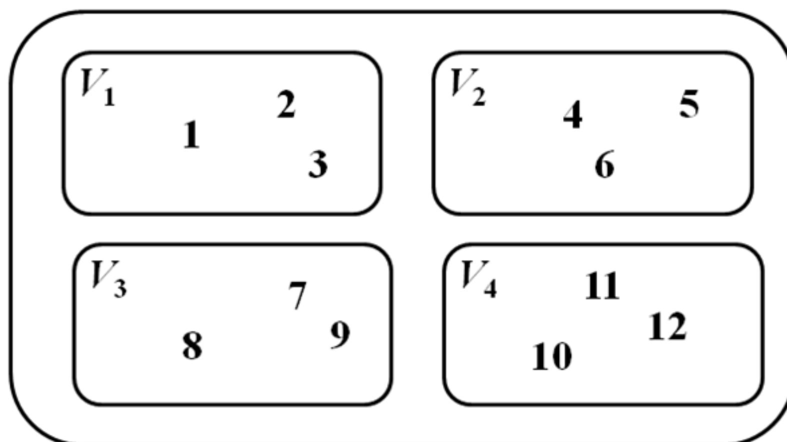

FIGURE 1.1 Exemple pour $v = 12$, $k = 6$ et $t = 2$

L'exemple que nous venons de présenter peut être qualifié de très petit (dans nos expériences, nous avons traité des exemplaires beaucoup plus grand, par exemple avec $v = 30$, $k = 15$ et $t = 5$). Mais nous pouvons déjà constater sur cet exemple à quel point la combinatoire de ce problème est grande. En effet, sur cet exemple il y a 66 t-subsets à couvrir et 924 blocs disponibles. Ainsi si on veut former une solution contenant 6 blocs, nous avons environ 6×10^{17} possibilités (donné par $\binom{\binom{v}{k}}{b}$) dont très peu forment une solution valide c'est-à-dire qui couvre tous les t-subsets. Pour des paramètres v, k et t donnés, il y a $\binom{v}{t}$ t-subsets à couvrir et on doit choisir parmi

$\binom{v}{k}$ blocs possibles pour former une solution : le nombre de t-subsets et de blocs augmente exponentiellement lorsque v augmente. De plus, il n'existe vraisemblablement pas d'algorithme polynomial pour résoudre le problème et il est irréaliste d'essayer toutes les possibilités. Nous avons cependant besoin de solutions, non nécessairement optimales, dans l'optique de pouvoir les utiliser dans des applications pratiques. Ce problème a en effet des applications dans le décodage par piégeage d'erreurs [7], la compression de données [13] et peut être utilisé pour les jeux de loteries [27] comme nous avons vu au début. Nous pouvons imaginer que les applications qui utilisent des covering designs cherchent à avoir de bonnes solutions, c'est-à-dire qui contiennent le moins de blocs possible.

1.2 Éléments de la problématique

Ce problème a beaucoup été étudié par le passé et suscite toujours de l'intérêt. De nombreux articles ont été écrits sur le sujet depuis environ 40 ans. Avant l'avènement des métaheuristiques et plus généralement des techniques informatiques, de nombreux mathématiciens s'évertuaient à chercher des résultats théoriques permettant de donner une borne supérieure ou une borne inférieure au nombre de blocs que pouvait contenir un covering design particulier [34] (ou une famille de covering designs définie par des paramètres spécifiques). Avec l'apparition des techniques numériques, une communauté beaucoup plus large s'est lancée dans la résolution du problème.

On trouve maintenant des archives sur internet contenant les meilleures solutions établies pour un grand nombre de jeux de données. Le site internet de Gordon : La Jolla Covering Repository [22] montre à quel point le problème suscite de l'intérêt parmi les mathématiciens et les informaticiens. Les contributeurs de ces archives se disputent en effet les meilleures solutions. Les résultats de La Jolla sont ainsi mis à jour quotidiennement et on y voit régulièrement des améliorations.

Notre objectif sera ici d'appliquer les métaheuristiques au problème. Les métaheuristiques sont une famille de techniques qui nous permettent, pour un problème d'optimisation difficile (typiquement NP-Difficile), d'obtenir de bonnes solutions rapidement à défaut de pouvoir obtenir des solutions optimales dans un temps raisonnable. Les métaheuristiques peuvent être classées selon différentes approches utilisées pour résoudre les problèmes. Nous avons des approches constructives (algorithme glouton, etc.), des approches de recherche locale (algorithme de descente, recuit simulé,

algorithme tabou, recherche à voisinage variable, etc.), des approches d'évolution (algorithme génétique, stratégie d'évolution, programmation évolutive, etc.), ainsi que des approches hybrides (algorithme mémétique, GRASP, colonies de fourmis, etc.).

Pour résoudre le problème de Covering Design, nous avons conçu un algorithme tabou. L'algorithme tabou est une technique de recherche locale. La recherche locale est un procédé qui permet d'explorer une suite de solutions afin de trouver la meilleure possible. On définit pour cela une notion de voisinage par laquelle chaque solution a un petit nombre de solutions qui sont ses voisines. L'algorithme peut alors parcourir l'espace des solutions en se déplaçant d'une solution à une de ses voisines selon certaines règles qui caractérisent l'algorithme.

Beaucoup de méthodes ont été proposées pour résoudre le problème de Covering Design. On trouve des méthodes s'appuyant sur la théorie de Turán, sur la géométrie projective, des méthodes d'exploration arborescente et des méthodes de construction. Les métaheuristiques ont également déjà été utilisées pour résoudre le problème de Covering Design. Il y a principalement deux métaheuristiques qui ont été proposées pour résoudre le problème. La première est un algorithme de recuit simulé développé par Nurmela et Östergård en 1993 [31]. La seconde est un algorithme multi-niveau coopératif avec opérateur tabou proposé par Dai et al. en 2005 [10].

En analysant ces deux heuristiques, il nous apparait que deux aspects sont particulièrement critiques lorsqu'on veut appliquer les métaheuristiques au problème de Covering Design. L'un apparait à bas niveau et l'autre à haut niveau. On appelle algorithmes de haut niveau les procédures qui déterminent de quelle façon se comporte l'algorithme pour résoudre le problème. Les algorithmes de haut niveau déterminent, par exemple, comment on choisit le prochain mouvement. Par opposition les algorithmes de bas niveau n'ont aucune influence sur le comportement de l'algorithme. Les algorithmes de bas niveau déterminent comment on effectue les sous-procédures (par exemple calculer le coût d'un voisin) et quelles sont les structures de données qu'on utilise. Elles ont seulement une influence sur la rapidité de l'algorithme.

Le premier aspect concerne les algorithmes de bas niveau. Nous cherchons à avoir des algorithmes **incrémentaux** pour implanter la recherche locale. Des algorithmes sont dits incrémentaux lorsque le calcul du coût d'une solution voisine est effectué en ne tenant compte que de ce qui a changé par rapport à la configuration courante. Ainsi pour les problèmes qui s'y prêtent bien, on peut gagner beaucoup en rapidité en évitant de recalculer complètement le coût d'une solution. Si on parvient à avoir en

temps constant le coût d'une configuration voisine, nous disons que les algorithmes sont **totalement incrémentaux**. Dans le problème de Covering Design, les algorithmes de bas niveau, et en particulier ceux qui permettent le calcul du coût d'une solution voisine, sont critiques. En effet, sans disposition particulière, la complexité en temps pour calculer le coût d'une solution est très élevée. Nurmela a développé des algorithmes incrémentaux pour son recuit simulé. Ces techniques ont même été reprises par Dai et al. Les résultats obtenus par Nurmela et par la suite par Dai ont été très bons. Cependant, il y a des limitations importantes quant à l'utilisation des algorithmes de bas niveau proposés par Nurmela. En effet, la taille mémoire des structures de données nécessaires à l'utilisation de ces algorithmes devient rapidement trop importante lorsque les paramètres v ou t deviennent trop grands. De plus, ces algorithmes même si relativement élaborés ne sont pas totalement incrémentaux. Nous avons donc pensé, à juste titre, qu'il était possible de faire mieux en concevant des algorithmes de bas niveau totalement incrémentaux. Les contraintes du problème ont une définition complexe (une contrainte correspond à la couverture d'un t-subset). C'est pourquoi il est complexe de concevoir des algorithmes simplement incrémentaux et encore plus ardu de créer des algorithmes totalement incrémentaux.

Le second aspect concerne un phénomène de stagnation de la recherche. Nous disons que la recherche stagne lorsque le coût des configurations ne s'améliore plus bien qu'une solution à coût nul existe. Ce phénomène a été observé par Dai et al. [10]. Afin d'empêcher ce phénomène, Dai et al. ont proposé un mécanisme multi-niveau. Nous n'avons cependant pas la preuve que le problème de stagnation a été résolu de manière idéale grâce au multi-niveau. Nous souhaitons donc mettre en œuvre de nouvelles techniques pour essayer de résoudre le problème de stagnation.

1.3 Objectifs de recherche

Notre objectif général est d'implémenter des métaheuristiques efficaces pour la résolution du problème de Covering Design. Plus spécifiquement :

- Nous avons développé un algorithme tabou pour la résolution du Covering Design.
- Afin d'accélérer la recherche, nous avons conçu et implémenté des algorithmes de bas niveau totalement incrémentaux.
- Par ailleurs, nous avons essayé d'analyser les phénomènes qui font que la re-

cherche stagne.

- Pour tenter d'y remédier, nous avons implanté une technique de diversification visant à ressortir de la région dans laquelle la solution est bloquée. Nous tentons de contrer la stagnation en combinant notre algorithme tabou avec un mécanisme de diversification.
- Nous avons aussi essayé d'obtenir un algorithme plus efficace en créant un algorithme mémétique. L'idée était d'insérer notre opérateur de recherche locale dans un mécanisme de plus haut niveau visant à guider la recherche en fournissant des solutions initiales prometteuses.

1.4 Plan du mémoire

Ce mémoire est organisé comme il suit :

Le chapitre 2 donne plus de détails sur le problème de Covering Design ainsi que quelques problèmes qui lui sont apparentés et ses applications pratiques. Nous décrivons quelles sont les méthodes qui ont été proposées pour résoudre le Covering Design. Nous présentons aussi dans cette partie certaines métaheuristiques afin de faciliter la compréhension de ces méthodes dans le mémoire.

Dans le chapitre 3, nous donnons une description détaillée des techniques que nous avons mises en œuvre afin de résoudre le problème. Cela comprend l'algorithme de recherche avec tabous et l'algorithme mémétique que nous avons implémentés, ainsi que les algorithmes de bas niveau que nous avons conçus et implémentés dans nos algorithmes afin de les rendre aussi rapides que possible.

Dans le chapitre 4, nous présentons l'ensemble des résultats que nous avons obtenus avec notre algorithme tabou : quelles sont les performances des algorithmes de bas niveau en terme de vitesse d'exécution et de consommation de mémoire et quels sont les résultats que nous avons réussi à obtenir. Nous analysons aussi le comportement de notre algorithme tabou avec diversification ainsi que les performances de notre algorithme mémétique.

Enfin, le chapitre 5 permet de conclure le mémoire.

Chapitre 2

REVUE DE LITTÉRATURE

Nous présentons, dans une première partie, les problèmes apparentés à savoir les Coverings et Packings et la version la plus généralisée du problème de Covering Design. Nous verrons aussi dans cette partie des applications du problème.

Pour résoudre le Covering Design, nous avons utilisé des heuristiques. C'est pourquoi nous ferons une présentation générale de ces heuristiques dans une seconde partie. Nous verrons enfin quelles méthodes ont été utilisées pour résoudre le problème et en particulier comment on a adapté au problème certaines des heuristiques.

2.1 Problèmes apparentés et applications

2.1.1 Problèmes apparentés au Covering Design

Le problème de Covering Design fait partie de la famille des problèmes de Covering. Un problème de Covering peut se voir comme un problème dans lequel on a deux types d'objets mathématiques. Nous appelons les objets du premier type des points et les objets du second type des blocs. Les blocs ont la propriété de couvrir des points. Le but du problème de Covering est de trouver un ensemble S de blocs tel que tout point est couvert par au moins un des blocs de S. On cherche à optimiser la solution S en cherchant à minimiser le nombre de blocs qu'elle contient.

Par exemple, le problème de Set Covering est le problème dans lequel les blocs sont des ensembles de points prédéfinis. On dit alors qu'un bloc couvre un point si le point considéré appartient à ce bloc. Dans le problème de Covering Design, les points correspondent aux t-subsets. On dit ici qu'un bloc couvre un t-subset s'il est inclus dans le bloc.

Le problème de Covering possède un problème dual qui est le problème de Packing. Dans un problème de Packing, on cherche à choisir un ensemble E de blocs tel que chaque point est couvert par au plus un bloc. On essaie ici de maximiser le nombre

de blocs contenus dans E. Les problèmes de Covering et de Packing ont été référencés par Mills et Mullin dans [30].

Le problème de Covering Design peut être énoncé de façon généralisée. On appelle un $t-(v,m,k,\lambda)$ Covering Design un ensemble de blocs (i.e. des sous-ensembles de $\mathcal{V}$ contenant k éléments) tel que tout sous-ensemble de $\mathcal{V}$ à m éléments (m-subset) ait une intersection d'au moins t éléments avec au moins λ blocs. Il est facile de voir que le problème que nous traitons correspond au cas où $\lambda = 1$ et $m = t$. Il est possible de trouver d'autres énoncés du problème qui sont des cas particuliers du problème généralisé (correspondant par exemple au sous-cas où $m = t$ et au sous-cas où $\lambda = 1$).

Le problème de Covering Code, par exemple, correspond à un cas particulier du Covering Design. Un covering code $C(v,k,R)$ est un ensemble S de blocs tel que n'importe quel bloc est à une distance de Hamming inférieure à R par rapport à l'un des blocs de S. Le problème consiste à trouver un covering code contenant le moins de blocs possible. Le problème de Covering Code tel qu'énoncé correspond au problème de Covering Design dans lequel $t = k - R$, $m = k$ et $\lambda = 1$.

Un autre problème s'avère être très proche du Covering Design : il s'agit du problème de Lotto Design. Le problème peut s'exprimer de la façon suivante : on appelle un lotto design noté $LD(n,k,p,t,b)$ un ensemble de b blocs (sous-ensembles à k éléments de $\{1,...,n\}$) tel que tout p-subset ait une intersection d'au moins t éléments avec au moins un bloc. Le problème du Lotto Design est donc un problème de décision où on cherche à savoir si $LD(n,k,p,t,b)$ existe. Notons que le problème de Covering Design peut alors se voir comme le problème où on cherche le plus petit b tel que $LD(v,k,t,t,b)$ existe.

Le problème de Covering Design possède un problème dual qui est le problème du Packing Design. Dans le problème de Packing Design, on cherche un ensemble de blocs tel que chaque t-subset soit contenu dans au plus λ blocs. On appelle cet ensemble de bloc un $t-(v,k,\lambda)$-packing design. On cherche à maximiser le nombre de blocs du packing.

2.1.2 Applications du problème de Covering Design

Nous avons trouvé peu d'informations quant aux applications. Les articles traitant du Covering Design citent des applications sans les décrire très en détail. L'application qui revient le plus est celle de la génération de tickets de loterie (*lottery schemes* en

anglais, voir [27]). Dans certaines loteries (on en trouve en Italie et en Suède), il est possible de gagner sans avoir tous les bons numéros et d'acheter des super tickets (c'est-à-dire plusieurs tickets regroupés en un seul). Si les numéros qu'on peut tirer vont de 1 à v et que le tirage est de k numéros, alors un $t-(v,k,k,1)$-covering design donne toutes les configurations qui permettent d'avoir au moins t numéros gagnants et $C(v,k,t)$ représente le nombre de tickets. Certains pensent qu'ils peuvent utiliser ces informations pour augmenter leurs chances de gagner. Certaines personnes en ont tiré avantage en vendant sur internet des coverings ou des programmes les générant (voir les liens sur le site de la Jolla [22]). Les autres applications citées sont la compression de données [13] et le décodage par piégeage d'erreurs [7] (*error trapping decoding* en anglais).

2.2 Présentation des heuristiques

La théorie de la complexité a permis de classer les problèmes combinatoires selon la difficulté à répondre à ces problèmes grâce à des algorithmes. Les problèmes NP-Difficile constituent une famille de problèmes ne pouvant pas, à l'heure actuelle, être résolus en temps polynomial sur une machine déterministe [18]. Ces problèmes nécessitent en général un temps CPU ou de l'espace mémoire bien au-delà de ce qu'on peut leur impartir afin de pouvoir être résolus de façon exacte. Les heuristiques constituent une famille de méthodes qui, à défaut d'être exactes, permettent de fournir de bons résultats à ce genre de problème dans un temps imparti généralement faible. Ces méthodes permettent de trouver des solutions qui se rapprochent de l'optimum là où il est impossible de trouver la solution optimale avec des méthodes exactes. On les utilise largement pour pouvoir résoudre des problèmes d'optimisation discrets. Ces problèmes consistent à trouver une configuration optimale parmi un ensemble de configurations données qui forment l'espace de recherche et qui sont évaluées qualitativement par une fonction. On cherche donc à minimiser (ou maximiser selon le problème) la fonction sur l'espace de recherche.

Une revue détaillée sur les métaheuristiques peut être trouvée dans [4]. Plus spécifiquement, on trouve des descriptions sur la recherche locale dans [1], l'algorithme tabou dans [19, 20, 21, 2], le recuit simulé dans [26, 6, 29], l'algorithme GRASP dans [14, 32], la recherche à voisinage variable dans [24], la recherche locale itérée dans [33], les algorithmes évolutionnaires dans [12, 15, 25] et les algorithmes de colonies de

fourmis dans [11]. Plusieurs heuristiques seront évoquées, voire décrites par la suite, aussi voilà ci-après une courte description de ces heuristiques.

2.2.1 L'algorithme glouton

Un algorithme glouton est un algorithme qui génère une solution en utilisant une approche de construction. Le principe consiste ainsi à construire itérativement une solution en complétant pas à pas une solution partielle.

On part d'une solution initiale, généralement vide, et on cherche à compléter étape par étape cette solution partielle. Les différents éléments possibles pour compléter la solution sont évalués par un score et on choisit pour compléter la solution l'élément qui possède le meilleur score. On s'arrête lorsqu'une solution complète est construite. Plusieurs stratégies peuvent être mises en œuvre, par exemple en changeant la façon d'évaluer les éléments. L'efficacité d'un algorithme glouton est d'ailleurs grandement liée à cette façon d'évaluer les candidats.

Les algorithmes gloutons sont des méthodes peu coûteuses à mettre en pratique en terme de temps CPU. Les solutions obtenues ne sont généralement pas optimales, mais fournissent des solutions meilleures que des solutions aléatoires. Des techniques plus élaborées, par exemple celles utilisant la recherche locale, permettent d'obtenir de meilleures solutions.

2.2.2 La recherche locale et l'algorithme de descente

La recherche locale consiste à parcourir une suite de configurations dans l'espace de recherche afin de trouver la meilleure possible. Pour parcourir cet espace, on se munit d'une structure de voisinage, en général la plus simple et naturelle possible et qui, à chaque configuration, associe des configurations qui sont appelées ses voisines.

Un algorithme de recherche locale est un algorithme qui parcourt une chaîne de recherche locale, c'est-à-dire une suite de configurations $(S_i)_{i=1..N}$ de l'espace de recherche telle que, pour tout i, S_{i+1} est un voisin de S_i. À partir d'un mécanisme d'exploration défini, l'algorithme détermine quel sera la prochaine configuration visitée (i.e. si la configuration courante est S_i, le mécanisme d'exploration permet de définir quel sera S_{i+1}).

Un des algorithmes les plus élémentaires qu'on peut imaginer et qui utilise la recherche locale est l'algorithme dit de *best improvement*. Il consiste, partant d'une

configuration en général prise au hasard, à évaluer le coût de chacun de ses voisins et à choisir le voisin qui possède le coût minimal (si on cherche à minimiser la fonction de coût, maximal si on veut la maximiser) et à recommencer partant de la nouvelle configuration choisie. On s'arrête lorsqu'il n'est plus possible d'améliorer la fonction de coût, on obtient de cette façon un optimum local.

Le principe de recherche locale est un principe puissant dans la mesure où il permet de parcourir efficacement un espace de recherche pour trouver des optima locaux. Des algorithmes plus sophistiqués, utilisant la recherche locale, existent et permettent de trouver des solutions de meilleure qualité en ressortant des zones contenant des optima locaux. Nous en présentons deux par la suite : le recuit simulé et la recherche avec tabous.

2.2.3 L'algorithme de recuit simulé

L'algorithme de recuit simulé constitue historiquement la première métaheuristique [29, 26, 6]. Le principe du recuit simulé est de permettre des mouvements aléatoires qui dégradent la solution selon une certaine distribution de probabilité qui évolue au cours de la recherche. À chaque itération, on engendre un mouvement aléatoire qui sera accepté ou non selon un critère qui évolue au cours de la recherche. Ainsi au début de la recherche, l'algorithme accepte fréquemment les mouvements dégradant la solution, tandis qu'à la fin de la recherche, ces mouvements sont presque tout le temps refusés. L'algorithme fait décroître lentement la probabilité d'acceptation des moins bons mouvements (l'algorithme accepte toujours les mouvements améliorant la fonction de coût). La probabilité d'acceptation est modélisée grâce à la notion de température : une haute température correspond à une probabilité d'acceptation élevée et une basse température à une probabilité faible. En faisant décroître la température, ni trop lentement ni trop rapidement, il est possible trouver de bonnes solutions.

L'algorithme de recuit simulé a connu beaucoup de succès et continue à être largement utilisé. Il permet en effet de fournir des résultats très satisfaisants sur de nombreux problèmes.

2.2.4 La recherche avec tabous

La recherche avec tabous est une méthode de recherche locale proposée par Fred Glover en 1986 [21] et qui a connu un grand succès depuis pour les bons résultats qu'elle a obtenus sur de très nombreux problèmes.

La composante principale d'un algorithme tabou est un mécanisme de diversification à court terme qu'on appelle liste taboue. La liste taboue permet d'empêcher de faire des cycles à court terme en interdisant l'algorithme de retourner dans des régions récemment visitées.

Deux mécanismes peuvent être mis en place pour aider la recherche avec tabous. Le premier est un mécanisme de diversification à long terme qui est une façon de se déplacer plus largement dans tout l'espace de recherche afin de visiter de nouvelles régions. Le second est un mécanisme d'intensification qui permet d'explorer plus en profondeur certaines régions qui paraissent contenir des configurations prometteuses.

2.2.5 L'algorithme mémétique

L'algorithme mémétique est une métaheuristique hybride qui combine un algorithme génétique avec un opérateur de recherche locale. Le concept d'un algorithme génétique est de faire évoluer une population de solutions en utilisant généralement deux opérateurs. Le premier est un opérateur de croisement qui permet de créer une nouvelle solution prometteuse en combinant les structures de deux solutions de la population. Le second est un opérateur de mutation qui permet d'altérer légèrement la structure d'une solution.

L'opérateur de recherche locale permet de rendre l'algorithme plus agressif en allant chercher systématiquement les meilleures solutions dans le voisinage des membres de la population. Il est possible de régler les paramètres pour accorder plus d'importance au côté évolutionniste ou à la recherche d'une bonne solution dans l'espace de recherche.

2.3 Les méthodes de résolutions existantes pour le Covering Design

Beaucoup de méthodes ont été mises en œuvre pour résoudre le problème de Covering Design. Les mathématiciens et les informaticiens cherchent en effet des méthodes toujours plus efficaces pour trouver de bons coverings. Le nombre de blocs d'un covering donné constitue une borne supérieure pour le nombre de blocs d'un covering optimal. Un bon covering est un covering dont le nombre de blocs est le plus petit connu.

Un grand nombre des techniques connues ont été décrites par Gordon et al. [23]. Les autres techniques décrites ici correspondent principalement à l'algorithme de recuit simulé de Nurmela et Östergård [31] et l'algorithme coopératif multiniveau avec opérateur tabou de Dai et al. [10]. Nous présenterons ci-après ces méthodes.

2.3.1 Les méthodes mathématiques

Les méthodes que nous appelons mathématiques sont les méthodes qui utilisent des concepts mathématiques avancés pour trouver des coverings. On trouve parmi ces concepts la théorie de Turán et la géométrie projective.

Notons que les travaux les plus anciens ont été réalisés par des mathématiciens qui ont cherché de bons coverings pour des paramètres ou des familles de paramètres spécifiques. D'autres travaux consistent pour des jeux de données particuliers à trouver le nombre de coverings non isomorphes qu'ils possèdent. Ce genre de travaux continue à être mise en œuvre (voir récemment [5, 9, 3]).

D'autres résultats ont été obtenus grâce aux nombres de Turán $T(n, l, r)$ qui représentent le plus petit nombre de r-subsets d'un ensemble à n éléments tels que tout l-subset contienne au moins un des r-subsets. La relation entre C et T s'exprime alors par :

$$C(v, k, t) = T(v, v - t, v - k) \tag{2.1}$$

Le parallélisme entre les deux problèmes a permis d'obtenir le résultat suivant :

$$C(v + 1, k + 1, t + 1) \leq \lfloor (2v - k)C(v, k, t)/v \rfloor + C(v - 1, k + 1, t + 1) \tag{2.2}$$

La construction des coverings peut se faire en utilisant la géométrie finie et en

particulier la géométrie projective [23]. Nous n'entrerons pas dans les détails quant à la manière de construire ces coverings, mais ces méthodes permettent d'obtenir des coverings pour des familles de jeux de données ayant des paramètres v, k et t bien spécifiques. En sont déduits les deux résultats suivants : Soit q, m et k trois entiers, on a :

$$C\left(\frac{q^{m+1}-1}{q-1}, \frac{q^{k+1}-1}{q-1}, k+1\right) \leq \begin{bmatrix} m+1 \\ k+1 \end{bmatrix}_q \tag{2.3}$$

$$C\left(q^m, q^k, k+1\right) \leq q^{m-k}\begin{bmatrix} m \\ k \end{bmatrix}_q \tag{2.4}$$

$$avec \begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{(q^n-1)(q^{n-1}-1)\dots(q^{n-k+1}-1)}{(q^k-1)(q^{k-1}-1)\dots(q-1)} \tag{2.5}$$

2.3.2 Les méthodes d'exploration arborescente

Ces méthodes explorent l'ensemble des solutions comme un arbre de recherche et sont donc des méthodes exactes.

Le plus ancien algorithme développé pour résoudre le Covering Design est d'après la littérature un simple algorithme de retour-arrière (*backtracking*) développé par Bates en 1978.

L'autre méthode d'exploration arborescente trouvée dans la littérature est le Branch-and-Cut de Margot [28]. La méthode de Branch-and-Cut consiste à explorer l'espace de recherche comme un arbre et à élaguer (*pruning*) les branches qui conduisent à des configurations isomorphes à celles qui peuvent se trouver dans une branche déjà explorée. La méthode permet de réduire grandement l'arbre de recherche à parcourir. La méthode n'est cependant pas très compétitive dans la mesure où les améliorations présentées par Margot correspondent à un petit nombre de jeux de données.

2.3.3 Les méthodes de construction de Gordon, Kuperberg et Patashnik

Ces méthodes permettent de construire des coverings à partir d'autres coverings existants. On peut par exemple construire un covering à partir d'un covering plus grand (méthode de réduction) ; ou construire un covering à partir de coverings plus petits (méthode de combinaison). Nous présentons en détail ces deux exemples.

Le principe de la méthode de réduction est le suivant : on connait un (v, k, t)-covering S et on souhaite construire un (v', k', t)-covering tel que $v' < v$ et $k' < k$. Pour ce faire, on choisit v' éléments aléatoirement parmi $\mathcal{V}$. Considérant S, on supprime de ses blocs tous les éléments ne faisant pas partie des v' choisis et on élimine les blocs contenant moins de t éléments.
Prenons alors un bloc, et soit l le nombre d'éléments de ce bloc :
- Si $l < k'$, il suffit d'ajouter des éléments quelconques.
- Si $l = k'$, le bloc a le bon nombre d'éléments et n'a donc pas besoin d'être modifié.
- Si $l > k'$ alors on remplace le bloc par les blocs d'un (l, k', t)-covering.
On obtient alors un (v', k', t)-covering.
Gordon et al. précisent que dans le cas où les paramètres sont "petits" cette méthode fonctionne bien si les rapports k'/k et v'/v sont similaires ; dans le cas où les paramètres sont "grands", si l est proche de $\frac{v'k}{v}$ et si on connait un bon (l, k', t)-covering.

La méthode de combinaison permet de construire un $(v_1 + v_2, k_1 + k_2, t_1 + t_2)$-covering de taille $b_1 \times b_2$ à partir d'un (v_1, k_1, t_1)-covering de taille b_1 et d'un (v_2, k_2, t_2)-covering de taille b_2. Ainsi, un bloc du covering résultant peut toujours être construit à partir de deux blocs appartenant chacun à l'un des coverings plus petits. Si on considère l'ensemble des blocs construits comme la concaténation d'un bloc du (v_1, k_1, t_1) covering et d'un bloc du (v_2, k_2, t_2) alors cet ensemble forme un $(v_1+v_2, k_1+k_2, t_1+t_2)$-covering design. On peut, pour améliorer cette solution, enlever d'emblée les blocs inutiles. Il est possible de trouver un covering, le meilleur possible, en cherchant la meilleure combinaison des petits coverings utilisés. Il est alors possible de construire une solution itérativement en utilisant la programmation dynamique.

D'autres méthodes de construction élémentaires permettent de donner les résultats suivants :

$$C(v, k+1, t) \leq C(v, k, t) \tag{2.6}$$
$$C(v+1, k+1, t) \leq C(v, k, t) \tag{2.7}$$
$$C(v+1, k, t) \leq C(v, k, t) + C(v, k-1, t-1) \tag{2.8}$$
$$C(v-1, k, t) \leq C(v, k, t) \tag{2.9}$$
$$C(v-1, k-1, t-1) \leq \left\lfloor \frac{k}{v} C(v, k, t) \right\rfloor \tag{2.10}$$
$$C(m.v, m.k, t) \leq C(v, k, t) \tag{2.11}$$

Le succès de ces méthodes dépend fortement de la qualité des coverings qui permettent de construire les nouveaux coverings. La méthode de combinaison est très importante car c'est celle qui a permis de trouver les meilleurs résultats sur un grand nombre de jeux de données des archives de la Jolla, en particulier pour les jeux de données que les métaheuristiques ne peuvent pas traiter. Ainsi lorsque des meilleures solutions sont découvertes pour des "petits" covering designs (v et t petits), cette méthode permet très souvent de trouver des améliorations pour des coverings plus grands.

2.3.4 Les méthodes heuristiques

Plusieurs heuristiques ont été appliquées pour résoudre le problème de Covering Design. Parmi ces méthodes, on trouve : un algorithme glouton proposé par Gordon et al. [23], un algorithme de recuit simulé proposé par Nurmela et Östergård [31] et un algorithme coopératif multiniveau avec opérateur tabou proposé par Dai et al. [10].

L'algorithme glouton

Gordon et al. présentent plusieurs variantes d'un algorithme glouton. L'algorithme générique se présente de la façon suivante : on arrange tous les blocs possibles dans une liste. Puis, on choisit itérativement le bloc qui couvre le plus de t-subsets et en cas d'égalité on choisit le premier dans la liste. On s'arrête lorsque tous les t-subsets sont couverts.

La différence entre les différentes variantes présentées provient de la façon dont on arrange les blocs. Les expériences effectuées par Gordon et al. montrent des résultats plutôt similaires, les arrangements utilisés étant (rangés en ordre décroissant d'efficacité) l'ordre lexicographique, l'ordre colex (idem que lexicographique, mais en lisant les nombres de la droite vers la gauche), le code de Gray et un ordre aléatoire.

Les coverings obtenus ont permis de définir des bornes supérieures meilleures que celles obtenues théoriquement. Les méthodes gloutonnes ont fourni les premiers résultats de l'archive La Jolla [22]. Des techniques plus élaborées les ont rapidement battues. On trouve par exemple les techniques utilisant la recherche locale qui permettent d'obtenir de meilleures solutions.

L'algorithme de recuit simulé de Nurmela et Östergård

Le recuit simulé proposé par Nurmela et Östergård constitue la première heuristique développée pour le problème de Covering Design [31]. Une contribution importante de ces auteurs est d'avoir conçu des structures de données efficaces (qui seront reprises par la suite par Dai et al. dans leur algorithme multiniveau).

L'espace de recherche est constitué de configurations S contenant un nombre b de blocs fixé, b étant un paramètre. La fonction de coût est définie par une approche de pénalité. Le problème est en effet un problème de satisfaction de contraintes. On considère le fait de couvrir un t-subset comme une contrainte. Le coût d'une configuration s'évalue alors comme le nombre de contraintes qu'elle viole. On évalue donc le coût comme le nombre de t-subsets non couverts par la configuration considérée. On cherche à minimiser ce coût (une configuration de coût nul est une solution du problème). Deux configurations sont définies comme voisines si elles n'ont qu'un seul bloc non commun et si les deux blocs différents ne diffèrent que par un élément.

Pour ne pas à avoir à calculer en permanence la liste des voisins d'une configuration, les blocs sont numérotés et on conserve pour chacun d'entre eux la liste des blocs qui sont ses voisins. De même, on conserve la liste de t-subsets couverts par chaque bloc. Enfin, on conserve pour chaque t-subset le nombre de blocs qui le couvrent. On peut alors calculer le coût d'un mouvement (où on remplace le bloc B par B') en cherchant dans un tableau le nombre de blocs couvrant les t-subsets couverts par B ou B'. Un bloc couvert par B uniquement et non couvert par B' fait augmenter la fonction de coût de 1. Un bloc non couvert avant le mouvement, mais couvert par B' fait diminuer la fonction de coût de 1. La performance d'un mouvement est donc évaluée en $O\binom{k}{t}$, $\binom{k}{t}$ correspond en effet au nombre de t-subsets couverts par un bloc. Le temps de calcul d'un mouvement est critique dans ce problème. En effet sans toutes les informations en mémoire, le calcul d'un mouvement a une complexité en temps CPU élevé. Un algorithme ne stockant aucune information en mémoire serait donc bien moins rapide.

L'algorithme de recuit simulé implémenté est assez classique. Le schéma de refroidissement choisi consiste à réduire la température par paliers : la température reste la même pour un certain nombre d'itérations. Elle est réduite de façon géométrique (i.e. $T := r \times T$, où r est un paramètre et T représente la température).

À l'époque de la publication des résultats, les tables de la Jolla n'existaient pas. L'algorithme proposé a donné un grand nombre d'améliorations dont certaines n'ont

jamais été battues. En outre, la méthode a été citée et reprise de nombreuses fois. Il existe cependant une limitation importante de la méthode due au fait que les structures de données proposées sont très coûteuses en espace mémoire. La méthode était donc limitée quant aux jeux de données qu'elle pouvait traiter. Le problème continue de restreintre l'ensemble des jeux de données traitables, même si les machines ont beaucoup gagné en capacité mémoire et en vitesse de calcul. Nous avons fait des tests avec cette implémentation et des comparaisons avec ce que nous proposons dans la partie 4.1.

L'algorithme coopératif multiniveau avec opérateur tabou (MLTS) de Dai, Li et Toulouse

Dai et al. ont présenté une méthode hybride novatrice pour résoudre le problème de Covering Design. L'hybridation consistant à associer une approche multiniveau avec un opérateur de recherche tabou.

Le concept général du multiniveau consiste à simplifier plusieurs fois un problème initial en créant des versions de plus en plus grossières de ce problème. Ces différentes versions sont donc de plus en plus simples à résoudre, mais apportent des solutions de moins en moins précises qui peuvent s'éloigner des bonnes solutions. Considérons deux versions A et B telles que B est la version simplifiée de A. Alors une façon de résoudre A consiste à : résoudre B ; projeter sur A la solution trouvée pour B ; raffiner cette solution dans une procédure résolvant A. On peut par exemple élargir le concept en partant du niveau correspondant à la version du problème la plus grossière et en projetant et raffinant de proche en proche jusqu'au niveau correspondant au problème initial. L'approche multiniveau est très souple et peut être mise en œuvre de beaucoup de façons [8]. Nous présenterons par la suite comment Dai et al. l'ont adaptée au Covering Design.

La méthode MLTS repose grandement sur l'idée qu'une solution est composée de bons blocs. Il faut en effet la bonne combinaison de blocs pour construire une solution. On essaie au cours de la recherche de promouvoir les blocs ayant participé à des configurations prometteuses dans l'espoir de trouver une bonne solution construite à partir de ces blocs.

L'approche multiniveau mise en place consiste, au fur et à mesure qu'on monte dans les niveaux, à rendre de moins en moins de blocs disponibles pour la recherche. En utilisant moins de blocs, l'espace de recherche est plus petit donc son parcours

est plus facile, mais on prend le risque de passer à côté de bonnes configurations. Si on appelle $(A_i)_{i \in 0..l}$ la famille de blocs qui constituent les différents niveaux alors A_0 contient tous les blocs et on a $A_l \subset A_{l-1} \subset .. \subset A_1 \subset A_0$. Le voisinage utilisé est le même que celui de Nurmela, mais à chaque niveau i, on utilise les blocs contenus dans A_i exclusivement. En particulier pour le voisinage, on ne peut pas accéder à une configuration voisine si celle-ci contient un bloc non contenu dans A_i. Deux opérateurs ont été mis en œuvre dans l'algorithme :

Un opérateur qu'on appelle *Remonter* consiste à promouvoir les blocs d'une configuration particulièrement bonne d'un niveau i donné en échangeant le niveau où ils sont autorisés avec celui de blocs de rang maximum l.

Soit S_i la meilleure configuration pour le niveau i et M le vecteur associant à chaque bloc le niveau le plus bas auquel il appartient. Le pseudo-code s'écrit :

Procédure **Remonter** (S_i)
  $S := S_i$;
  Tant que S non vide
    Choisir un bloc B dans S aléatoirement ;
    $S := S - B$;
    Si $(M[B] \neq l)$ *alors*
      Choisir un bloc $B' \in A_l$ tel que $B' \notin S_l \cup S_{l-1} \cup ... \cup S_i$;
      // B' est choisi de façon à ne pas rendre illégale à leur niveau les meilleures configurations
      $M[B'] := M[B]$;
      $M[B] := l$;
    FinSi
  FinTantque

Le second opérateur appelé interpolation consiste simplement à initialiser la recherche à un niveau i avec une configuration d'un niveau j tel que $j > i$ (dans l'algorithme, soit $j = i + 1$, soit $i = 0$ et $j = l$). Il est utilisé comme opérateur d'intensification.

L'approche utilisée pour l'opérateur de recherche locale est la même que celle présentée par Nurmela : elle utilise le même espace de recherche et la même fonction de coût. Considérons le mouvement qui consiste à remplacer un bloc B par un bloc B'. Le mécanisme tabou est composé de deux listes : la première est composée de mouvements, lorsque le mouvement est effectué, on rend tabou le mouvement inverse (remplacer B' par B) mais aussi le même mouvement (remplacer B par B'). La

seconde liste contient des blocs, elle consiste après un mouvement à empêcher le bloc entrant (ici B' par exemple) de sortir à nouveau. La seconde liste étant plus restrictive, la longueur de la liste utilisée est plus courte. Notons aussi que la seconde liste empêche en particulier de remplacer B' par B et est donc un peu redondante.

On définit la procédure $TabuSearch(A, S)$ comme la procédure effectuant une recherche avec tabous partant d'une configuration initiale S et en autorisant à entrer dans la configuration seulement des blocs appartenant à l'ensemble A. La procédure retourne la meilleure configuration rencontrée.

La procédure générale est décrite ci-après et consiste dans sa boucle principale à alterner différentes sous-procédures qu'on se propose de décrire par la suite.

Procédure **MLTS**(.)
  Créer les différents niveaux $A_i,\ i \in 0..l$;
  Pour $(i = l; i \leq 0; i--)$ *faire*
    Générer une configuration S composée de b blocs de A_i ;
    $S_i := TabuSearch(A, S)$;
  FinFor
  $j := 1$;
  Tant que le critère de terminaison n'est pas satisfait
    Si $(j\ mod\ 4 \neq 0)$
      $SearchCycle(j, ExplorationFactor)$;
    Si $(j\ mod\ 4 = 0)$
      $InterpolationCycle(j)$;
    Si $((j\ mod\ 4 = 1) \& (j \neq 1))$
      $RestartSearch(j, h, r)$;
    $j{+}{+}$;
  FinTantque

En cours d'exécution l'algorithme exécute de façon répétée les procédures : $SearchCycle, SearchCycle, InterpolationCycle, SearchCycle, RestartSearch$.
La procédure $SearchCycle$ est la procédure de recherche standard. Elle comporte plusieurs phases de recherche tabou, de la diversification possible et l'utilisation de l'opérateur $Remonter$.

Procédure **SearchCycle**(j,$ExplorationFactor$)
  Pour $(i = l; i \geq 0; i--)$ *faire*
    $S := TabuSearch(A_i, S_i)$; //on repart de la meilleur configuration du cycle précédent : S_i
    Si $(i \neq l)$ *alors*

```
Si (f(S) ≥ f(S_i)) alors //f est la fonction de coût
   S := TabuSearch(A_i − A_{i+1}, S) ; //On lance une procédure de diversification
                                         dans le cas où la recherche n'aurait
                                         pas fait mieux qu'avant.
   S := TabuSearch(A_i, S) ;
Sinon
   S := TabuSearch(A_i, S_i) ;
FinSi
//On choisit de faire remonter les blocs de la nouvelle meilleure configuration si celle-ci
   est meilleure ou pas beaucoup moins bien que l'ancienne meilleure configuration
Si (f(S) ≤ f(S_i)+ExplorationFactor) alors
   Remonter(S) ;
   S_i := S ;
Sinon
   Remonter(S_i) ;
FinSi
  FinSi
FinFor
```

La procédure *InterpolationCycle* est la procédure au cours de laquelle on utilise l'opérateur d'interpolation, qui constitue une phase d'intensification.
Le principe consiste à initialiser la recherche d'un niveau i avec la meilleure configuration du niveau $i+1$ cela pour i allant de 0 à $l-1$ et à remonter les blocs des meilleures configurations trouvées. On effectue alors une recherche au niveau l. Enfin on initialise la recherche au niveau 0 avec la meilleure configuration trouvée au niveau l. On remonte à nouveau les blocs de la meilleure solution trouvée.

```
Procédure InterpolationCycle(j)
  For (i = 0; i ≤ l − 1; i++) do
    S := TabuSearch(A_i, S_{i+1}) ; // on initialise la recherche au niveau i avec la meilleure
                                       solution du niveau i + 1
    S_i := S ;
    Remonter(S_i) ;
  FinFor
  S_l := TabuSearch(A_l, S_l) ;
  S_0 := TabuSearch(A_0, S_l) ;
  Remonter(S_0) ;
```

Enfin la procédure *RestartSearch* est la procédure qui permet de relancer périodiquement la recherche à partir d'une configuration construire aléatoirement à partir d'un ensemble de blocs présélectionnés.

Procédure **RestartSearch**(j,h,r)
  Construire un ensemble R contenant les blocs des meilleurs solutions des h derniers
  cycles plus r blocs pris au hasard dans $A_0 - A_1$;
  Calculer une configuration initiale S composée de b blocs pris au hasard dans R ;
  $S_j := TabuSearch(A_j, S)$;

Les résultats obtenus avec cet algorithme ont été particulièrement bons. 38 nouveaux records ont été soumis à la Jolla en 2005, dont 23 qui sont encore d'actualité.

Chapitre 3

TECHNIQUES PROPOSÉES

Dans ce chapitre, nous présentons les techniques que nous avons mises en œuvre pour résoudre le problème de Covering Design. Nous présenterons d'abord les algorithmes de haut niveau, c'est-à-dire les métaheuristiques que nous avons développées : un algorithme tabou et un algorithme mémétique. Nous décrivons ensuite les structures de données et algorithmes de bas niveau utilisés pour accélérer substantiellement l'algorithme.

3.1 Algorithmes de haut niveau

Nous ne résolvons pas directement le problème de Covering Design mais un problème de décision que nous appelons *b-CD Problem*. Ce problème consiste à trouver une solution contenant un nombre de blocs fixé b (b étant alors un nouveau paramètre). On peut en effet tâcher de trouver la solution du problème d'origine en essayant de trouver une succession de solutions pour des b décroissants.

Pour la suite, nous utilisons les notations suivantes : Nous appelons $\mathcal{V}$ un ensemble de référence contenant v éléments, $\mathcal{K}$ l'ensemble des blocs, $\mathcal{T}$ l'ensemble des t-subsets et $\mathcal{I}$ l'ensemble des indices des b blocs contenus dans la configuration courante : $\mathcal{V} = \{1, \ldots, v\}$; $\mathcal{K} = \{P \subset \mathcal{V} \ : \ |P| = k\}$; $\mathcal{T} = \{P \subset \mathcal{V} \ : \ |P| = t\}$; $\mathcal{I} = \{1, \ldots, b\}$.

Espace de recherche et fonction de coût

Pour pouvoir résoudre le b-CD problem, nous adoptons une approche par pénalité. On élargit l'espace de recherche en autorisant des configurations qui ne couvrent pas tous les t-subsets. L'idée consiste à donner une pénalité pour chaque contrainte violée, une contrainte étant de couvrir un t-subset. Les pénalités permettent de quantifier le degré de violation des contraintes - voir [17]. Conformément aux principes de cette approche, une configuration S est un ensemble quelconque de b blocs représentée par

un vecteur :

$$S = (B_1, B_2, \ldots, B_b) \in \mathcal{K}^b \tag{3.1}$$

Soit une configuration S, son coût $f(S)$ est défini par le nombre de t-subsets qui ne sont pas couverts par S :

$$f(S) = |\{\mathrm{T} \in \mathcal{T} \ : \ \forall i \in \mathcal{I}, \ \mathrm{T} \not\subset B_i\}| \tag{3.2}$$

Le but est de trouver une configuration dont le coût est nul, i.e. un covering design contenant b blocs. Pour trouver de telles configuration nous proposons un algorithme tabou et un algorithme mémétique dont une description détaillée est présentée par la suite.

3.1.1 Algorithme tabou

Les principes de l'algorithme tabou et plus généralement de la recherche locale ont été présentés dans les parties 2.2.4 et 2.2.2 respectivement. Nous rappelons que l'algorithme tabou est une technique de recherche locale qui empêche l'algorithme de revenir vers des zones déjà visitées quitte à dégrader la fonction de coût. Nous présentons d'abord les éléments nécessaires à définir le fonctionnement de notre opérateur tabou, à savoir la définition du voisinage et le fonctionnement de la liste taboue.

Fonction de voisinage

Effectuer un mouvement sur une configuration $S = (B_1, \ \ldots, \ B_b)$ consiste à modifier un seul bloc B_i en lui enlevant un élément y et en y insérant un élément x : un tel mouvement est noté $< i, x, y >$. La nouvelle configuration obtenue ainsi est notée $S \oplus < i, x, y >$. Ainsi $S' = S \oplus < i, x, y > = \left(B'_j\right)_{j=1..b}$ est défini par :

- $\forall j \neq i \ B'_j = B_j$
- $B'_i = (B_i - \{y\}) \cup \{x\}$

On appelle performance d'un mouvement son impact sur la fonction de coût. Si nous considèrons une configuration S, la performance d'un mouvement $< i, x, y >$ notée $\delta(i, x, y)$ vaut :

$$\delta(i, x, y) = f(S \oplus < i, x, y >) - f(S) \tag{3.3}$$

Nous introduisons maintenant la définition de ce que nous appelons un mouvement critique. Un mouvement $< i, x, y >$ est dit critique si le fait d'insérer l'élément x dans

le bloc B_i rend possible de couvrir au moins un t-subset non couvert. Dans ce cas, on dit également que la paire (i, x) est critique :

$$\text{le mouvement} < i, x, y > \text{est critique} \Leftrightarrow \text{la paire } (i, x) \text{ est critique}$$
$$\Leftrightarrow (\exists \mathrm{T} \in \mathcal{T} \ : (\forall j \in \mathcal{I},\ \mathrm{T} \not\subset B_j) \ et \ \mathrm{T} \subset (B_i \cup \{x\}))$$

Si nous voyons les t-subsets non couverts comme des erreurs présentes dans la solution alors les mouvements critiques sont des mouvements qui essaient de“réparer” les erreurs, tandis que les mouvements non critiques n'affectent que des parties de la solution où aucune erreur n'est présente. Conformément à une stratégie fréquemment utilisée (voir [17]), et contrairement aux algorithmes proposés par Nurmela [31] et Dai et al. [10], notre algorithme n'effectue que des mouvements critiques, car nous supposons a priori que cette approche est plus efficace.

On peut voir que, si $f(S) > 0$, il existe au moins un mouvement critique. Cette propriété est importante car elle garantit que, tant que l'algorithme n'a pas découvert une solution ($f(S) > 0$), la recherche ne sera pas entravée faute de mouvements critiques. Cette propriété peut être montrée comme il suit :

Supposons que $f(S) > 0$, considérons un t-subset non couvert T, et un bloc B_i tel que $|T - B_i|$ est minimum :

$$\forall j \in \mathcal{I}, \quad 0 < |T - \ B_i| \leq |T - \ B_j|$$

Si $T - B_i = \{x\}$, alors (i, x) est critique.
Sinon, $|T - B_i| \geq 2$.
Considérons un ensemble $E \subseteq B_i - T$, tel que $|E| = |T - B_i| - 1$, et un élément $x \in T - B_i$.
Soit maintenant $T' = E \cup (B_i \cap T) \cup \{x\}$.
Alors $T' - B_i = \{x\}$, donc $|T' - \ B_i| = 1$.
Si T' n'est pas couvert (i, x) est critique.
Sinon, soit B_j le bloc qui couvre T', B_j couvre $(B_i \cap T) \cup \{x\}$ donc $|T - B_j| = |T - B_i| - 1$, contradiction.
On a donc prouvé que (i, x) était nécessairement critique.

Mécanisme tabou

L'algorithme choisit ainsi toujours le meilleur mouvement critique, même si ce mouvement augmente la valeur de la fonction de coût. Afin d'éviter des cycles à court terme dans la recherche, nous introduisons un mécanisme tabou. L'idée est la

suivante : lorsqu'un élément est inséré dans un bloc, l'algorithme empêche cet élément d'être enlevé du bloc pour un certain nombre d'itérations. De même, lorsqu'un élément est retiré d'un bloc, l'algorithme lui interdit d'y être réinséré trop rapidement. De façon plus formelle, nous définissons deux listes taboues qu'on appelle ***Tabu-In*** et ***Tabu-Out*** qui représentent des ensembles contenant des paires $(i, x) \in \mathcal{I} \times \mathcal{V}$. Le statut tabou d'un mouvement dépend du contenu de ***Tabu-In*** et ***Tabu-Out*** :

$$\textit{le mouvement} < i, x, y > \ \textit{est tabou} \Leftrightarrow (i, x) \in \textbf{Tabu-In} \ \textit{ou} \ \ (i, y) \in \textbf{Tabu-Out}$$

Après qu'un mouvement $< i, x, y >$ soit effectué, le couple (i, x) est inséré dans ***Tabu-Out*** pendant $lgtl_{out}$ itérations et le couple (i, y) dans ***Tabu-In*** pendant $lgtl_{in}$ itérations.

Nous avons essayé de résoudre le problème de stagnation en utilisant des listes taboues de longueurs variables. Notre idée consiste à faire varier périodiquement et de façon radicale la longueur de chaque liste. Nous donnons ci-après les formules exactes que nous avons utilisées ainsi qu'une figure représentant la variation de la longueur de la liste. Les valeurs de $lgtl_{out}$ et $lgtl_{in}$ dépendent d'un paramètre $lgtl_0$ et du numéro de l'itération courante :

$$lgtl_{out} = lgtl_0 \times ctr[iter \ mod \ 128] \tag{3.4}$$

tel que

$$\begin{aligned}(ctr[i])_i = (&1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1,\\ &1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,\\ &8, 8, 8, 8, 8, 8, 8, 8, 8, 7, 6, 5, 4, 3, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1,\\ &4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1)\end{aligned}$$

$$lgtl_{in} = lgtl_{out} \times \left\lfloor \frac{v - k}{k} \right\rfloor \tag{3.5}$$

Le fait d'utiliser une longueur de liste variable procure d'avantage de robustesse à l'algorithme, car son efficacité dépend beaucoup moins de la valeur du paramètre $lgtl_0$. Le paramètre est ainsi plus facile à régler.

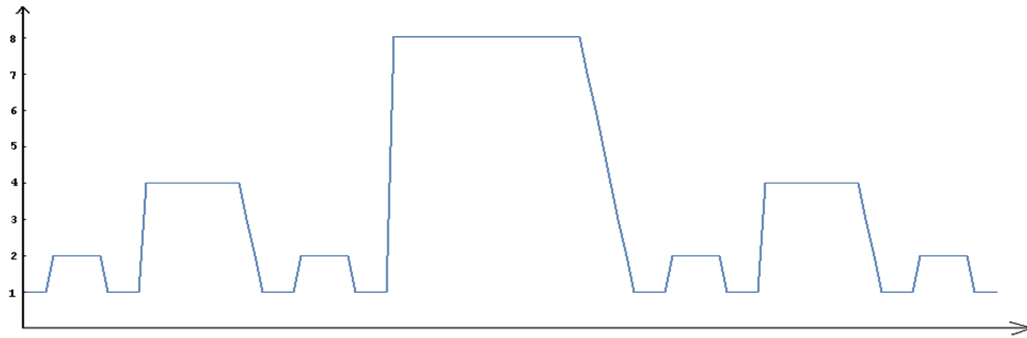

FIGURE 3.1 Schéma représentant une période des variations de la liste tabou entre $lgtl0$ et $8lgtl0$

Mécanisme de diversification

Le mécanisme tabou utilisé dans l'algorithme tabou est, dans de nombreux cas, suffisant pour éviter des cycles à court terme tant qu'une valeur appropriée du paramètre $lgtl_0$ est choisie. Cependant, afin de permettre à l'algorithme de visiter de nouvelles zones de l'espace de recherche et dans l'espoir de contrer le phénomène de stagnation, nous utilisons une technique supplémentaire de diversification. La technique employée est une technique de diversification continue s'appuyant sur la fréquence des attributs de la configuration [35]. Le principe est le suivant : l'algorithme tabou alterne entre des phases normales et des phases plus courtes consacrées à la diversification. Pendant les phases normales, l'algorithme mémorise la fréquence des attributs présents dans les configurations visitées. Puis, durant la phase de diversification qui suit, la fonction de coût est modifiée pour pénaliser les attributs les plus fréquents. Nous définissons les attributs comme des couples (i, x) où i est l'indice d'un bloc et x un élément, et on dit qu'un attribut (i, x) est présent dans la configuration $S = (B_j)_{j=1..b}$ si et seulement si $x \in B_i$. La fréquence d'un attribut (i, x) est noté $Freq(i, x)$. Si on retient les fréquences pour un intervalle $[it_1, it_2 - 1]$, on a quelque soit $(i, x) \in I \times V$:

$$Freq\,(i,x) = \frac{|\{it \in [\mathrm{it}_1,\, \mathrm{it}_2{-}1] : (i,x)\ \ est\ contenu\ dans\ S^{it}\ \}|}{it_2 - it_1} \tag{3.6}$$

où S^{it} représente la configuration générée par l'algorithme tabou à l'itération it. La fonction de coût utilisée pour évaluer S lors des phases de diversification est :

$$f_{div}(S) = f(S) + \alpha \times \sum_{(i,x):x \in B_i} Freq(i,x) \tag{3.7}$$

Pour nos expériences, la valeur de α était fixée à 4.

Pseudo-code de l'opérateur tabou standard

Nous décrivons la procédure **TS-b-reg()** qui effectue une recherche avec tabous sans diversification. En plus de v, k et t, la procédure reçoit 3 paramètres : Sa configuration initiale S_0, le nombre maximum d'itérations $iter_{max}$, et la valeur du paramètre $lgtl_0$ utilisée pour définir la liste taboue. La configuration S_0 est utilisée comme point de départ de la recherche. À chaque itération, l'algorithme choisit le meilleur mouvement critique non tabou, i.e. le mouvement ayant la plus petite valeur $\delta(.)$ (avec choix aléatoire parmi les ex aequo). Les listes taboues sont alors mises à jour selon la technique présentée dans la partie précédente concernant le mécanisme tabou. La matrice des fréquences est aussi mise à jour. La recherche s'arrête lorsqu'une configuration de coût nul est trouvée ou après que le nombre maximum d'itérations ($iter_{max}$) est atteint. Cette procédure renvoie la configuration obtenue à la fin de la recherche ainsi que la matrice des fréquences. Le pseudo-code de cette procédure est le suivant :

Procédure **TS-b-reg**($S_0, iter_{max}, lgtl_0$)
  $S := S_0$;
  $iter := 0$;
  Initialiser la matrice $Freq$;
  Tabu-out := {} ; Tabu-in := {} ;
  Tant que ($iter < iter_{max}$ *et* $f(S) > 0$) *faire*
    Choisir le meilleur mouvement critique $< i, x, y >$ non tabou ;
    $S := S \oplus < i, x, y >$;
    Calculer $lgtl_{out}$ et $lgtl_{in}$ conformément aux équations (3.4) et (3.5) ;
    Insérer (i, x) dans Tabu-out pour $lgtl_{out}$ itérations ;
    Insérer (i, y) dans Tabu-in pour $lgtl_{in}$ itérations ;
    Mettre à jour la matrice $Freq$;
    $iter := iter + 1$;
  FinTantque
  Renvoyer $(S, Freq)$;

Opérateur de diversification

La procédure **TS-b-div()** est la procédure conçue pour effectuer la diversification. Ses paramètres d'entrée sont les mêmes que pour **TS-b-reg()** avec en plus la matrice $Freq$. La procédure est semblable à **TS-b-reg()** mis à part que durant la recherche, la fonction de coût utilisée est f_{div} qui est influencée par la fréquence des attributs (voir équation (3.7)).

Algorithme tabou avec diversification

La procédure **TS-b-CD()** est la procédure de niveau supérieur qui appelle alternativement les opérateurs **TS-b-reg()** et **TS-b-div()**. Les paramètres d'entrée additionnels sont $iter_{reg}$ et $iter_{div}$ qui indiquent le nombre d'itérations maximum des phases normale et de diversification respectivement. La procédure **TS-b-CD()** est la suivante :

Procédure **TS-b-CD**($S_0, iter_{max}, iter_{reg}, iter_{div}, lgtl_0$)
  $S := S_0$;
  $Nb_{steps} := (iter_{max} - iter_{reg})/(iter_{div} + iter_{reg})$;
  Faire Nb_{steps} fois
    $(S, Freq)$:=TS-b-reg($S, iter_{reg}, lgtl_0$) ;
    Si $f(S) = 0$ *alors* Renvoyer S ;
    $S :=$ TS-b-div($S, iter_{div}, lgtl_0, Freq$) ;
  $S :=$ TS-b-reg($S, iter_{reg}, lgtl_0$) ;
  Renvoyer S ;

3.1.2 Algorithme mémétique

Dans cette section nous présentons l'algorithme mémétique que nous avons implémenté. Nous rappelons que l'algorithme mémétique est un algorithme hybridant une approche évolutive et un opérateur de recherche locale (voir partie 2.2.5). L'approche évolutive permet de générer de nouveaux membres en combinant la structure de membres existants. L'opérateur de recherche locale permet de chercher de bonnes solutions dans le voisinage des configurations de la population. Nous utilisons l'algorithme mémétique afin de générer des configurations prometteuses qui servent de points de départ dans l'espace de recherche pour notre opérateur tabou. Nous présenterons

d'abord l'algorithme en général pour ensuite nous intéresser aux différents croisements que nous avons essayés.

Procédure générale

Notre algorithme mémétique fait évoluer une population P de configurations. Il résout le problème de décision b-CD et renvoie VRAI s'il trouve une solution et FAUX sinon. Le pseudo-code de notre algorithme mémétique se présente comme suit :

Procédure **MA-CD**$(b, iter_{max}, taille_{pop}, nb_{gen})$
  P :=InitPopulation$(taille_{pop})$;
  Pour $i = 0..nb_{gen}$ *faire*
    (S_1, S_2) :=ChoixParents(P) ;
    S :=Croisement(b, S_1, S_2) ;
    S :=TS-b-reg$(S, iter_{max}, ..)$;
    Si $f(S) = 0$ *alors*
      Renvoyer $VRAI$;
    P :=MiseAJour(P, S) ;
  finPour
  Renvoyer $FAUX$;

Les paramètres de l'algorithme sont :
- b : correspond à la taille du covering design qu'on cherche.
- $iter_{max}$: donne le nombre maximum d'itérations effectuées à chaque appel de l'algorithme TS-b-reg.
- $taille_{pop}$: indique la taille de la population P manipulée.
- nb_{gen} : correspond au nombre de générations effectuées par l'algorithme avant de s'arrêter.

Après avoir créé une population initiale, l'algorithme effectue des générations. Lors d'une génération, l'algorithme choisit deux membres de la population S_1 et S_2 (grâce à la procédure ChoixParents) afin de générer grâce à un croisement un nouvel individu S (procédure Croisement). Cet individu, qui est en fait une nouvelle configuration, sera le point de départ d'une recherche avec tabous. Si cette configuration aboutit à une solution (c'est-à-dire qui couvre tous les t-subsets) alors l'algorithme renvoie $VRAI$. L'algorithme met à jour la population P en tenant compte du nouvel individu S (procédure MiseAJour). Puis on recommence une nouvelle génération. Si aucune

solution n'a été trouvée au cours de toutes les générations, alors l'algorithme renvoie $FAUX$.

Procédure InitPopulation : La procédure InitPopulation est la procédure qui génère et initialise la population P de configurations. Cette procédure consiste à générer aléatoirement $taille_{pop}$ configurations puis à effectuer une recherche avec tabous à partir de chacune des configurations en utilisant l'algorithme TS-b-reg. Pour rappel, l'algorithme TS-b-reg prend une configuration S, effectue $iter_{max}$ mouvements (s'il ne trouve pas de solution) et renvoie la configuration courante à laquel il a abouti. Si l'algorithme trouve une solution alors c'est cette solution qu'il renvoie.

Procédure ChoixParents : La procédure ChoixParents est la procédure qui détermine quels membres de la population sont utilisés pour engendrer de nouvelles configurations. Notre procédure choisit deux configurations distinctes de la population au hasard.

La procédure Croisement, dont nous parlerons par la suite, permet d'engendrer la nouvelle configuration.

Procédure MiseAJour : La procédure MiseAJour est la procédure qui décide comment la population évolue.
Nous avons choisi de faire un algorithme génétique stationnaire. Cela signifie qu'une fois les configurations parentes choisies et que la nouvelle configuration est engendrée, la configuration engendrée remplace la configuration parente qui possède le moins bon score.

Opérateur de croisement

La procédure Croisement permet d'engendrer une nouvelle configuration à partir de deux configurations existantes. Cette procédure est cruciale pour le bon fonctionnement de l'algorithme mémétique. En effet, c'est cette procédure qui permet de créer de nouveaux points de départ pour l'opérateur tabou dans l'espace de recherche. Un croisement vise à placer l'algorithme dans des régions potentiellement prometteuses de l'espace de recherche. Nous décrivons ici les différentes procédures de croisement que nous avons essayées :

Croisement X1 utilisant les distances entre les blocs des configurations : Le premier croisement que nous avons implanté est inspiré du croisement utilisé dans l'algorithme mémétique conçu par Galinier et Hao pour le problème de coloriage de graphe dans [16]. Le croisement X1 essaie de créer une configuration en choisissant des blocs dans les deux configurations parentes alternativement. Chaque bloc est choisi de façon à être le plus différent possible des blocs qui n'appartiennent pas à la même configuration d'origine et qui sont déjà présents dans le résultat partiel qui formera la configuration engendrée. En cas d'égalité, l'algorithme choisit un bloc au hasard parmi les ex aequo. En particulier, le tout premier bloc choisi le sera au hasard, car il n'y a aucun bloc déjà présent dans le résultat partiel.

Afin d'effectuer ce croisement, nous définissons une notion de distance entre deux blocs. On utilise une distance de Hamming : la distance entre deux blocs se définit comme le nombre d'éléments qui sont présents dans un bloc, mais pas dans l'autre. Ainsi, deux blocs identiques se situent à une distance nulle l'un de l'autre, tandis que deux blocs n'ayant aucun élément en commun (si $v \geq 2k$) se situent à une distance maximale (égale à $2k$). Le croisement est décrit de la façon suivante :

Procédure $\mathbf{X1}(b, S_1, S_2)$
  $(B1_i)_{i=1..b} = S_1$; $(B2_j)_{j=1..b} = S_2$;
  Pour $i, j = 0..b$ *faire*
    $D(B1_i, B2_j) :=$ distance entre $B1_i$ et $B2_j$;
  $S := \{\}$;
  Pour $i = 0..b$ *faire*
    Si b est pair *alors*
      Choisir le bloc $B \in S_1 - S$ tel que
      $min_{B' \in S_2 \cap S}\{D(B, B')\}$ est maximal ;
      $S := S \cup B$;
    Sinon
      Choisir le bloc $B' \in S_2 - S$ tel que
      $min_{B \in S_1 \cap S}\{D(B, B')\}$ est maximal ;
      $S := S \cup B'$;
  finPour
  Renvoyer S ;

Par souci de clarté, nous n'avons pas indiqué dans ce pseudo-code les cas d'ex aequo qui sont départagés au hasard.

Croisement X2 utilisant une technique gloutonne : De la même façon que dans le croisement X1, l'algorithme choisit les blocs alternativement dans les configurations parentes. Maintenant, par contre, l'algorithme choisit à chaque fois le bloc qui couvre le plus de t-subsets non déjà couverts par les blocs déjà sélectionnés (on choisit au hasard entre les éventuels ex aequo). En cela, nous pouvons dire que le croisement X2 a le même comportement qu'un algorithme glouton, mais qui choisirait parmi un ensemble plus restreint de blocs (les blocs des solutions parentes). Le pseudo-code du croisement X2 est le suivant :

Procédure **X2**(b, S_1, S_2)
  $S := \{\}$;
  Pour $i = 0..b$ *faire*
    Si b est pair *alors*
      Choisir le bloc $B \in S_1 - S$ qui couvre
      le plus de t-subsets non déjà couverts par les blocs de S ;
      $S := S \cup B$;
    Sinon
      Choisir le bloc $B' \in S_2 - S$ qui couvre
      le plus de t-subsets non déjà couverts par les blocs de S ;
      $S := S \cup B'$;
  finPour
  Renvoyer S ;

Croisement X3 essayant de préserver la cohérence d'une configuration : Le troisième croisement que nous avons essayé est un croisement qui essaie de conserver la structure formée par une configuration. Le principe consiste à former une demi-configuration à partir des blocs de la première configuration et à la compléter avec des blocs de la seconde configuration.

Pour former la première moitié de la configuration résultat, l'algorithme sélectionne des blocs proches les uns des autres au sens de la distance définie pour le croisement X1.

La seconde moitié sera complétée de façon gloutonne en choisissant des blocs qui couvrent à chaque fois un maximum de t-subsets non déjà couverts de la même façon que dans le croisement X2. L'algorithme est le suivant :

Procédure **X3**(b, S_1, S_2)
  $S := \{\}$;

Pour $i = 0..\frac{b}{2}$ *faire*
Choisir le bloc $B \in S_1 - S$ tel que
$min_{B' \in S}\{D(B, B')\}$ est minimal ;
$S := S \cup B$;
finPour
Pour $i = (\frac{b}{2} + 1)..b$ *faire*
Choisir le bloc $B' \in S_2 - S$ qui couvre
le plus de t-subsets non déjà couverts par les blocs de S ;
$S := S \cup B'$;
finPour
Renvoyer S ;

3.1.3 Résolution du problème d'optimisation

Les procédures tabou et mémétique présentées plus haut résolvent le problème b-CD, c'est-à-dire le problème de décision qui consiste, pour un b donné, à chercher un covering de taille b. Afin de résoudre le problème de Covering Design, c'est-à-dire le problème d'optimisation cherchant à trouver une solution avec le moins de blocs possible, nous avons développé une procédure de haut niveau utilisant l'algorithme tabou avec diversification présenté auparavant. La procédure essaie de résoudre successivement le problème de décision à b fixé pour des valeurs de b qu'on décrémente en partant d'une valeur fixée par un paramètre.

La procédure **TS-CD()** construit tout d'abord une configuration aléatoire de b_0 blocs. Puis elle appelle la procédure **TS-b-CD()** avec le paramètre b égal à b_0. Si la procédure parvient à trouver une solution, i.e. une configuration S à coût nul, un bloc de S est enlevé afin de construire une configuration de $b_0 - 1$ blocs (le bloc supprimé est celui qui permet d'obtenir une nouvelle configuration couvrant le maximum de t-subsets). La nouvelle configuration est transmise à la procédure **TS-b-CD()** avec b égal à $b_0 - 1$. Ce processus est répété avec des valeurs décroissantes de b jusqu'à ce la procédure **TS-b-CD()** échoue.

Procédure **TS-CD**($b_0, iter_{max}, iter_{div}, iter_{reg}, lgtl_0$)
Pour $i = 1..b_0$ *faire*
$B_i :=$ Choisir aléatoirement k entiers différents
compris entre 1 et v ;
$S := (B_1, \ldots, B_{b_0})$;
$stop := FAUX$;

$b := b_0$;
Tant que $stop = FAUX$ *faire*
  S :=TS-b-CD($S, iter_{max}, iter_{reg}, iter_{div}, lgtl_0$) ;
  Si $f(S) = 0$ *alors*
    $S := S-$ {le bloc tel que la $(b-1)$-configuration obtenue
        couvre un maximum de t-subsets} ;
    $b := b - 1$;
  Sinon
    $stop := VRAI$;
Renvoyer $b + 1$;

3.2 Algorithmes de bas niveau

Dans cette partie, nous décrivons l'implémentation de l'algorithme tabou et les algorithmes de bas niveau utilisés. Dans l'algorithme tabou, il faut calculer à chaque itération la performance de chacun des mouvements possibles. Le problème de Covering Design est un problème dans lequel calculer la performance d'un mouvement est a priori cher en temps de calcul. Il y a en effet une explosion combinatoire du nombre de contraintes (une contrainte consiste à couvrir un t-subset et il y a un nombre exponentiel de t-subsets). Il est donc nécessaire de rendre efficaces les opérations permettant de calculer la performance d'un mouvement.

Nous rappelons que rendre un algorithme incrémental consiste à accélérer le calcul de la performance d'un mouvement en ne considérant que ce qui a changé dans la configuration (voir partie 1.2). Nous rappelons aussi que Nurmela a déjà proposé des algorithmes de bas niveau incrémentaux (voir partie 2.3.4). Nous proposons des algorithmes de bas niveau plus efficaces que ceux de Nurmela et que nous appelons *totalement incrémentaux*. Nos algorithmes totalement incrémentaux permettent d'évaluer la performance d'un mouvement en temps constant.

Nous montrons dans cette partie comment concevoir de telles structures de données et comment on peut les mettre à jour.

3.2.1 Définition des structures de données

Par la suite, on considérera une configuration de référence $S = (B_i)_{i=1..b}$. Étant donné un t-subset T, nous appelons $\mathbf{cov(T, S)}$ le nombre de blocs dans S qui couvrent

$T : cov(T,S) = |\{i \in \mathcal{I} \ : \ T \subseteq B_i\}|$. Soit l'indice d'un bloc $i \in \mathcal{I}$, $x \in V - B_i$ et $y \in B_i$, nous définissons $< +, i, x >$ l'opération consistant à insérer x dans le bloc B_i, et $< -, i, y >$ l'opération consistant à retirer y de B_i :

$$S' = S \oplus < +, i, x > \Leftrightarrow \left(\forall j \neq i B'_j = B_j et B'_i = B_i \cup \{x\}\right) \quad (3.8)$$

$$S'' = S' \oplus < -, i, y > \Leftrightarrow \left(\forall j \neq i B''_j = B'_j et B''_i = B'_i - \{y\}\right) \quad (3.9)$$

Nous définissons alors $\gamma^+ (\mathbf{i}, \mathbf{x})$ le nombre de t-subsets qui ne sont pas couverts dans S mais qui seraient couverts si l'élément x serait inséré dans B_i. De même, $\gamma^- (\mathbf{i}, \mathbf{y})$ représente le nombre de t-subsets couverts dans S et qui ne le seraient plus si on enlevait l'élément y de B_i :

$$\gamma^+ (i, x) = f (S \oplus < +, i, x >) - f (S) \quad (3.10)$$

$$\gamma^- (i, y) = f (S) - f (S \oplus < -, i, y >) \quad (3.11)$$

Enfin, nous définissons $\theta (i, x, y)$ la composante corrective égale à $\gamma^+ (i, x) - \gamma^- (i, y) - \delta (i, x, y)$ de sorte que :

$$\delta (i, x, y) = \gamma^+ (i, x) - \gamma^- (i, y) - \theta (i, x, y) \quad (3.12)$$

Dans l'implémentation proposée, nous utilisons un vecteur de taille $\binom{v}{t}$, trois matrices de tailles respectives $b \times k$, $b \times (v - k)$ et $b \times v \times v$ afin de conserver les valeurs de cov, γ^-, γ^+ et θ, respectivement. Ces structures de données sont initialisées au début de la recherche. Elles sont mises à jour à chaque itération lorsqu'un mouvement est effectué. La performance $\delta (i, x, y)$ de chaque mouvement $< i, x, y >$ peut alors être obtenue en temps constant grâce à l'équation (3.12), car les valeurs nécessaires au calcul sont simplement lues dans les matrices γ^+, γ^- and θ.

3.2.2 Formules de mise à jour de γ^+, γ^- et θ

Afin de déterminer comment initialiser et mettre à jour les structures de données, nous analysons ce qu'il se passe au niveau de chaque t-subset. Ainsi, nous introduisons les nouvelles notations suivantes : $\gamma^+_T(.)$, $\gamma^-_T(.)$ et $\theta_T(.)$ qui correspondent à la projection de $\gamma^+(.)$, $\gamma^-(.)$ et $\theta(.)$ sur un t-subset T en particulier.
De plus, nous définissons $\mathcal{I}^0_{T,S}$ l'ensemble des indices des blocs de la configuration S

qui couvrent T :

$$\mathcal{I}^0_{T,S} = \{i \ : |T - B_i| = 0\} \tag{3.13}$$

$\mathcal{I}^1_{T,S}$ les indices des blocs qui couvrent tous les éléments de T sauf un :

$$\mathcal{I}^1_{T,S} = \{i \ : |T - B_i| = 1\} \tag{3.14}$$

$\mathcal{I}^2_{T,S}$ les indices des blocs qui couvrent tous les éléments de T sauf deux :

$$\mathcal{I}^2_{T,S} = \{i \ : |T - B_i| = 2\} \tag{3.15}$$

Enfin $\mathcal{I}^{>2}_{T,S}$ l'ensemble des indices des autres blocs :

$$\mathcal{I}^{>2}_{T,S} = \mathcal{I} - \mathcal{I}^0_{T,S} - \mathcal{I}^1_{T,S} - \mathcal{I}^2_{T,S} \tag{3.16}$$

On remarque que $\mathcal{I}^0_{T,S}$, $\mathcal{I}^1_{T,S}$, $\mathcal{I}^2_{T,S}$ et $\mathcal{I}^{>2}_{T,S}$ forment une partition de $\mathcal{I}$,
et que $cov(T,S) = \left|\mathcal{I}^0_{T,S}\right|$.
Nous définissons $\mathbf{f(T,S)}$ par :

- $f(T,S) = 1$ si $cov(T,S) = 0$
- $f(T,S) = 0$ sinon

Ainsi, $f(T,S)$ correspond à la contribution d'un t-subset T à la fonction de coût f et on a :

$$f(S) = \sum_{T \in \mathcal{T}} f(T,S) \tag{3.17}$$

Nous introduisons les définitions suivantes, $\forall i \in I,\ \forall x \in \mathcal{V} - B_i, \forall y \in B_i$:

$$\gamma^+_{T,S}(i,x) = f(T, S\oplus <+,i,x>) - f(T,S) \tag{3.18}$$

$$\gamma^-_{T,S}(i,y) = f(T,S) - f(T, S\oplus <-,i,y>) \tag{3.19}$$

$$\theta_{T,S}(i,x,y) = \gamma^+_{T,S}(i,x) - \gamma^-_{T,S}(i,y) - (f(T, S\oplus <i,x,y>) - f(T,S)) \tag{3.20}$$

On peut remarquer que :

$$\gamma^+(i,x) = \sum_{T \in \mathcal{T}} \gamma^+_{T,S}(i,x) \tag{3.21}$$

$$\gamma^-(i,y) = \sum_{T \in \mathcal{T}} \gamma^-_{T,S}(i,y) \tag{3.22}$$

$$\theta(i,x,y) = \sum_{T \in \mathcal{T}} \theta_{T,S}(i,x,y) \tag{3.23}$$

Nous avons la propriété suivante :
Pour tout t-subset T, les valeurs de $\gamma^+_{T,S}$, $\gamma^-_{T,S}$ et $\theta_{T,S}(i,x,y)$ valent 0, sauf dans les cas indiqués dans le tableau 3.1 ci-après.

Tableau 3.1 Contributions aux matrices γ et θ selon les différents cas de figures

Cas	Condition	Eléments de $\gamma^+_{T,S}(.)$, $\gamma^-_{T,S}(.)$, et $\theta_{T,S}(.)$ différents de zéro.
C	$cov(T,S) = 0$	$\forall i \in \mathcal{I}^1_{T,S}, T - B_i = \{x\}, \quad \gamma^+_{T,S}(i,x) = 1$ $\forall y \in T \cap B_i, \quad \theta_{T,S}(i,x,y) = 1$
R	$cov(T,S) = 1$	$\mathcal{I}^0_{T,S} = \{i\}, \ \forall y \in T, \quad \gamma^-_{T,S}(i,y) = 1$

Les matrices γ^+, γ^- et θ peuvent être initialisées de la façon suivante : on énumère tous les t-subsets tels que $cov(T,S) = 0$ *ou* 1 ; on met à jour les structures de données en fonction de la valeur de $cov(T,S)$. Considérons à présent un mouvement m. Nous appelons $\Delta\gamma^+_{T,S}(m,i,x)$ le changement dans $\gamma^+_{T,S}(i,x)$ après que le mouvement m soit effectué et défini par :

$$\Delta\gamma^+_{T,S}(m,i,x) = \gamma^+_T(i,x,S \oplus m) - \gamma^+_T(i,x,S) \tag{3.24}$$

$\Delta\gamma^-_{T,S}(m,i,y)$, $\Delta\theta_T(m,i,x,y,S)$, $\Delta cov(m,T,S)$ se définissent de la même façon. En considérant tous les cas possibles, on remarque que les valeurs de $\Delta\gamma^+_{T,S}(m,i,x)$, $\Delta\gamma^-_{T,S}(m,i,y)$, $\Delta\theta_{T,S}(m,i,x,y)$ et $\Delta cov(m,T,S)$ sont nulles sauf dans des cas précis détaillés dans le tableau suivant. Il faut remarquer que nous utilisons des notations simplifiées dans le tableau. Ainsi, $\Delta\gamma^+_T(i,x)$ remplace $\Delta\gamma^+_{T,S}(m,i,x)$, $\Delta\gamma^-_T(i,y)$ remplace $\Delta\gamma^-_{T,S}(m,i,y)$, $\Delta\theta_T(i,x,y)$ remplace $\Delta\theta_{T,S}(m,i,x,y)$ et $cov(T)$ remplace $cov(T,S)$.

3.2.3 Algorithmes de bas niveau

Après l'exécution d'un mouvement $m = (i_m, x_m, y_m)$, les matrices γ^+, γ^- et θ peuvent être mises à jour ainsi : pour chaque t-subset T, l'algorithme vérifie si T vérifie les conditions correspondant à l'une des lignes de tableaux 3.2 ou 3.3. Si c'est le cas, l'algorithme met à jour les matrices γ^+, γ^- et θ et le tableau *cov* selon les valeurs de $\Delta\gamma^+_{T,S}(.)$, etc. En pratique, plutôt que d'énumérer tous les t-subsets, il est plus efficace d'énumérer les t-subsets qui vérifient les conditions de la ligne l, pour $l = 1..7$.

Tableau 3.2 Modification des matrices γ et θ selon les différents cas de figures (cas 1 à 4)

Cas	Conditions	Eléments de $\Delta\gamma^+_T(.)$, $\Delta\gamma^-_T(.)$,$\Delta\theta_T(.)$, et $\Delta cov(.)$ différents de zero
1	$cov\,(T,S) = 0$ $x_m \in T$ $y_m \notin T$ $i_m \in I^1_T$	$\Delta cov(T) = +1$ $\Delta\gamma^+_T\,(i_m, x_m) = -1$ $\Delta\gamma^-_T\,(i_m, x_m) = +1$ $\forall y' \in T - \{x_m\}\,, \quad \Delta\theta_T\,(i_m, x_m, y') = -1$ $\Delta\gamma^-_T\,(i_m, y') = +1$ $\forall j \in I_{T,1}, j \neq i_m,\; p = T - B_j, \quad \Delta\gamma^+_T\,(j,p) = -1$ $\forall q \in T - \{p\}\,, \quad \Delta\theta_T\,(j,p,q) = -1$
2	$cov\,(T,S) = 1$ $x_m \in T$ $y_m \notin T$ $i_m \in I^1_T$	$\Delta cov(T) = +1$ $soit\; I_{T,0} = \{l\}\,, \forall q \in T, \quad \Delta\gamma^-_T\,(l,q) = -1$
3	$cov\,(T,S) = 1$ $x_m \notin T$ $y_m \in T$ $i_m \in I^0_T$	$\Delta cov(T) = -1$ $\Delta\gamma^+_T\,(i_m, y_m) = +1$ $\Delta\gamma^-_T\,(i_m, y_m) = -1$ $\forall x' \in T - \{y_m\}\,, \quad \Delta\theta_T\,(i_m, x', y_m) = +1$ $\Delta\gamma^-_T\,(i_m, x') = -1$ $\forall j \in I_{T,1}, j \neq i_m,\; p = T - B_j, \quad \Delta\gamma^+_T\,(j,p) = +1$ $\forall q \in T - \{p\}\,, \quad \Delta\theta_T\,(j,p,q) = +1$
4	$cov\,(T,S) = 2$ $x_m \notin T$ $y_m \in T$ $i_m \in I^0_T$	$\Delta cov(T) = -1$ $soit\; I_{T,0} = \{i_m, l\}\,, \; \forall q \in T, \quad \Delta\gamma^-_T\,(m,l,q) = +1$

TABLEAU 3.3 Modification des matrices γ et θ selon les différents cas de figures (cas 5 à 7)

Cas	Conditions	Eléments de $\Delta\gamma_T^+(.)$, $\Delta\gamma_T^-(.)$,$\Delta\theta_T(.)$, et $\Delta cov(.)$ différents de zero
5	$cov(T,S)=0$ $x_m \in T$ $y_m \notin T$ $i_m \in I_T^2$	$p = T - B_{i_m} - \{x_m\}, \quad \Delta\gamma_T^+(i_m,p) = +1$ $\forall q \in T - \{p\}, \quad \Delta\theta_T(i_m,p,q) = +1$
6	$cov(T,S)=0$ $x_m \notin T$ $y_m \in T$ $i_m \in I_T^1$	$p = T - B_{i_m}, \quad \Delta\gamma_T^+(i_m,p) = -1$ $\forall q \in T - \{p\}, \quad \Delta\theta_T(i_m,p,q) = -1$
7	$cov(T,S)=0$ $x_m \in T$ $y_m \in T$ $i_m \in I_T^1$	$\Delta\gamma_T^+(i_m,x_m) = -1$ $\forall y' \in T - \{x_m\}, \quad \Delta\theta_T(i_m,x_m,y') = -1$ $\Delta\gamma_T^+(i_m,y_m) = +1$ $\forall x' \in T - \{y_m\}, \quad \Delta\theta_T(i_m,x',y_m) = +1$

3.2.4 Illustration

Dans cette partie, nous expliquons à l'aide de figures comment s'effectuent les mises à jour décrites dans les parties précédentes. Pour cela, nous montrons comment obtenir les formules dans les tableaux 3.1,3.2 et 3.3. Nous rappelons que les valeurs dans la matrice γ^+ indiquent des gains de t-subsets couverts donc de contraintes respectées, les valeurs de γ^- des pertes et les valeurs dans θ des termes correctifs.

Le tableau 3.1 présente deux situations qu'on appelle situations statiques. Lorsqu'un t-subset et un bloc sont dans une de ces situations, ils contribuent aux valeurs des matrices γ^+, γ^- ou θ.

La première situation, que nous appelons situation *Candidat* ou situation “C”, est la situation dans laquelle un bloc B_i est en bonne position pour couvrir un t-subset T non couvert. Ainsi, si on a un t-subset T non couvert (i.e. tel que $cov(T,S)=0$) et un bloc B_i couvrant tous les éléments de T sauf un (tel que $i \in \mathcal{I}_{T,S}^1$), alors soit x l'élément de T non couvert ($x = T - B_i$). La contribution $\gamma_{T,S}^+(i,x)$ à la matrice γ^+ est de 1 car le fait d'ajouter x au bloc B_i permet de couvrir le t-subset T comme on peut le voir sur la situation “C” de la figure 3.2 ci-après. Les contributions à la matrice θ permettent de prendre en compte le fait que si on ajoute x à B_i mais qu'on lui retire un élément qui appartient aussi au t-subset T alors T ne sera pas couvert

après le mouvement (et ainsi $\gamma^+_{T,S}(i,x) - \theta_{T,S}(i,x,y) = 1 - 1 = 0$).

La seconde situation, que nous appelons situation *Rempart* ou situation “R”, est la situation dans laquelle un bloc B_i est le seul bloc à couvrir un t-subset T. Ainsi, dans cette situation, le seul bloc qui couvre T est B_i donc $cov(T,S) = 1$ (un seul bloc de la configuration S couvre T) et $\mathcal{I}^0_{T,S} = \{i\}$ (l'ensemble des blocs qui couvre T est le singleton $\{i\}$). Alors, pour tous les éléments y de T, la contribution $\gamma^-_{T,S}(i,y)$ à la matrice γ^- est de 1 car le fait de retirer l'élément y du bloc B_i découvrirait le t-subset T comme on peut le voir sur la situation “R” de la figure 3.2 ci-après.

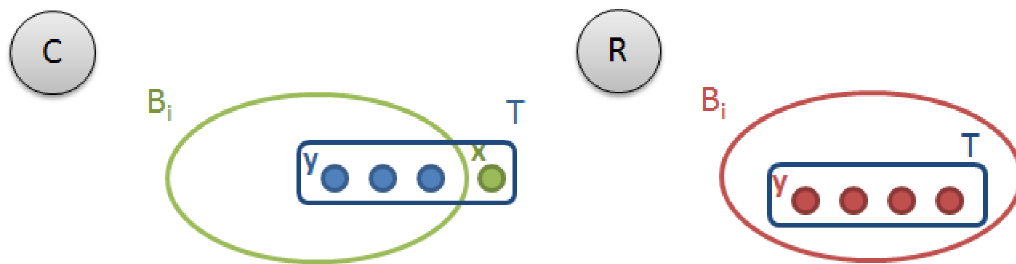

FIGURE 3.2 Situations statiques : Ce schéma représente les éléments par des disques (par exemple l'élément x est représenté par un disque vert). Le t-subset T est représenté par un rectangle bleu. Le bloc B_i est représenté par un ovale vert dans la première situation et bleu dans la seconde. Remarquons qu'ici un élément est contenu dans un ensemble, si le disque qui représente l'élement est à l'intérieur de la figure représentant cet ensemble. Par exemple sur le schéma de gauche, l'élément x est contenu dans T car le disque qui le représente est contenu dans le rectangle bleu, mais n'est pas contenu dans B_i car le disque vert n'est pas à l'intérieur de l'ovale vert.

Considérons à présent le mouvement $< i, x, y >$. Les mises à jour dans les tableaux 3.1,3.2 et 3.3 correspondent simplement aux changements de situations des blocs considérés par rapport à un t-subset T. Par exemple, dans le cas 1 du tableau, le bloc B_i et le t-subset T sont dans la situation *Candidat* avant le mouvement (car $cov(T,S) = 0$, $i \in \mathcal{I}^1_{T,S}$ et $x \in T$) et dans la situation *Rempart* après (car $y \notin T$ donc le bloc B_i couvre T après le mouvement et est le seul dans ce cas). Les éventuels blocs B_j avec $j \neq i$ qui étaient aussi dans la situation *Candidat* pour couvrir T avant le mouvement ne sont plus dans cette situation après (lorsqu'un bloc n'est dans aucune des deux situations particulières on dira qu'il est en situation “0”).

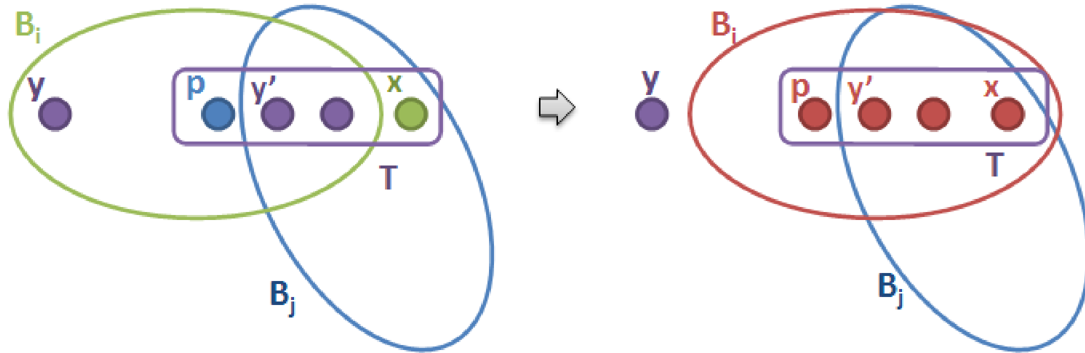

FIGURE 3.3 Cas 1 : Le bloc B_i en situation *Candidat* couvre le t-subset T grâce au mouvement

On peut, à partir de cette analyse, déduire les variations affichées dans le tableau 3.2 de la façon suivante.

Avant le mouvement, B_i et les éventuels B_j sont en situation *Candidat*, donc on a :

- $\gamma^{+}_{T,S}(i,x) = 1$
- $\forall y' \in T - \{x\}, \theta_{T,S}(i,x,y') = 1$
- $\gamma^{-}_{T,S}(i,x) = 0$
- $\forall y' \in T - \{x\}, \gamma^{-}_{T,S}(i,y') = 0$
- $\forall j \in I_{T,1}, j \neq i, p = T - B_j, \gamma^{+}_{T,S}(j,p) = 1$
- $\forall q \in T - \{p\}, \theta_{T,S}(j,p,q) = 1$

Après le mouvement, B_i est en situation *Rempart* et les B_j dans aucune des deux situations, donc on a :

- $cov(T,S) = 1$
- $\gamma^{+}_{T,S}(i,x) = 0$
- $\forall y' \in T - \{x\}, \theta_{T,S}(i,x,y') = 0$
- $\gamma^{-}_{T,S}(i,x) = 1$
- $\forall y' \in T - \{x\}, \gamma^{-}_{T,S}(i,y') = 1$
- $\forall j \in I_{T,1}, j \neq i, p = T - B_j, \gamma^{+}_{T,S}(j,p) = 0$
- $\forall q \in T - \{p\}, \theta_{T,S}(j,p,q) = 0$

En observant les changements avant et après, on déduit les variations du cas 1 données dans le tableau 3.2.

Notons qu'on représente en vert (ou en bleu pour B_j dans les figures 3.3 et 3.5)

le bloc en situation *Candidat* et l'élément associé (c'est à dire celui qu'il faut ajouter au bloc pour couvrir le t-subset T) et en rouge le bloc en situation *Rempart* ainsi que tous les éléments du t-subset concerné.

Les autres cas peuvent s'analyser de façon similaire. Le tableau 3.4 ci-après donne les situations des blocs par rapport à un t-subset T avant et après le mouvement. Les figures illustrant ces différentes situations sont données par la suite.

TABLEAU 3.4 Situation des blocs par rapport au t-subset T selon les différents cas énoncés

Cas	Bloc	Avant	Après	Figure
1	B_i B_j	C C	R 0	figure 3.3
2	B_l	R	0	figure 3.4
3	B_i B_j	R 0	C C	figure 3.5
4	B_l	0	R	figure 3.6
5	B_i	0	C	figure 3.7
6	B_i	C	0	figure 3.8
7	B_i	C	C	figure 3.9

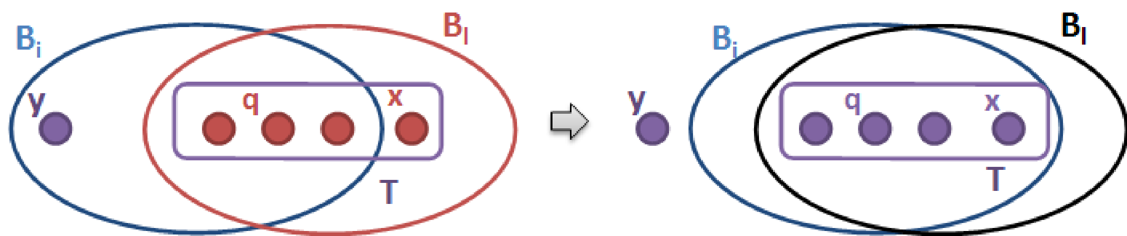

FIGURE 3.4 Cas 2 : Le bloc B_l quitte la situation *Rempart* suite à la couverture du t-subset T par le bloc B_i après le mouvement

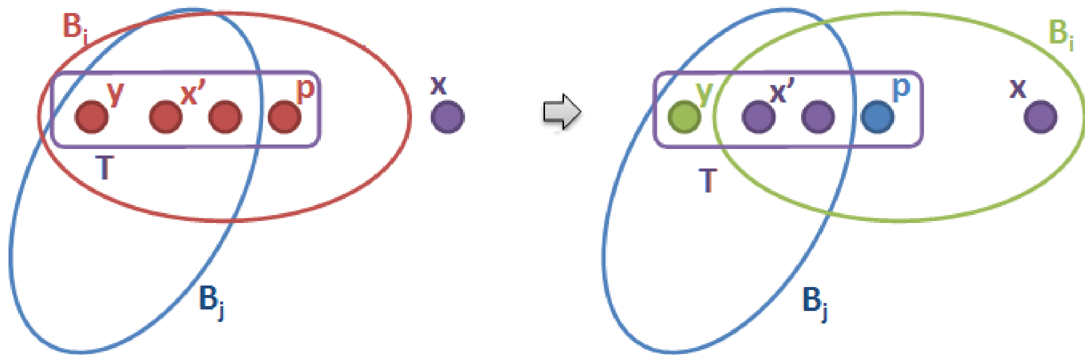

FIGURE 3.5 Cas 3 : Le bloc B_i en situation *Rempart* avant le mouvement passe en situation *Candidat* en découvrant le t-subset T

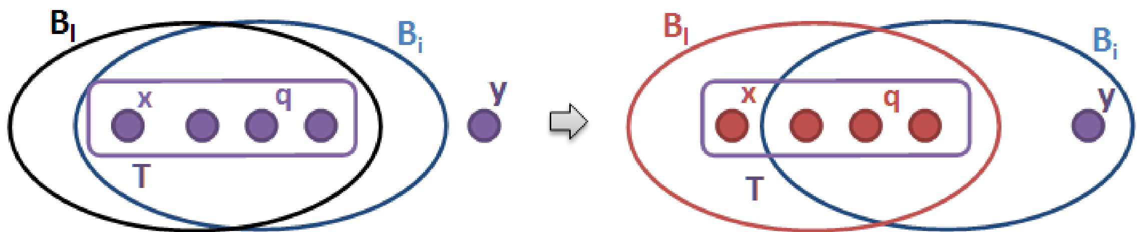

FIGURE 3.6 Cas 4 : Le bloc B_l passe en situation *Rempart* suite à la découverture du t-subset T par B_i

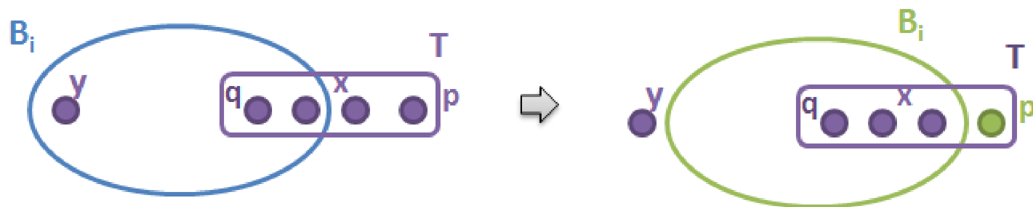

FIGURE 3.7 Cas 5 : Le mouvement fait entrer le bloc B_i en situation *Candidat* par rapport au t-subset T

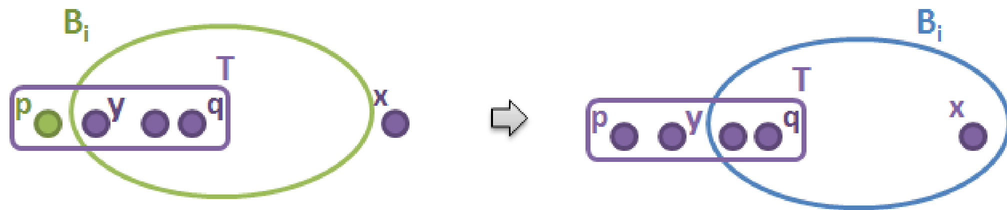

FIGURE 3.8 Cas 6 : Le mouvement fait sortir le bloc B_i de la situation *Candidat* par rapport au t-subset T

FIGURE 3.9 Cas 7 : Le mouvement modifie la situation *Candidat* du bloc B_i par rapport au t-subset T qui était candidat avec l'élément x avant le mouvement et qui est candidat avec l'élément y après

Un dernier point concerne la façon dont on trouve les t-subsets, par exemple ceux qui vérifient les conditions du cas 1 du tableau 3.2. En pratique, l'algorithme de mise à jour énumère tous les $(t-1)$-subsets contenus dans $(B_i - \{y\})$. Soit E un tel $(t-1)$-subset, on pose $T = E \cup \{x\}$. Si $cov(T, S) = 0$, alors on est sûr que le t-subset T vérifie toutes les conditions du cas 1 du tableau 3.2.

3.2.5 Indiçage des t-subsets

Un problème rencontré lors de l'implémentation consiste à trouver un moyen efficace d'indicer les t-subsets. Nous en avons besoin en particulier pour manipuler le tableau *cov* et nous assurer qu'il a une taille optimale.

Le problème consiste à trouver une bijection entre l'ensemble des t-subsets (sous-ensembles à t éléments de $\mathcal{V}$) et l'ensemble $\{1 \dots \binom{v}{t}\}$. Il faut pour cela ranger les t-subsets selon l'ordre lexicographique. Des algorithmes permettant de trouver un t-subset à partir de sa position ou le contraire peuvent être trouvés dans [36].

Dans nos algorithmes, nous cherchons la position de t-subsets donnés dans l'ensemble des t-subsets rangés en ordre lexicographique. Soit $(m_1, \dots, m_t)$ un t-subset (tel que $m_1 < \dots < m_t$) et i la position de ce t-subset, la formule donnant i est :

$$i = \sum_{j=1}^{k} \binom{m_j}{j} \tag{3.25}$$

Pour trouver la position d'un t-subset, il faut utiliser des coefficients binomiaux. Nous avons pour cela précalculé le triangle de pascal pour avoir tous les coefficients binomiaux $\binom{i}{j}$ avec i et j plus petits que v.

3.2.6 Implémentation efficace d'ensembles d'éléments

Dans nos algorithmes, les opérations les plus élémentaires et qui influencent grandement la rapidité d'exécution sont les opérations concernant les ensembles d'éléments tels que les blocs ou les t-subsets. Il arrive en effet fréquemment que l'algorithme teste si un élément ou un t-subset appartient à un bloc. Cela arrive dans la procédure de mises à jour pour tester la plupart des conditions des tableaux 3.2 et 3.3. Pour pouvoir faire des opérations efficaces, il est crucial d'utiliser une bonne représentation numérique des ensembles d'éléments.

Considérons un bloc qui contient donc k éléments. Une façon simple de représenter ce bloc serait par exemple de lui associer un vecteur de taille k dans lequel serait rangé chaque élément appartenant au bloc. Cependant, cette façon de faire n'est pas efficace. En effet pour savoir si un élément appartient à un bloc, l'algorithme est obligé de comparer l'élément en question avec les éléments contenus dans le vecteur représentant le bloc (opération qui se fait en $O(log(k))$ si on considère l'opération consistant à comparer deux nombres comme élémentaire).

Notre approche consiste à utiliser la représentation binaire des nombres qui peuvent être vus comme des vecteurs de bits. Par exemple, le bloc $(1, 2, 5)$ peut se représenter par le nombre 19 qui s'écrit 10011 en nombre binaire (on écrit $19_{10} = 10011_2$). De façon plus mathématique soit l le nombre représentant le bloc $(m_1, \ldots, m_k)$ alors :

$$l = \sum_{j=1}^{k} 2^{m_j - 1} \tag{3.26}$$

L'avantage de cette représentation est qu'elle accélère les tests d'appartenance grâce aux opérateurs logiques : pour savoir si un élément j appartient à un bloc B représenté par l, il suffit de considérer le nombre résultant d'un ET binaire entre ces deux nombres. Si $[(2^{j-1}\ ET\ l) = 0]$ est vérifiée alors $j \notin B$. De même pour tester si un bloc B représenté par l couvre un t-subset représenté par r, on teste si l'égalité $[(l\ ET\ r) = r]$ est vérifiée.

Nous pouvons maintenant décrire plus en détail les opérations consistant à savoir si l'indice d'un bloc appartient à $\mathcal{I}^0_{T,S}$, $\mathcal{I}^1_{T,S}$ ou $\mathcal{I}^2_{T,S}$. Soit B_i un bloc représenté par l et T un t-subset représenté par l. Savoir si $i \in \mathcal{I}^0_{T,S}$ consiste à savoir si B_i couvre T et a déjà été décrit précédemment. Tester si $i \in \mathcal{I}^1_{T,S}$ peut se faire en listant tous les éléments n'appartenant pas à B_i ; soit j un tel élément, il suffit de tester si T est couvert par $B_i \cup \{j\}$. Il suffit donc de tester si l'égalité $[((l + 2^{j-1})\ ET\ r) = r]$ est vraie. Comme il faut tester tous les éléments non contenus dans B_i et qu'on ne sait pas a priori si un élément appartient à B_i ou non, l'opération se fait en $O(v)$. Il faut tester si $i \in \mathcal{I}^2_{T,S}$ dans le cas 5 du tableau et ce cas uniquement. Notons que dans ce cas, nous considèrons le mouvement $< i_m, x_m, y_m >$ et nous avons une condition supplémentaire qui est $x_m \in T$. En particulier $x_m \notin B_i$. Ainsi pour savoir si $i \in \mathcal{I}^2_{T,S}$ il suffit de tester si $B_i \cup \{x_m\} \cup \{j\}$ couvre T où j est un élément différent de x_m n'appartenant pas à B_i. De même que précédemment, il suffit donc de considérer

tous les j possibles et de tester si l'égalité $[((l + 2^{x_m - 1} + 2^{j-1})\ ET\ r) = r]$ est vraie. L'opération se fait donc aussi en $O(v)$.

Une dernière remarque concerne le calcul de 2^{j-1}. 2^{j-1} est la représentation binaire de l'élément j (en effet si par exemple $j = 4$ alors $2^3_{10} = 8_{10} = 1000_2$). Le calcul de 2^{j-1} peut se faire simplement en utilisant l'opérateur de décalage de bits vers la gauche (<< en C++). Ainsi en C++, 2^{j-1} se calcule en utilisant l'expression $[1<<(j-1)]$ (on décale un bit $j - 1$ fois).

3.2.7 Pseudo-code des algorithmes de mise à jour

Nous donnons dans cette partie deux procédures pour faire des opérations ensemblistes et la procédure permettant de faire les mises à jour. La procédure *Union* prend en argument deux ensembles et renvoie l'union ensembliste de ces deux ensembles.

Procédure **Union**($ensemble1, ensemble2$)
  Renvoyer $ensemble1 + ensemble2$;

La procédure *Appartient* prend en argument deux ensembles et renvoie VRAI si le premier ensemble est inclus dans le second.

Procédure **Appartient**($ensemble1, ensemble2$)
  Si $[(ensemble1\ ET\ ensemble2) = ensemble1]$ *alors* Renvoyer $VRAI$;
  Sinon Renvoyer $FAUX$;

Pour implémenter ces deux procédures, on utilise les vecteurs de bits comme décrit dans la partie 3.2.6. Pour simplifier par la suite on notera $E_1 \cup E_2$ à la place de Union(E_1,E_2) et $E_1 \in E_2$ à la place de Appartient(E_1,E_2).

Nous décrivons à présent la procédure *MiseAJour* qui permet d'effectuer les mises à jour du tableau *cov* et des matrices γ^+, γ^- et θ comme décrit dans les parties 3.2.2 et 3.2.3.

Procédure **MiseAJour**(S, b, i, x, y)
  $\forall E$ sous-ensemble de $B_i - \{y\}$ à $(t-1)$-éléments *faire*
    $T_1 = E \cup \{x\}$;
    Si $cov(T_1) = 0$ *alors* //traitement du cas 1
      $\gamma^+(i, x) - -$;
      Pour $q = 1..v$, *si* $q \in T_1$ *faire*
        $\gamma^-(i, q) + +$;

```
        Si q ≠ x
            θ(i, x, q) − −;
        finSi
    finPour
    Pour j = 1..b, j ≠ i faire
        Pour p = 1..v, si p ∈ T_1 faire
            Si NON p ∉ B_j ET T_1 ∈ B_j ∪ {p} alors
                γ+(j, p) − −;
                Pour q = 1..v, si q ∈ T_1 ET q ≠ p faire
                    θ(j, p, q) − −;
                finPour
            finSi
        finPour
    finPour
finSi
Si cov(T_1) = 1 alors //traitement du cas 2
    Pour j = 1..b, si T_1 ∈ B_j faire
        Pour q = 1..v, si q ∈ T_1 faire
            γ−(j, q) − −;
        finPour
    finPour
finSi
cov(T_1) + +;
T_2 = E ∪ {y};
Si cov(T_2) = 1 alors //traitement du cas 3
    γ+(i, x) + +;
    Pour q = 1..v, si q ∈ T_2 faire
        γ−(i, q) − −;
        Si q ≠ y
            θ(i, y, q) + +;
        finSi
    finPour
    Pour j = 1..b, j ≠ i faire
        Pour p = 1..v, si p ∈ T_2 faire
            Si NON p ∉ B_j ET T_2 ∈ B_j ∪ {p} alors
                γ+(j, p) + +;
                Pour q = 1..v, si q ∈ T_2 ET q ≠ p faire
                    θ(j, p, q) + +;
                finPour
            finSi
        finPour
```

finPour
finSi
Si $cov(T_1) = 2$ *alors* //traitement du cas 4
Pour $j = 1..b$, *si* $T_2 \in B_j$ *faire*
Pour $q = 1..v$, *si* $q \in T_2$ *faire*
$\gamma^-(j, q) + +$;
finPour
finPour
finSi
$cov(T_2) - -$;
$\forall E'$ sous-ensemble de $B_i - \{y\}$ à $(t-2)$-éléments *faire*
Pour $p = 1..v$, *si* $p \notin E' \cup \{x\} \cup \{y\}$ *faire*
$T_3 = E' \cup \{x\} \cup \{p\}$; //traitement du cas 5
Si $cov(T_3) = 0$ *alors*
$\gamma^+(i, p) + +$;
Pour $q = 1..v$, *si* $q \in T_3$ ET $q \neq p$ *faire*
$\theta(i, p, q) + +$;
finPour
finSi
$T_4 = E' \cup \{y\} \cup \{p\}$; //traitement du cas 6
Si $cov(T_4) = 0$ *alors*
$\gamma^+(i, p) - -$;
Pour $q = 1..v$, *si* $q \in T_3$ ET $q \neq p$ *faire*
$\theta(i, p, q) - -$;
finPour
finSi
finPour
$T_5 = E' \cup \{x\} \cup \{y\}$; //traitement du cas 7
Si $cov(T_5) = 0$ *alors*
$\gamma^+(i, x) - -$
Pour $q = 1..v$, *si* $q \in T_5$ ET $q \neq p$ *faire*
$\theta(i, x, q) - -$;
finPour
$\gamma^+(i, y) + +$
Pour $q = 1..v$, *si* $q \in T_5$ ET $q \neq p$ *faire*
$\theta(i, y, q) + +$;
finPour
finSi

Chapitre 4

RÉSULTATS EXPÉRIMENTAUX

Dans cette partie, nous présentons l'ensemble des résultats obtenus lors de nos expérimentations. Celles-ci ont consisté dans un premier temps à mesurer le gain obtenu lorsqu'on utilise les structures de données que nous avons conçues. Puis nous présentons les performances de notre algorithme TS-CD lorsque nous l'avons testé sur les jeux de données présents dans les archives de La Jolla Covering Repository [22]. Ensuite nous étudions comment se comporte notre algorithme tabou avec diversification sur un petit nombre de cas choisis. Enfin, nous présentons les résultats obtenus par notre algorithme mémétique.

4.1 Améliorations due aux structures de données

Nous avons voulu évaluer l'efficacité de nos structures de données en comparant nos algorithmes de bas niveau à ceux proposés précédemment par Nurmela puis par Dai et al. Nous avons pour cela implémenté un algorithme tabou identique au nôtre, mais utilisant les anciennes structures de données (celles développées par Nurmela [31]). Nous comparons à la fois la taille mémoire nécessaire à la mise en œuvre de chacune des implémentations et la vitesse d'exécution en temps CPU. La taille mémoire a été calculée tandis que le temps CPU a été mesuré en utilisant les mêmes conditions (même algorithme de haut niveau, mêmes ressources matérielles et même compilateur pour chacune des implémentations).

Pour rappel, dans l'implémentation de Nurmela les structures de données principales sont les suivantes :

- La matrice contenant pour chaque bloc les numéros de ses voisins : il y a $\binom{v}{k}$ blocs, chacun ayant $k(v-k)$ voisins soit au total $\binom{v}{k} \times k(v-k)$ pour la taille de cette matrice.
- La matrice contenant pour chaque bloc les numéros des t-subsets qu'il couvre : un bloc couvre $\binom{k}{t}$ t-subsets donc la taille de cette matrice est $\binom{v}{k} \times \binom{k}{t}$.
- Le vecteur contenant pour chaque t-subset le nombre de bloc qui le couvre : ce vecteur est de taille $\binom{v}{t}$ qui est le nombre de t-subsets.

Nos structures de données contiennent :

- Le vecteur *cov* qui contient pour chaque t-subset le nombre de bloc qui le couvre et qui est de taille $\binom{v}{t}$.
- La matrice γ^+ de taille $b \times (v-k)$
- La matrice γ^- de taille $b \times k$
- La matrice θ de taille $b \times v \times v$

Étant donné que chaque entier est enregistré sur 4 octets, les tailles totales des structures de données sont données par les formules suivantes :

$$Size_1 = 4/1024\left(\binom{v}{k} \times (v-k) + \binom{v}{k} \times \binom{k}{t} + \binom{v}{t}\right) \tag{4.1}$$

$$Size_2 = 4/1024\left(b \times v + b \times v \times v + \binom{v}{t}\right) \tag{4.2}$$

Où $Size_1$ et $Size_2$ désignent les tailles en ko des structures de données de Nurmela et des nôtres respectivement.

Nous avons comparé les deux implémentations dans le tableau ci-après pour différents jeux de données. Les 4 premières colonnes donnent les caractéristiques des jeux de données (v, k, t et b). Les colonnes 5 et 6 donnent le temps CPU en secondes nécessaire pour faire 1000 itérations. La colonne 5 donne les temps de calcul pour l'algorithme tabou utilisant les structures de données de Nurmela tandis que la colonne 6 donne les temps de calcul de notre algorithme tabou utilisant les structures de données que nous proposons. Les tests ont été effectués sur une machine Intel Xeon @ 2.2Ghz avec 8Go de mémoire vive. Les programmes ont été écrits en C++ (compilés avec g++ avec l'option O3). La colonne 7 donne l'accélération obtenue avec les nouvelles structures de données. Il s'agit du rapport entre la valeur qui se trouve dans la colonne 6 et celle de la colonne 5. Pour plusieurs jeux de données, notre machine n'a pas pu exécuter l'algorithme avec l'implémentation de Nurmela faute d'espace mémoire. Pour ces jeux de données, il n'y a pas de valeur dans les colonnes 5 et 7. Les colonnes 8 et 9 indiquent la taille mémoire en ko des principales structures de données nécessaires à l'exécution de chaque implémentation (colonne 8 pour celle de Nurmela et colonne 9 pour la nôtre). Ces nombres sont calculés grâce aux formules données précédemment pour $Size_1$ et $Size_2$. Nous avons vérifié sur 3 jeux de données que les nombres calculés correspondaient bien à la taille mémoire effectivement utilisée par la machine.

D'après ces données, le temps d'une exécution de 10^6 itérations varie pour notre implémentation entre moins de 4 minutes (pour $(v, k, t) = (12, 5, 3)$) et plus de 48 heures (pour $(v, k, t) = (25, 16, 8)$). Sur les jeux que peut traiter l'autre implémentation, cela irait théoriquement jusqu'à 58 jours (pour $(v, k, t) = (22, 14, 7)$). Nous observons par ailleurs que l'accélération obtenue par nos algorithmes par rapport à ceux de Nurmela varie entre 5 et 95 sur les jeux de données testés. L'accélération est faible (inférieure à 10) pour seulement les deux jeux de données les plus petits. De plus, l'accélération a tendance à augmenter lorsque la taille des jeux de données augmente (i.e. la taille de v, t et b).

Nous remarquons que la quantité de mémoire nécessaire à l'exécution de l'algorithme avec les structures de données de Nurmela explose : pour $(v, k, t) = (30, 16, 5)$ l'algorithme aurait théoriquement besoin de 2.6 To de mémoire pour s'exécuter (contre 778 ko pour notre implémentation). Tous les jeux de données qui nécessitent plus de 8 Go de mémoire n'ont d'ailleurs pas pu être exécutés. Nos structures de données utilisent l'espace mémoire de façon plus raisonnable et n'ont jamais eu besoin de plus de 5 Mo sur les jeux de données testés.

En résumé, les données présentées ci-dessus montrent que l'implantation que nous avons proposée ne pose pas de problème d'explosion de la taille de la mémoire, contrairement à la meilleure technique

Tableau 4.1 Comparaison entre deux implémentations : celle de Nurmela (Impl.1) et la nôtre (Impl.2) pour différents jeux de données

Jeu de données				Temps cpu pour 1000 itérations			Espace mémoire (en ko)	
v	**k**	**t**	**b**	Impl. 1	Impl. 2	Accélération	Impl. 1	Impl. 2
12	5	3	28	1.04	0.22	**5**	140	18
12	7	5	58	5.10	0.55	**9**	176	38
13	9	6	38	11.42	0.99	**12**	342	34
16	9	6	168	99.05	4.26	**23**	6600	210
18	12	8	205	857.01	16.20	**53**	41287	445
20	9	5	219	315.02	9.30	**34**	147682	420
20	12	7	185	1178.37	39.15	**30**	437261	606
21	11	6	233	1074.47	24.16	**44**	788312	632
22	14	7	133	5020.98	64.74	**78**	4427482	929
23	7	4	252	148.86	12.61	**12**	140808	578
23	12	4	30	169.61	1.79	**95**	3311569	99
23	12	5	80	756.34	10.54	**72**	4880288	304
23	16	8	150	-	96.82	**-**	12434057	2239
24	11	3	19	47.36	0.80	**59**	3003181	52
24	12	5	85	-	18.21	**-**	9887236	365
25	16	8	297	-	174.95	**-**	103860774	4979
27	16	7	232	-	170.70	**-**	591597954	4154
30	11	4	143	-	13.62	**-**	115016180	627
30	16	5	61	-	16.70	**-**	2608519789	778

connue jusqu'à présent (l'implémentation proposée par Nurmela [31]). De plus, elle est généralement de 10 à 100 fois plus rapide. La technique proposée constitue donc un progrès considérable par rapport aux techniques existantes. Elle pourra être utilisée avec profit par ceux qui implanteront dans l'avenir des algorithmes de recherche locale pour le problème du Covering Design.

4.2 Résultats obtenus avec l'algorithme TS-CD

Dans cette partie, nous présentons les résultats obtenus avec notre algorithme TS-CD. Nous avons testé tous les jeux de données tels que $v \leq 32$ et $t \leq 8$ avec k suffisamment grand pour avoir un nombre de blocs b plus petit que 300. De fait, c'est plus de 700 jeux de données qui ont été testés.

Au cours des expérimentations, nous avons utilisé les paramètres suivants :
$iter_{max} = 10^6$, $iter_{div} = 10^3$, $iter_{reg} = 2 \times 10^5 - 10^3$. Nous avons choisi de prendre $iter_{max} = 10^6$ car cela permet à une exécution de durer moins de 30 minutes pour la plupart des jeux de données. Pour $lgtl_0$, nous avons testé les valeurs 2, 3 et 4. Étant donné que la longueur de liste taboue varie beaucoup, une de ces trois valeurs est adéquate considérant les jeux de données testés. Nous avons

fixé le paramètre b_0 à $UB - 1$, où UB représente la meilleure borne supérieure connue au moment où nous avons fait nos expériences, afin de chercher directement des coverings meilleurs que ceux actuellement existant. Nous avons effectué 10 exécutions pour chaque jeu de données testé.

Ces expérimentations nous ont permis de trouver de nouveaux records pour 75 jeux de données, dont 70 toujours valides (voir le tableau en Annexe). Chaque ligne du tableau correspond à un jeu de données. La première colonne donne les valeurs des paramètres v, k et t pour le jeu de données considéré. La quatrième colonne (notée “LB”) donne la meilleure borne inférieure connue. La cinquième colonne (“UB”) indique la borne supérieure connue avant que nous commencions nos expériences. Les deux colonnes suivantes donnent la nouvelle borne supérieure trouvée par notre algorithme (“new UB”) et la différence entre la nouvelle borne et l’ancienne (“diff.”). Les deux colonnes qui suivent (“nb.succ.”) indiquent combien d’exécutions (parmi 10) ont conduit à une amélioration de la précédente borne supérieure (colonne 9) et combien ont pu atteindre la nouvelle borne (colonne 8). Enfin, la dernière colonne (“$lgtl_0$”) affiche la valeur du paramètre $lgtl_0$ utilisée pour atteindre le résultat présenté ici. Par exemple pour $(v, k, t) = (28, 11, 4)$ (avant dernière ligne de la première page du tableau en annexe), l’ancienne borne supérieure valait $UB = 108$. Notre algorithme a donc été exécuté 10 fois avec $b_0 = UB - 1 = 107$. Les meilleures solutions trouvées par l’algorithme contiennent 104 blocs, ce qui constitue une amélioration de 4 unités. Pour ce jeu de données, l’algorithme est parvenu à trouver 3 fois une solution constituée de 104 blocs ; 2 autres exécutions ont conduit à une solution légale composée de 105, 106 ou 107 blocs ; tandis que les 5 exécutions restantes ont échoué dans la recherche d’une solution avec 107 blocs.

Nous observons dans le tableau en annexe que les bornes supérieures ont été améliorées de 1 unité dans 29 cas, de 2 dans 20 cas, de 3 dans 13 cas, de 4 dans 4 cas, de 5 dans 2 cas, de 6 dans 2 cas et de 9 ou plus dans 3 cas. Notre algorithme a par exemple permis d’améliorer un des anciens records de 20 unités (pour $(v, k, t) = (30, 10, 4)$). On remarque aussi que notre algorithme n’a été en mesure d’obtenir qu’une seule amélioration (pour 10 exécutions) dans un peu moins de la moitié des cas.

En plus des test décrits ci-dessus, nous avons réalisé des tests complémentaires en utilisant d’autres techniques. En particulier, un record a été battu grâce à la méthode des filons (c.f. partie 4.4.4). Il s’agit du jeu de données $(30, 11, 4)$ (situé sur la deuxième page du tableau en annexe).

Jusqu’à présent, les meilleurs résultats obtenus en utilisant des métaheuristiques, étaient ceux utilisant l’algorithme MLTS [10]. L’algorithme MTLS avait été utilisé pour traiter des jeux de données tels que $v \leq 20$, $t \leq 8$ et $b \leq 200$. Il a permis de trouver des nouveaux records pour 38 jeux de données dont 23 sont encore d’actualité. Si nous analysons le tableau en annexe, nous remarquons que notre algorithme a été efficace pour deux catégories de jeux de données. La première contient 69 jeux de données et correspond aux plus grandes valeurs de v ($20 \leq v \leq 32$) et aux petites valeurs de t ($3 \leq t \leq 6$) : ces jeux de données correspondent aux 69 premières lignes du tableau. Notons que ces jeux de données (sauf les trois pour lesquels $v = 20$) n’ont pas pu être traités avec l’algorithme MLTS à cause de la valeur trop grande de v. La seconde catégorie correspond à 6 jeux de données pour lesquels v est assez petit ($15 \leq v \leq 20$) et la valeur de t est grande ($t = 7$ *ou* 8). De plus, nous pouvons remarquer que notre algorithme n’est pas efficace pour de grandes valeurs de b avec seulement deux records battus pour $b \geq 150$.

Il est remarquable de constater que notre algorithme est arrivé à battre des records pour des jeux de données assez petits comparés à l'ensemble des jeux de données qu'on peut trouver sur le site de Gordon [22] ($v \leq 32$ alors que La Jolla contient des jeux de données pouvant aller jusqu'à $v = 99$ et $b < 300$ alors qu'il existe des coverings contenant plusieurs milliers de blocs). Ces jeux sont donc plus faciles à traiter mais plus difficile à améliorer, car ils ont été vraisemblablement beaucoup plus testés par les gens qui s'intéressent au problème que les gros jeux dans l'archive.

Nous n'avons pas comparé nos résultats à ceux obtenus avec les autres métaheuristiques. Les résultats présentés par Nurmela et Östergård sont anciens et assez peu pertinents (les jeux de données traités sont petits et peu nombreux). Quant aux résultats présentés par Dai et al., il nous manque un certain nombre de renseignements pour pouvoir faire une comparaison qui a du sens. En effet, Dai et al. nous donnent des plages de paramètres pour l'ensemble des jeux de données plutôt que des paramètres spécifiques pour chaque jeu de données. Nous ne connaissons que la meilleure valeur de la fonction de coût trouvée pour un b donné pour l'ensemble des jeu de données. Dai et al. précisent pour les jeux de données pour lesquels ils ont trouvé une amélioration un nombre d'itérations et un temps CPU, mais nous ne savons pas si ce sont des moyennes ou les meilleurs résultats trouvés, ni sur quelle machine chaque résultat a été trouvé.

4.3 Expérimentations effectuées avec l'algorithme mémétique

Nous présentons ici les tests que nous avons effectués avec l'algorithme mémétique. Nous nous sommes intéressés à deux aspects : savoir lequel des trois croisements que nous avons proposés est le plus efficace et savoir si en réglant correctement les paramètres nous pouvions faire mieux que l'algorithme tabou avec diversification.

Nous avons tenté de comparer le nombre de réussites pour 10 exécutions sur chacun des 5 jeux de données. Les jeux de données ont été choisis de façon à n'être ni trop faciles (nous ne voulons pas que toutes les techniques réussissent 10 fois) ni trop difficiles (nous ne voulons pas non plus qu'elles échouent 10 fois). Les jeux de données ont été pris parmi ceux qui ont été améliorés par notre algorithme tabou. Afin de ne pas introduire de biais, nous avons aussi relancé notre algorithme TS-b-CD sur les jeux de données et nous présentons pour celui-ci nos derniers résultats.

4.3.1 Comparaison des différents croisements proposés

Nous avons comparé les trois croisements dans les mêmes conditions à savoir avec les paramètres suivants : $taille_{pop} = 5$, $nb_{gen} = 25$ et $iter_{max} = 40000$.

Les résultats sont présentés dans le tableau 4.2 ci-après :

- Chaque ligne correspond à un croisement (X1, X2 et X3).
- Chaque colonne correspond à un jeu de données présenté par ses paramètres sous la forme (v, k, t) et b le nombre de blocs pour lequel nous cherchons une solution (nous rappelons qu'on résout ici le problème de décision consistant à trouver une solution pour un b fixé).

- Dans chaque case du tableau, nous indiquons le nombre de fois où une solution est trouvée sur un total de 10 essais.

TABLEAU 4.2 Nombre de solutions trouvées par chacun des croisement pour 10 exécutions sur différents jeux de données

(v, k, t) : b	$(16, 9, 6)$ 169	$(30, 15, 5)$ 94	$(24, 15, 5)$ 36	$(27, 11, 4)$ 95	$(28, 8, 3)$ 73
X1	0	4	0	0	0
X2	1	5	0	1	1
X3	0	9	0	0	0

Il nous apparait que le meilleur croisement est X2. Pour les 5 jeux de données, X1 n'a jamais fait mieux que X2 et n'a trouvé que des $(30, 15, 5)$-covering designs (4 sur 10 contre 5 pour X2). X3 a fait mieux que X2 sur ce même jeu de données en trouvant une solution 9 fois sur 10, mais n'a rien trouvé non plus pour les autres jeux de données. X2 a en effet réussi à trouver au moins une solution pour 4 jeux de données sur les 5 essayés.

4.3.2 Réglages des paramètres avec le croisement X2 et comparaison avec l'algorithme tabou avec diversification

Dans cette nouvelle série d'expériences, nous avons testé différentes combinaisons de valeurs pour $taille_{pop}$, nb_{gen} et $iter_{max}$. Nous présentons donc le tableau ci-après où :

- Chaque colonne correspond à un jeu de données comme précédemment.
- La deuxième ligne donne les résultats de notre algorithme de décision TS-b-CD pour 10^6 d'itérations. Nous avons relancé les tests, car nous avons constaté un biais par rapport aux résultats originaux (nous avons en effet choisi les jeux de données, spécifiquement à cause de la bonne performance de l'algorithme tabou exécuté la première fois sur ces jeux).
- Les lignes suivantes correspondent toutes à des exécutions de l'algorithme mémétique avec le croisement X2 et des jeux de paramètres différents. Nous les dénotons dans le tableau par un triplet de la forme
  $(taille_{pop}, nb_{gen}, iter_{max}/1000)$.
- Dans chaque case du tableau, nous indiquons le nombre de fois où une solution est trouvée sur un total de 10 essais chaque.

Notons enfin que pour que la comparaison soit équitable, nous avons attribué le même nombre d'itérations que pour l'algorithme tabou à toutes les exécutions. Ainsi $nb_{gen} \times iter_{max} = 10^6$ pour toutes les exécutions.

Les résultats de ce tableau ne montrent pas vraiment de jeu de paramètres particulièrement plus efficace que les autres ou que l'algorithme tabou. Si nous nous intéressons, pour chaque ligne, au nombre de jeux de données pour lesquels nous avons trouvé au moins une solution, alors seul le jeu de paramètres $(4, 25, 40)$ a réussi à faire aussi bien que l'algorithme tabou en réussissant pour 4

TABLEAU 4.3 Tests avec le croisement X2 pour différents jeux de paramètres

(v,k,t) : b	$(16,9,6)$ 169	$(30,15,5)$ 94	$(24,15,5)$ 36	$(27,11,4)$ 95	$(28,8,3)$ 73
TS-b-CD	1	6	1	0	1
$(10,100,10)$	0	7	1	0	2
$(4,25,40)$	2	8	1	0	1
$(5,12,83)$	0	7	0	0	0
$(5,50,20)$	0	8	0	2	0
$(10,25,40)$	0	9	0	0	1
$(10,50,20)$	1	4	0	0	1

jeux de données. Ce jeu de paramètres est d'ailleurs le seul qui a réussi à faire mieux que tabou sur tous les jeux de données, bien que sa domination ne soit pas écrasante. D'autres jeux de paramètres arrivent à faire ponctuellement mieux que $(4,25,40)$ ou tabou, mais les résultats sont généralement très proches les uns des autres.

Considérant ces résultats, il apparait que l'algorithme mémétique avec le croisement proposé n'est pas la solution miracle, même si ses résultats semblent un peu meilleurs que ceux de l'algorithme tabou avec diversification.

4.4 Analyse du comportement de l'algorithme

Dans cette partie, nous présentons et analysons des profils d'exécution. Un profil d'exécution donne une idée du comportement de l'algorithme en montrant comment évolue la fonction de coût au cours de la recherche. Nous avons grâce à cela mis en évidence des phénomènes apparaissant lors de l'exécution de notre algorithme tabou. Nous caractérisons en particulier les différentes manifestations du phénomène de stagnation. Le phénomène de stagnation se caractérise par le fait que la fonction de coût se maintient autour d'une valeur donnée malgré une amélioration possible (i.e. la fonction de coût peut encore diminuer). Nous montrons en particulier pour certains jeux de données une évolution inattendue de la fonction de coût. Certains profils présentent en effet un phénomène de chute qui se caractérise par une diminution importante et brusque de la fonction de coût après une stagnation qui a lieu à une valeur élevée de la fonction de coût et que nous appelons “hyperstagnation”.

4.4.1 Profil d'exécution

Un profil d'exécution est une courbe reliant des points dont les coordonnées sont en abscisse le numéro d'itération et en ordonnée la valeur de la fonction de coût pour la configuration présente à l'itération considérée.

Nous présentons ci-dessous un profil d'exécution typique pour un algorithme de recherche locale sur un problème de satisfaction de contraintes. La fonction de coût, dans ce cas, correspond au

nombre de contraintes violées. L'objectif est donc de réduire ce nombre à zéro en satisfaisant les contraintes.

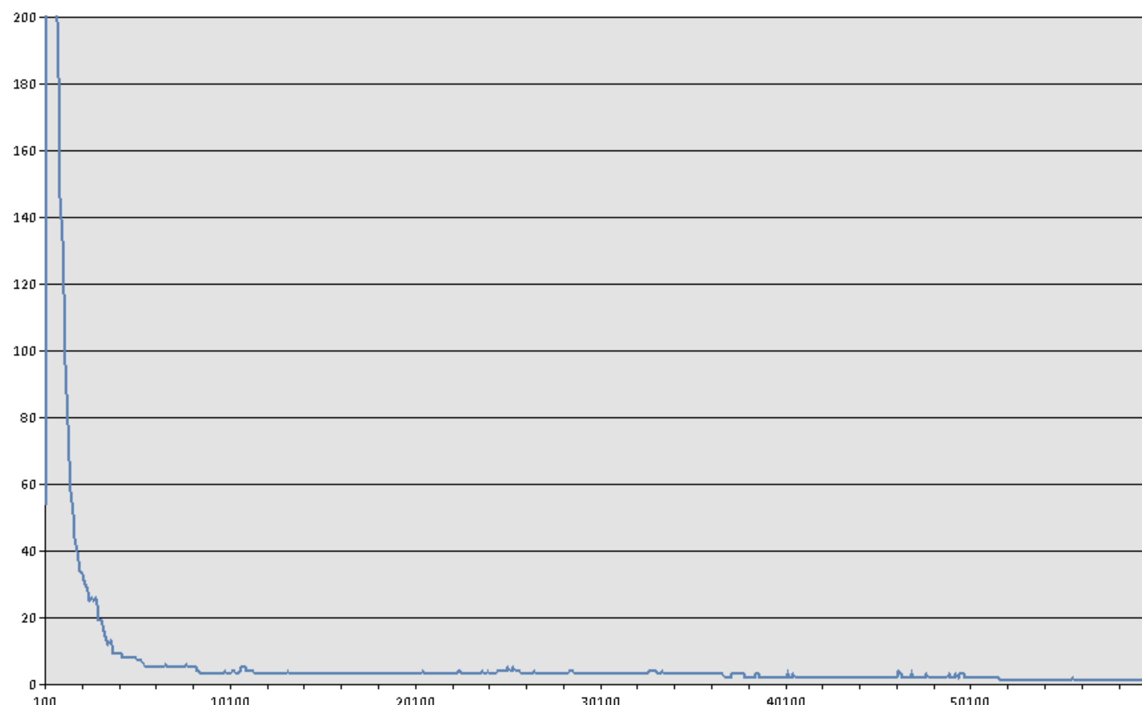

FIGURE 4.1 Exemple d'un profil typique d'un algorithme de recherche local pour un problème de satisfaction de contraintes

Nous observons ici une décroissance rapide au départ. Ce comportement s'explique aisément car il y a beaucoup de contraintes violées au début de la recherche et qu'il est donc facile d'en satisfaire un certain nombre. À mesure que le nombre de contraintes violées diminue, il devient difficile de satisfaire celles qui restent. C'est pour cela que la courbe diminue moins rapidement (stagne presque) pour finalement obtenir une solution. C'est ce genre de courbe que nous nous attendions à obtenir pour notre problème.

4.4.2 Observation du phénomène de stagnation

Nous observons le profil d'exécution de notre algorithme tabou sans diversification.

Nous observons qu'au tout début de la recherche lors des 10000 premières itérations ($x < 10$), la fonction de coût diminue de façon très rapide pour atteindre une valeur stationnaire autour de 10. La recherche semble alors rester dans la même région de l'espace des configurations tout au long de l'exécution. Nous disons que la recherche stagne car nous savons que cet exemplaire est réalisable (il est possible de trouver une configuration à coût nul). Notons que ce phénomène peut être beaucoup

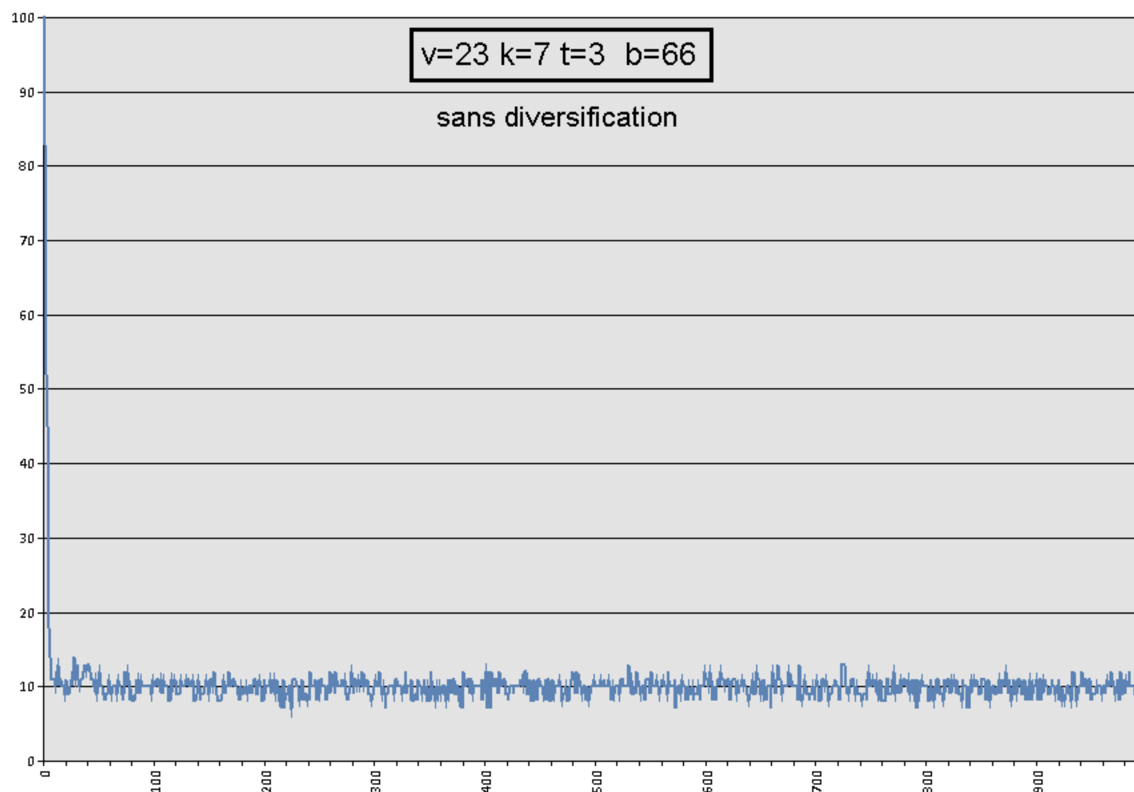

FIGURE 4.2 Profil d'exécution de l'algorithme tabou sans diversification

plus marqué et qu'on peut stagner à des niveaux plus élevés (qu'on observera par la suite). Lors de la stagnation, il ne sert à rien de continuer. C'est pourquoi nous faisons appel à un mécanisme de diversification.

Comparons la courbe précédente à la courbe de la même exécution, mais cette fois avec le mécanisme de diversification.
Nous pouvons déjà faire plusieurs observations concernant cette courbe.

Nous pouvons facilement observer l'effet du mécanisme de diversification qui se déclenche toutes les 200000 itérations. Nous observons, aux abscisses 200, 400, 600 et 800, une augmentation de la fonction de coût qui monte jusqu'à une valeur d'environ 80. Après la phase de diversification (de 1000 itérations), la fonction de coût diminue plus ou moins brutalement (en moyenne ici sur 40000 itérations) pour atteindre à nouveau des valeurs stationnaires.

Entre deux pics dus à la diversification, la valeur de f reste en général stationnaire. Cette valeur change après chaque pic de diversification.

On observe ici que la stagnation a lieu autour de valeurs relativement faibles. Ce jeu de données peut être qualifié de relativement facile (un jeu vraiment facile est un jeu pour lequel notre algorithme trouve une solution rapidement et presque systématiquement).

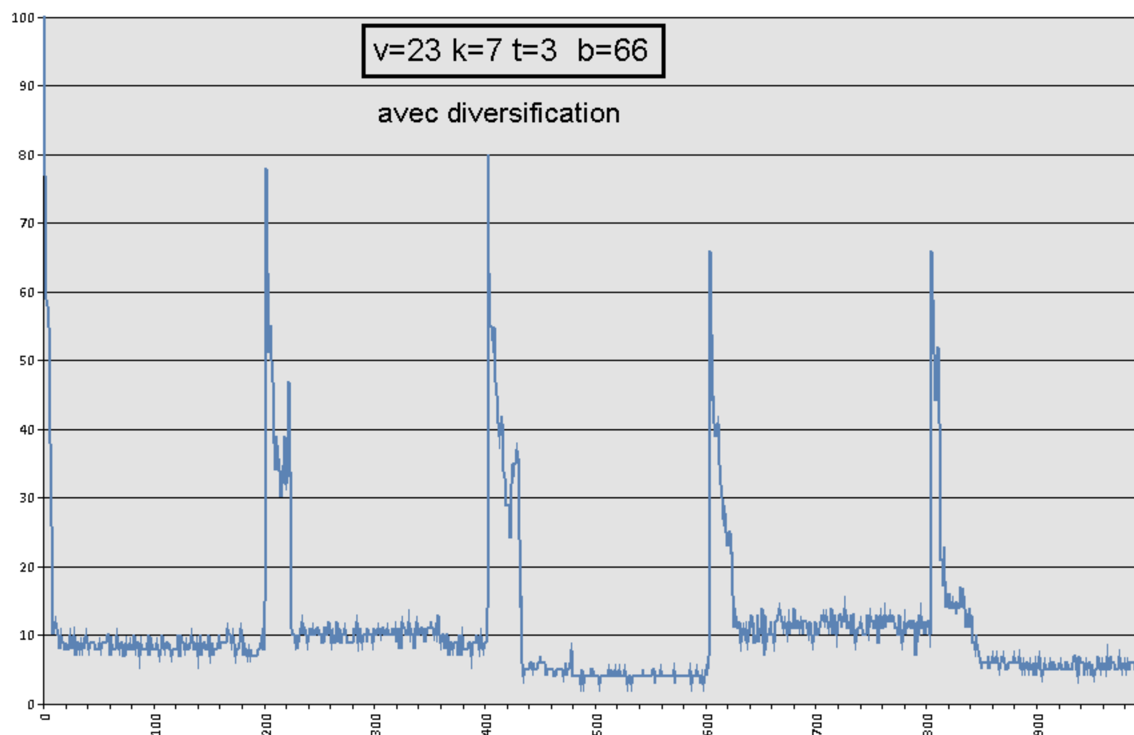

Figure 4.3 Profil d'exécution de l'algorithme tabou avec diversification

Nous présentons ici le profil d'une exécution pour laquelle on peut observer très nettement le phénomène de stagnation. La fonction de coût se stabilise en effet très vite.

Nous observons ici une évolution normale de la fonction de coût lors des 100000 premières itérations (pour $x < 100$). La diversification à l'abscisse 200 ne semble pas avoir eu d'effet sur la recherche, car la fonction de coût stagne au même niveau avant et après. Notons que la phase de descente après la diversification est ici très courte. La fonction de coût reste quasiment constante à une valeur d'environ 40 jusqu'à l'abscisse 400. Après la diversification qui a lieu à l'abscisse 400, la recherche parvient à atteindre une région où la fonction de coût est plus petite (environ égale à 20) et stagne à ce niveau. La diversification ayant lieu à l'abscisse 600 dégrade la recherche en l'amenant dans une zone où la fonction de coût stagne à une valeur de 230. Enfin, la dernière diversification permet de faire redescendre la fonction de coût en dessous de 70. À la fin de la recherche, il ne semble pas y avoir de stagnation.

On voit bien, sur cette figure, que le phénomène de stagnation peut se produire à différents paliers : on distingue trois valeurs autour desquelles la fonction de coût stagne : une à 40 ($100 < x < 400$), une à 20 ($400 < x < 600$) et une à 230 ($600 < x < 800$). Il est clair qu'on passe d'un palier de stagnation à un autre grâce au mécanisme de diversification. La procédure de diversification parfois améliore, parfois dégrade la recherche. Nous constatons ici que le mécanisme de diversification n'est pas adapté à ce jeu de données. Dans cet exemple en effet, on peut reprocher à la diversification

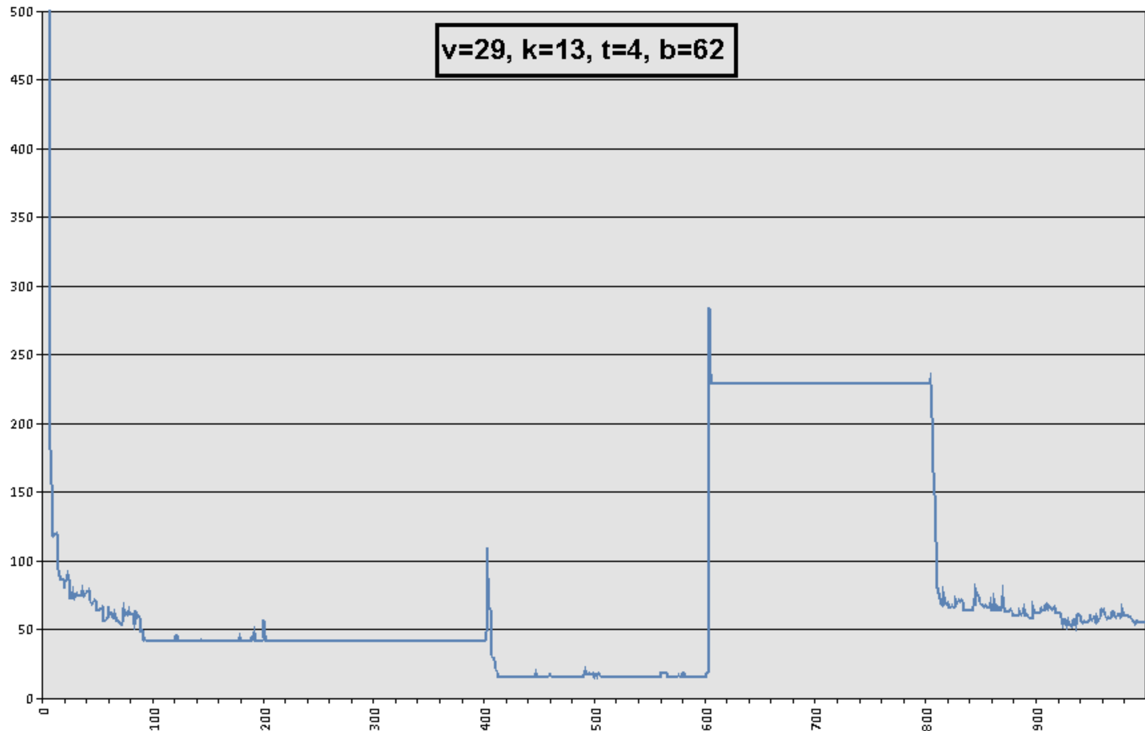

FIGURE 4.4

de se déclencher trop tard et donc de laisser la recherche stagner trop longtemps. Cependant, étant donné que le comportement varie énormément d'un jeu de données à l'autre, il est très difficile de trouver un réglage de la diversification qui conviendrait à tous les jeux de données.

4.4.3 Identification du phénomène d'hyperstagnation avec chute

Nous avons remarqué un phénomène assez particulier sur certains jeux de données que nous appelons phénomène d'hyperstagnation avec chute. Le phénomène se caractérise par une stagnation de la fonction de coût autour d'une valeur très élevée (ce que nous appelons "hyperstagnation") puis d'une diminution brusque de la fonction de coût (que nous appelons "chute") pour stagner à un palier beaucoup plus bas. Nous illustrons ce phénomène par les 10 profils d'exécutions effectuées avec le jeu de données $(v, k, t, b) = (21, 12, 6, 127)$ que voici.

Nous observons ici que pour 9 exécutions sur les 10 effectuées la fonction de coût décroit très rapidement d'une valeur palier située entre 1400 et 1500 jusqu'à une valeur plus petite que 300. La chute survient à différents moments (entre les abscisses 280 et 840) et il faut parfois attendre beaucoup avant de la voir arriver. La dernière exécution (couleur vert clair) présente une fonction

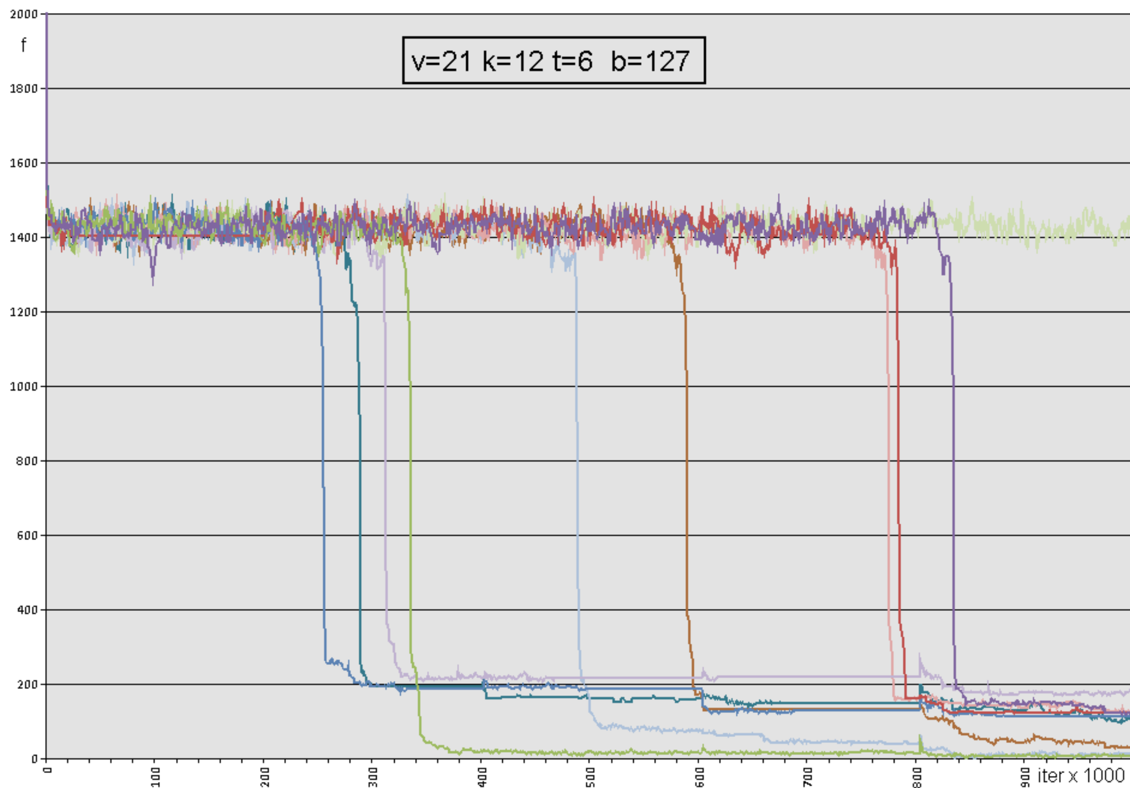

FIGURE 4.5 Hyperstagnation avec chute

de coût qui ne descend jamais en dessous de 1300 (il n'y a pas de chute).

Notons que pendant la période où l'exécution est dans sa phase d'hyperstagnation (i.e. lorsque la fonction de coût stagne autour de 1400, avant la chute) la diversification semble n'avoir aucun effet sur la recherche. D'abord, la diversification ne fait pas remonter la fonction de coût. Ensuite, la diversification ne permet pas de sortir de l'hyperstagnation.

En effet, ce phénomène ne semble pas être lié à la diversification : nous voyons par exemple 3 courbes effectuer leur descente entre les abscisses 280 et 360 alors que les diversifications ont lieu aux abscisses 200 et 400.

Après la chute, f stagne autour d'une valeur comprise entre 0 et 300 selon les exécutions. Il semble qu'à ce moment, la recherche est dans une phase de stagnation simple comme celle observée dans la partie précédente. On peut de plus constater que la diversification fait à nouveau effet sur la recherche.

Ce phénomène est une version plus grave de la stagnation simple. Il faut attendre la chute pour retourner à un comportement habituel.

Le phénomène observé touche cependant assez peu de jeu de données. Les jeux de données sont souvent des jeux pour lesquels il est difficile de trouver une solution.

4.4.4 Identification du phénomène d'hyperstagnation avec descente lente

Nous présentons les exécutions effectuées sur le jeu de données $(v, k, t) = (30, 10, 4)$ et $b = 230$.

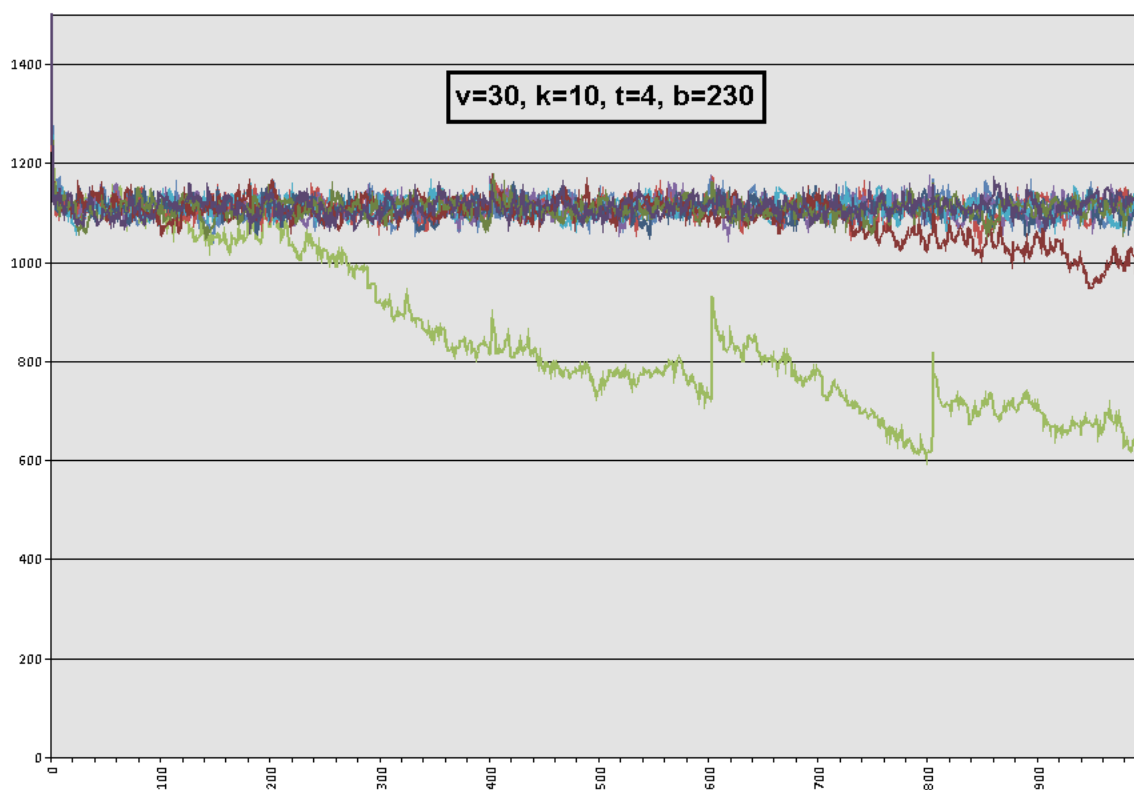

FIGURE 4.6 Hyperstagnation avec descente lente

On observe le phénomène d'hyperstagnation seulement pour toutes les courbes sauf deux (courbes en marron et en vert clair). La fonction de coût stagne autour de la valeur 1100. Pour ces courbes, il ne semble y avoir aucune évolution de la recherche. Les deux dernières courbes montrent une décroissance lente de la fonction de coût. La décroissance de la première courbe a lieu pour $x > 120$ et celle de la seconde pour $x > 720$. La décroissance de la fonction de coût est lente. En outre les phases de diversification font remonter la fonction de coût. Nous observons une descente progressive, mais pas de stagnation.

Le phénomène d'hyperstagnation sans chute n'est ni plus ni moins le même que celui présenté juste avant sans le phénomène de chute. Notons que ce cas est un cas aggravé du précédent. D'abord la descente est plus rare (par exemple ici elle correspond à 2 cas sur 9). Et de plus, lorsque la descente se produit elle se produit beaucoup plus lentement.

Idéalement, il faudrait laisser faire l'opérateur tabou et ne pas déclencher de diversification tant

que la fonction de coût diminue. Le phénomène de descente nous a donné une nouvelle idée que nous appelons "méthode des filons". Au lieu d'initialiser la recherche avec une configuration aléatoire, on relance la recherche en partant d'une configuration qu'on trouve à une itération prise pendant la période où la fonction de coût diminue. La méthode des filons nous a permis de battre un record (voir partie 4.2).

4.4.5 Phénomène d'hyperstagnation pure

Nous montrons ici le cas le plus grave du phénomène, c'est-à-dire celui dans lequel aucune descente ne se produit. Les courbes sont très proches les unes des autres. Aussi par soucis de lisibilité nous ne présentons que trois profils d'exécutions pour $v = 20$, $k = 13$, $t = 8$ et $b = 262$.

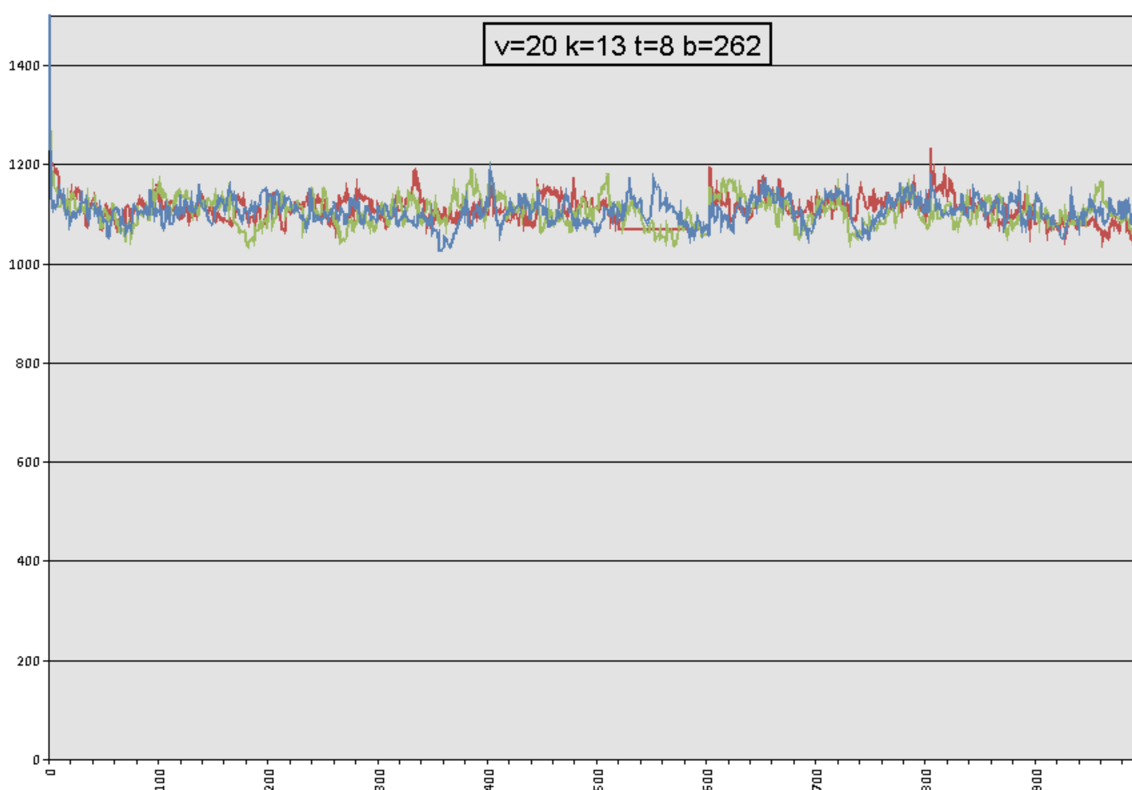

FIGURE 4.7 Hyperstagnation pure

On ne voit que de l'hyperstagnation pour les trois exécutions. L'algorithme est totalement impuissant.

Si on récapitule, la stagnation a lieu en général pour des jeux de données difficiles. Nous avons observé un phénomène de stagnation simple, pour lequel chaque phase de diversification entraine une perturbation, suivie d'une descente puis de stagnation. Nous avons observé un phénomène d'hyper-

stagnation avec chute : la fonction de coût stagne à une valeur élevée, puis diminue brutalement pour ensuite revenir à un comportement habituel de stagnation simple. Nous avons ensuite vu l'hyperstagnation avec descente lente : on observe de l'hyperstagnation et pour certaines exécutions (rarement) une décroissance lente de la fonction de coût. Enfin nous avons observé des cas d'hyperstagnation pure, pour lesquels la fonction de coût ne diminue jamais et qui semblent être des cas désespérés.

4.4.6 Comportement de la procédure de haut niveau qui résout le problème d'optimisation

Nous observons ici ce qu'il se passe avec la procédure d'optimisation lorsqu'une solution de coût nul est trouvée. Dans ce cas, l'algorithme retire un bloc de la solution trouvée (celui qui permet de découvrir le moins de t-subsets), et transmet la configuration résultante (contenant un bloc de moins) à la procédure TS-b-CD avec le paramètre b décrémenté (voir la description de l'algorithme dans la partie 3.1.3).

Nous montrons sur la figure ci-après à la fois la fonction f (représentée par la courbe bleue) et la fonction b (représentée par la courbe rouge) qui donne la valeur de b au cours de la recherche. Les deux fonctions partagent le même graphe, mais ne sont pas à la même échelle. En plus de l'échelle pour f en noir, nous indiquons en rouge l'échelle pour b.

La courbe bleue, qui représente f, décrit une décroissance progressive sur 800000 itérations jusqu'à la valeur 0.

La courbe rouge décrit la décroissance de b. Nous voyons que b reste constant jusqu'à l'abscisse 855, moment où une solution est trouvée pour $b = 230$. Plusieurs solutions pour des valeurs de b comprises entre 230 et 225 sont rapidement trouvées entre les abscisses 855 et 865. La fonction de coût reste alors à une valeur de b de 225 entre les abscisses 865 et 1000. Au moment où une solution est trouvée pour $b = 225$, nous observons à nouveau une décroissance de la courbe. Des solutions pour b compris entre 225 et 219 sont trouvées en l'espace de 1000 itérations. On arrive par la suite rapidement à une solution pour $b = 213$ autour de l'abscisse 1020 (remarquons que nous avons amélioré la solution de 12 unités en l'espace de quelques milliers d'itérations). L'exécution trouve ensuite des solutions pour $b = 212$ et $b = 211$ autour des abscisses 1120 et 1770 respectivement. Une dernière remarque concernant f : à partir du moment où l'algorithme trouve la première solution jusqu'à la fin de l'exécution, la valeur de f ne remonte jamais au dessus de 40. L'exemple présenté illustre un phénomène important : il est efficace de continuer la recherche quand l'algorithme a trouvé une solution plutôt que de repartir d'une solution aléatoire qui risque de conduire à un échec avec une forte probabilité.

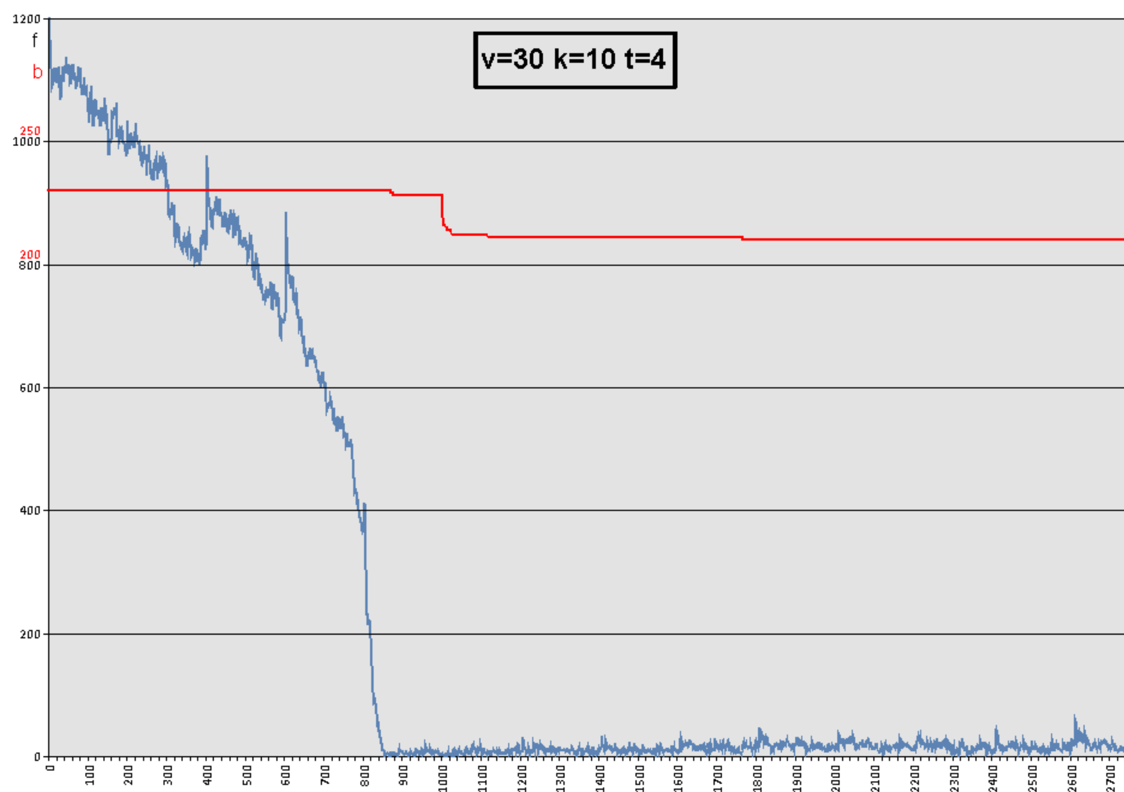

FIGURE 4.8 Mise en évidence du mécanisme qui décrémente b en repartant de la configuration trouvée

Chapitre 5

CONCLUSION

Nous résumons ici le travail que nous avons effectué et les contributions que nous avons apportées. Enfin nous présentons les limitations des solutions proposées et les voies vers lesquelles il pourrait être intéressant d'aller pour des améliorations futures.

5.1 Synthèse des travaux

Notre travail a consisté à concevoir et implémenter des métaheuristiques permettant de résoudre le problème de Covering Design. Nous avons développé un algorithme tabou efficace. Afin de contrer le phénomène de stagnation, nous avons muni notre mécanisme tabou d'une longueur de liste variable. Cela n'était pas suffisant, c'est pourquoi nous avons ensuite combiné notre mécanisme tabou à un mécanisme de diversification permettant périodiquement de changer la zone d'exploration de l'algorithme dans l'espace de recherche. L'algorithme TS-CD intégrant à la fois l'opérateur tabou et l'opérateur de diversification n'a permis de résoudre que partiellement le problème de stagnation. Nous avons donc inséré notre opérateur tabou dans un algorithme mémétique. Nous avons développé trois croisements pour cet algorithme afin de trouver une alternative au mécanisme de diversification. Pour accélérer nos algorithmes, nous avons conçu et implémenté de nouvelles structures de données.

Nous avons effectué des expérimentations afin de tester nos algorithmes. Nous avons comparé nos structures de données aux structures précédemment proposées par Nurmela et Östergård [31] et également utiisées par Dai et al. [10]. Nous avons ensuite testé nos algorithmes sur plus de 700 jeux de données correspondant à tous les jeux vérifiant $v \leq 32$, $3 \leq t \leq 8$ et $b \leq 300$. Nous avons comparé les résultats obtenus aux meilleurs résultats connus. Nous avons effectué des tests pour mesurer la performance de notre algorithme mémétique et des croisements que nous avons proposés. Enfin, nous avons analysé le comportement de notre algorithme à travers l'analyse de plusieurs profils d'exécution. Nous avons ainsi mis en évidence le phénomène de stagnation et nous avons montré qu'il se manifeste sous plusieurs formes que nous avons décrites.

5.2 Principales contributions

La contribution la plus importante de ce mémoire est constituée de nos nouvelles structures de données et les algorithmes de bas niveau que nous avons conçus et implémentés pour le problème de Covering Design. Nos nouvelles structures de données sont très économiques en taille mémoire et rendent nos algorithmes de bas niveau très rapides. Par rapport aux algorithmes de bas ni-

veau précédents, nos algorithmes sont entre 10 et 100 fois plus rapides sur les jeux de données testés. En outre, l'accélération augmente quand la taille des jeux de données augmente. Nos structures de données et nos algorithmes de bas niveau pourraient constituer le nouveau standard pour les méthodes de recherche locale dédiées au problème de Covering Design. Les practiciens en métaheuristiques pourront en effet s'appuyer sur ces structures de données pour développer de nouvelles heuristiques ou accélérer les anciennes.

Notre algorithme TS-CD a obtenu de très bons résultats sur les jeux de données testés. Dai et al. s'étaient félicités en 2005 d'avoir battu 38 records. Nous avons obtenu 77 records et nous en possèdons toujours 71, ce qui constitue environ un record pour 10 jeux de données essayés.

5.3 Limitations et perspectives

Nous avons remarqué plusieurs limitations dans les algorithmes proposés. D'abord, il y a le fait que nous soyons limités par le nombre v d'éléments ($v \leq 32$). Cela est dû au fait que nous utilisons des entiers comme vecteurs de bits et que les plus grands entiers utilisables en C++ ont 32 bits. Il serait possible de se passer de l'utilisation de vecteurs de bits, mais cela créerait vraisemblablement un ralentissement important de notre algorithme. Une façon de contourner le problème en continuant d'utiliser des vecteurs de bits serait de passer le code en langage C ce qui permettrait en utilisant une machine 64 bits de pouvoir agir sur les jeux de données tels que $v \leq 64$.

En outre, notre algorithme tabou ne fonctionne pas bien lorsque les valeurs de t ou b sont grandes (quand $t > 6$ ou $b > 150$). De façon générale, nos algorithmes ne fournissent pas un comportement satisfaisant au vu des profils d'exécutions. Nous observons en effet un phénomène de stagnation voire d'hyperstagnation lorsque la taille des jeux est très grande. Ni notre opérateur de diversification, ni notre algorithme mémétique ne semblent fournir une solution véritablement satisfaisante au problème de stagnation.

La diversification telle que nous l'avons faite est criticable et pourrait vraisemblablement être améliorée. Notre algorithme appelle la procédure de diversification à intervalles fixes. Dans certains cas, la diversification est tardive et l'algorithme perd du temps à stagner trop longtemps. Dans d'autres cas, la diversification intervient alors que la recherche est dans une phase de descente. Il serait plus efficace d'avoir un système qui déclenche le mécanisme de diversification de façon dynamique. On pourrait par exemple observer périodiquement la décroissance de la fonction de coût afin de déterminer le moment où la recherche commmence à stagner et déclencher la diversification à ce moment.

Références

[1] AARTS, E. et LENSTRA, J. K., éditeurs (1997). *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

[2] BATTITI, R. et TECCHIOLLI, G. (1994). The reactive tabu search. *INFORMS Journal on Computing*, 6, 126–140.

[3] BERTOLO, R., BLUSKOV, I. et HÄMÄLÄINEN, H. (2004). Upper bounds on the general covering number $c_\lambda(v, k, t, m)$. *J. Combin. Designs*, 362–380.

[4] BLUM, C. et ROLI, A. (2003). Metaheuristics in combinatorial optimization : Overview and conceptual comparison. *ACM Comput. Surv.*, 35, 268–308.

[5] BLUSKOV, I. et HÄMÄLÄINEN, H. (1998). New upper bounds on the minimum size of covering designs. *J. Combin. Designs*, 21–41.

[6] CERNY, V. (1985). Thermodynamical approach to the travelling salesman problem. *J. Optimization Theory and Applications*, 45, 41–51.

[7] CHAN, A. H. et GAMES, R. A. (1981). (n,k,t)-covering systems and error-trapping decoding. *IEEE Trans Inf Theory*, 643–646.

[8] COTTA, C., SEVAUX, M. et SÖRENSEN, K., éditeurs (2008). *Adaptive and Multilevel Metaheuristics*, vol. 136 de *Studies in Computational Intelligence*. Springer.

[9] CRESCENZI, P., MONTECALVO, F. et ROSSI, G. (2004). Optimal covering designs : complexity results and new bounds. *Discrete Applied Mathematics*, 281–290.

[10] DAI, C., LI, P. C. et TOULOUSE, M. (2006). A cooperative multilevel tabu search algorithm for the covering design problem. *Artificial Evolution*. Springer-Verlag, vol. 3871 de *LNCS*, 119–130.

[11] DORIGO, M., CARO, G. D. et GAMBARDELLA, L. M. (1999). Ant algorithms for discrete optimization. *Artificial Life*, 5, 137–172.

[12] EIBEN, A. et SCHIPPERS, C. (1998). On evolutionary exploration and exploitation. *Fundamenta Informaticae*, 1–16.

[13] ETZION, T., WEI, V. et ZHANG, Z. (1995). Bounds on the sizes of constant weight covering codes. *Designs Codes and Cryptography*, 217–239.

[14] FEO, T. et RESENDE, M. (1995). Greedy randomized adaptive search procedures. *J. Global Optim.*, 109–133.

[15] FOGEL, L. J. (1962). Toward inductive inference automata. *IFIP Congress*. 395–400.

[16] GALINIER, P. et HAO, J.-K. (1999). Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim.*, 3, 379–397.

[17] GALINIER, P. et HAO, J.-K. (2004). A general approach for constraint solving by local search. *Journal of Mathematical Modelling and Algorithms*, 73–88.

[18] GAREY, M. et JOHNSON, D. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, San Francisco.

[19] GLOVER, F. (1989). Tabu search - part i. *INFORMS Journal on Computing*, 1, 190–206.

[20] GLOVER, F. (1990). Tabu search - part ii. *INFORMS Journal on Computing*, 2, 4–32.

[21] GLOVER, F. et LAGUNA, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Boston, USA.

[22] GORDON, D. M. (2009). http ://www.ccrwest.org/cover.html.

[23] GORDON, D. M., KUPERBERG, G. et PATASHNIK, O. (1995). New constructions for covering designs. *J. Combin. Designs*, 269–284.

[24] HANSEN, P., MLADENOVIC, N. et MORENO-PÉREZ, J. A. (2008). Variable neighbourhood search : methods and applications. *4OR*, 6, 319–360.

[25] HOLLAND, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan press.

[26] KIRKPATRICK, S., JR., D. G. et VECCHI, M. P. (1983). Optimization by simmulated annealing. *Science*, 220, 671–680.

[27] LI, P. C. et VAN REES, G. H. J. (2000). New constructions for lotto designs. *Utilitas Mathematica*, 45–64.

[28] MARGOT, F. (2003). Small covering designs by branch-and-cut. *Mathematical Programming*, 207–220.

[29] METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A. et TELLER, E. (1953). Simulated annealing. *J. Chem. Phys.*, 1087–1092.

[30] MILLS, W. H. et MULLIN, C. (1992). Covering and packings. *Contemporary Design Theory : A Collection of Surveys*, Wiley-Interscience Series in Discrete Mathematics and Optimization. 371–399.

[31] NURMELA, K. J. et ÖSTERGåRD, P. R. J. (1993). Constructing covering designs by simulated annealing. Rapport technique, Helsinki University of Technology.

[32] PITSOULIS, L. et RESENDE, M. (2002). Greedy randomized adaptive search procedures. P. Pardalos et M. Resende, éditeurs, *Handbook of Applied Optimization*, Oxford University Press. 178–183.

[33] RAMALHINHO-LOURENÇO, H., MARTIN, O. C. et STÜTZLE, T. (2000). Iterated local search. Economics Working Papers 513, Department of Economics and Business, Universitat Pompeu Fabra.

[34] SCHÖNHEIM, J. (1964). On coverings. *Pacific Journal of Mathematics*, 1405–1411.

[35] SORIANO, P. et GENDREAU, M. (1996). Diversification strategies in tabu search algorithms for the maximum clique problem. *Annals of Operations Research*, 189–207.

[36] STANTON, D. W. et WHITE, D. E. (1986). *Constructive Combinatorics*. Springer-Verlag, New York.

ANNEXES

Annexe 1 : Améliorations obtenues avec notre algorithme TS-CD

v	k	t	LB	UB	new UB	diff.	nb. succ.		$lgtl_0$	battu
22	9	3	27	29	28	-1	9	9	2	N
23	7	3	63	67	66	-1	1	1	2	O
26	8	3	59	65	64	-1	2	2	2	N
27	8	3	65	70	69	-1	3	3	2	N
27	10	3	33	36	35	-1	5	5	3	N
27	16	3	9	11	10	-1	1	1	3	N
28	8	3	70	76	73	-3	2	4	2	N
28	11	3	28	32	31	-1	1	1	3	N
30	8	3	83	97	94	-3	1	1	2	O
31	9	3	66	74	73	-1	1	1	3	N
32	9	3	72	78	77	-1	1	1	3	N
32	14	3	19	23	22	-1	1	1	2	N
23	10	4	63	77	74	-3	1	7	3	N
24	10	4	70	89	87	-2	1	4	2	N
24	11	4	46	59	57	-2	2	2	3	O
25	10	4	80	98	97	-1	3	3	3	N
25	11	4	55	70	66	-4	1	5	2	N
25	14	4	22	27	26	-1	10	10	3	N
26	10	4	89	119	118	-1	1	1	2	N
27	10	4	103	138	129	-9	1	1	3	N
27	11	4	72	97	95	-2	2	4	2	N
27	12	4	54	65	63	-2	1	1	3	N
27	15	4	22	27	26	-1	7	7	3	N
28	11	4	84	108	104	-4	3	5	2	N
28	12	4	59	79	76	-3	1	1	2	N

v	k	t	LB	UB	new UB	diff.	nb. succ.		$lgtl_0$	battu
28	13	4	39	55	54	-1	5	5	2	N
29	11	4	90	118	115	-3	1	2	2	N
29	13	4	47	63	59	-4	1	4	3	N
29	16	4	22	28	27	-1	9	9	3	N
30	10	4	165	231	211	-20	1	1	4	O
30	11	4	112	144	133	-11	1	1	3	N
30	12	4	73	102	96	-6	1	7	2	N
30	14	4	35	50	48	-2	2	8	3	N
31	12	4	86	115	112	-3	1	3	3	N
31	14	4	47	62	60	-2	1	2	3	N
31	17	4	22	30	28	-2	1	3	2	N
31	18	4	19	25	24	-1	6	6	2	N
31	19	4	15	20	19	-1	5	5	2	N
31	20	4	13	17	16	-1	6	6	2	N
32	13	4	72	101	96	-5	1	4	3	N
32	15	4	39	52	49	-3	1	3	3	N
32	17	4	27	30	29	-1	2	2	2	N
32	19	4	16	24	22	-2	1	2	2	N
20	10	5	90	108	105	-3	1	3	3	N
20	11	5	50	65	64	-1	1	1	3	N
20	13	5	24	33	31	-2	1	1	3	N
21	12	5	42	56	54	-2	3	6	3	N
21	13	5	25	38	37	-1	1	1	3	N
22	14	5	24	34	33	-1	9	9	3	N
23	13	5	43	56	54	-2	1	4	3	N

v	k	t	LB	UB	new UB	diff.	nb. succ.		$lgtl_0$	battu
23	14	5	32	44	42	-2	1	5	2	N
23	15	5	23	30	29	-1	4	4	2	N
24	13	5	50	67	66	-1	7	7	2	N
24	15	5	24	38	36	-2	3	7	3	N
25	15	5	32	47	44	-3	1	10	2	N
25	16	5	24	35	32	-3	4	5	2	N
26	13	5	76	103	102	-1	1	1	3	N
26	15	5	39	54	52	-2	2	7	3	N
27	16	5	34	49	46	-3	1	4	2	N
28	16	5	39	54	52	-2	1	3	3	N
29	16	5	46	61	59	-2	1	1	3	N
30	15	5	68	96	94	-2	3	10	3	N
31	17	5	50	62	61	-1	1	1	3	N
31	18	5	35	55	52	-3	1	3	3	N
32	18	5	40	61	56	-5	1	7	3	N
32	19	5	32	48	47	-1	1	1	3	N
21	12	6	88	128	127	-1	1	1	3	N
23	14	6	56	93	91	-2	1	2	2	N
26	16	6	52	84	80	-4	1	1	3	N
15	9	7	220	276	270	-6	1	1	3	N
15	10	7	78	112	108	-2	1	1	3	N
16	11	7	62	83	80	-3	3	3	3	N
19	13	7	56	81	77	-4	2	2	3	N
17	12	8	61	132	130	-2	1	1	2	O
20	14	8	80	125	119	-6	1	1	3	N