Titre: Title:	Heuristiques efficaces pour le problème de partitionnement de graphe
Auteur: Author:	Ziad Boujbel
Date:	2007
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Boujbel, Z. (2007). Heuristiques efficaces pour le problème de partitionnement de graphe [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7958/

UNIVERSITÉ DE MONTRÉAL

HEURISTIQUES EFFICACES POUR LE PROBLÈME DE PARTITIONNEMENT DE GRAPHE

ZIAD BOUJBEL
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2007

Your file Votre référence
ISBN: 978-0-494-29215-0
Our file Notre référence
ISBN: 978-0-494-29215-0

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

HEURISTIQUES EFFICACES POUR LE PROBLÈME DE PARTITIONNEMENT DE GRAPHE

présenté par: BOUJBEL Ziad
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

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

Remerciement

Je tiens tout d'abord à exprimer ma gratitude envers mon directeur de recherche M. Philippe Galinier d'avoir accepté de m'encadrer et de m'avoir conseillé et soutenu tout au long de ce projet. Je le remercie aussi pour l'aide financière qu'il m'a octroyée.

Je remercie également M. Michel Gagnon et M. Alejandro Quintero d'avoir accepté de faire partie de mon jury.

Je tiens enfin à remercier mes parents, mes proches et mes amis de leur encouragement et soutien permanent.

Résumé

Ces dernières années, le problème de partitionnement de graphe a suscité l'attention d'un grand nombre de chercheurs, vu l'accroissement de ses applications dans différents domaines (le calcul scientifique, le design *VLSI*, l'ordonnancement, etc.). Le problème consiste à partitionner un graphe en portions de tailles approximativement égales tout en minimisant le nombre d'arêtes coupées.

Le problème de partitionnement de graphe est NP-difficile. Les graphes dans ce problème atteignent plusieurs centaines de milliers de sommets (voire plusieurs millions), ce qui augmente sa complexité. Plusieurs heuristiques ont été développées dans le but de trouver une approximation raisonnable. La plupart de ces heuristiques sont basées sur le concept multi-niveaux. Ce concept consiste à réduire la complexité du problème en diminuant itérativement la taille du graphe jusqu'à l'obtention d'un graphe très réduit. Sur le graphe le plus réduit, on produit une partition initiale que l'on projette ensuite sur le graphe original.

Dans ce mémoire, nous présentons des heuristiques pour le problème de partitionnement de graphes. Suite à différents tests et comparaisons avec les meilleurs résultats connus sur des graphes de référence, nos algorithmes ont montré une grande facilité à produire de meilleures partitions que les algorithmes les plus performants connus.

Abstract

These last years, the graph partitioning problem attracts the attention of many of researchers considering the increase of its applications in various domains. The problem consists in partitioning the vertices of a graph in roughly equal parts, such that the number of edges connecting vertices from different partitions is minimized.

The graph partitioning problem is NP-hard. The graph size in this problem reaches several hundreds of thousand of vertices (even several millions), thus making it a very complex problem. Several heuristics has been developed to find it a reasonable approximation. The majority of these heuristics are based on the multilevel approach. The basic idea of this approach consists in reducing the complexity of the problem by decreasing, in an iterative way, the size of the graph. We produce an initial partition of the most reduced graph which we then project on the original graph.

In this report, we present new heuristics, for the graph partitioning problem. After various comparisons with the best known results of reference graphs, we show that our algorithms produce better partitions than the one produced by the most powerful algorithms found in the literature.

Table des matières

Liste des tableaux

Liste des figures

Liste des algorithmes

Liste des sigles et abréviations

VLSI	Very-Large-Scale Integration
FPGA	Field-programmable gate array
HDL	Hardware description language
RBC	Récursif bissection coordonnées
RBI	Récursif bissection inertielle
RBS	Récursif de bissection spectrale
KL	Kernighan and Lin
KL1	Algorithme de Kernighan et Lin avec une seule itération
CPM	Couplage de poids maximum
Tabu-GP	Tabu graph partitioning
Tabu-GPG	Guided tabu graph partitioning
MA-GP	Memetic algorithm graph partitioning

Chapitre 1. INTRODUCTION

Le partitionnement de graphe est un problème important qui comporte de nombreuses applications dans différents domaines tels que le calcul scientifique, le design *VLSI*, l'ordonnancement, etc. Le problème consiste à partitionner les sommets d'un graphe en classes de tailles approximativement égales de telle sorte que le nombre d'arêtes qui relient des sommets appartenant à différents classes soit minimisé. Le partitionnement de graphe est un problème connu NP-difficile (Garey et al.,1976). Pour partitionner les graphes qui ont plusieurs milliers de sommets, les algorithmes heuristiques représentent la seule solution viable. Ces dernières années, plusieurs algorithmes d'optimisation (heuristiques) ont été conçus afin d'essayer de trouver des approximations raisonnables à ce problème. La plupart de ces algorithmes sont basées sur le concept multi-niveaux. L'idée de base de ce concept consiste à réduire la complexité du problème en diminuant de façon itérative la taille du graphe à partitionner jusqu'à atteindre un graphe réduit, dont la taille n'excède pas quelques centaines de sommets. Sur le graphe le plus réduit, on produit une partition initiale que l'on projette ensuite de proche en proche sur les graphes intermédiaires, jusqu'à obtenir une partition du graphe original. À chaque graphe intermédiaire, on applique souvent un algorithme de recherche locale de type Kernighan & Lin (1970) pour raffiner la partition projetée.

Le concept multi-niveaux a montré son efficacité à produire des bonnes partitions dans des laps de temps très court. Par exemple, pour un graphe bien connu de la littérature, nommé *auto* (nombre de sommets est égal à 448695), l'algorithme multi-niveaux *k*-Metis (Karypis et al., 1998) n'a besoin que de 39,67 secs pour partitionner le graphe en 256 classes. Cependant de nos jours, les meilleures résultats trouvés sur un ensemble de graphes de référence (Walshaw et al., 2006), sont obtenus par des algorithmes beaucoup plus longs comme JOSTLE Evolutionary (Soper et al. 2004) et PMSATS (Banos et al., 2004). Ces algorithmes ont besoin de beaucoup de temps pour produire leurs partitions. Pour partitionner les plus grands graphes de référence

(Walshaw et al., 2006), PMSATS a besoin de plusieurs heures de calcul et Jostle Evolutionary a besoin de plusieurs jours.

Il faut bien noter que dans le cas des algorithmes de partitionnement de graphes, il y a deux catégories différentes d'algorithmes : la catégorie des algorithmes rapides (comme *k*-Metis) et la catégorie des algorithmes longs (comme PMSATS). En effet, selon les applications et les objectifs mis aux points par les concepteurs, un algorithme peut avoir comme but ultime la qualité de la solution, alors qu'un autre peut avoir comme priorité de limiter le temps d'exécution. Les algorithmes qui se préoccupent principalement de la qualité se permettent de prendre le temps qu'il faut pour produire des solutions de haute qualité. Au contraire de ces derniers, les algorithmes rapides se permettent de limiter la qualité des partitions en vu de préserver une grande rapidité d'exécution.

La tendance actuelle pour résoudre le problème de partitionnement de graphe est basée principalement sur le concept multi-niveaux. Certains auteurs (Banos et al., 2004) vont même jusqu'à dire que c'est "la tendance actuelle de l'état de l'art". Sans nier l'efficacité de l'approche multi-niveaux, la tendance actuelle nous parait assez étrange. En effet, pratiquement aucun algorithme, parmi les plus performants, ne se base simplement sur les algorithmes traditionnels de la métaheuristique (une famille d'algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficiles pour lesquels on ne connaît pas de méthode classique plus efficace). Pourtant, les algorithmes traditionnels ont montré une grande efficacité à s'adapter aux différents problèmes et à produire des solutions de bonne qualité pour de nombreuses applications. Certains auteurs (Banos et al., 2004) ont la tendance de croire qu'à cause des grandes tailles de graphes dans le problème de partitionnement, il est pratiquement impossible d'explorer efficacement l'espace de recherche. Cette idée reçue doit être remise en question lorsqu'on sait que pour un problème assez proche du partitionnement de graphes (le coloriage de graphes), les algorithmes traditionnels ont montré une grande efficacité à produire des bonnes solutions dans des temps raisonnables (Hertz et al., 1987 ; Galinier et al., 1999). La ressemblance primordiale entre le coloriage et le

partitionnement est que dans les deux problèmes les solutions sont des partitions (Par contre différence est que les deux problèmes utilisent des fonctions d'évaluation différentes).

Le défi de notre recherche est de concevoir des heuristiques compétitives pour le problème de partitionnement de graphes sans avoir recourt à l'approche multi-niveaux. En se basant simplement sur les concepts de la métaheuristique, nous allons ainsi mettre au point des algorithmes bien adaptés au problème de partitionnement de graphes. Nos algorithmes doivent concurrencer les algorithmes longs comme JOSTLE Evolutionary (Soper, Walshaw & Cross, 2004) et PMSATS (Banos et al., 2004) qui détiennent de nombreux records sur les graphes de référence (Walshaw et al., 2006).

Pour développer des algorithmes pour le problème de partitionnement, nous allons principalement nous inspirer du problème de coloriage de graphes. La catégorie d'algorithme qui s'est démarquée par son efficacité, dans le coloriage de graphes, est celle des algorithmes hybrides (Galinier et al., 1999).

Dans le cadre de notre recherche, nous allons principalement proposer trois algorithmes pour le problème de partitionnement de graphes. Le premier est un algorithme tabou qui s'inspire largement de l'algorithme TabuCol (Hertz et al., 1987) conçu pour le problème de coloriage. Ce type d'algorithme essaye d'améliorer une solution initiale en effectuant des mouvements qui minimisent une fonction objective. Le deuxième est un algorithme qui utilise deux fonctions d'évaluations différentes : une fonction pour améliorer les solutions et une autre pour guider la recherche à long terme. Le troisième est un algorithme hybride qui s'inspire lui aussi d'un algorithme de coloriage (Galinier et al., 1999). Ce type d'algorithme fait évoluer un ensemble (population) de solutions. Il utilise une fonction de recombinaison qui génère des solutions à partir de ceux de la population et une fonction de recherche locale qui améliore les solutions générées. La recombinaison cherche à créer des partitions avec de nouvelles potentialités dans la génération future et la recherche locale améliore les solutions générées.

Dans ce travail de recherche, nous allons étudier le problème de partitionnement de graphe dans le cadre général et non pas comme application spécifique à un problème particulier. Notre étude consiste à développer des algorithmes standards applicables à tous les types de graphes. Contrairement à certains algorithmes spécialisés dans le partitionnement d'un type particulier de graphes (par exemple, les graphes issus de la modélisation par éléments finis où un sommet ne peut être relié qu'aux sommets qui lui sont proche dans l'espace), nos algorithmes devraient être capables de produire de bonnes partitions sur n'importe quel graphe. Pour valider les performances de nos algorithmes, nous utiliserons des graphes bien connus de la littérature (Soper et al., 2004). Ces graphes issus de différents domaines (circuit électronique, maille à l'élément fini 3D, réseau de route, etc.) sont utilisés par la plupart des concepteurs pour tester et évaluer les performances de leurs algorithmes. Ils représentent ainsi un bon repère de comparaison.

La principale partie de notre mémoire est structurée en trois chapitres. Dans le premier chapitre, nous commençons par expliquer le problème de partitionnement de graphe et présenter une revue bibliographique des heuristiques qui cherchent à l'optimiser. Dans le second, nous présentons nos propres algorithmes heuristiques et nous expliquons leur fonctionnement. Ensuite, nous allons tester et comparer les performances de nos algorithmes par rapport aux algorithmes les plus performants de la littérature. À la fin de la mémoire, nous concluons notre étude et nous discutons des améliorations qui peuvent être apportées à nos algorithmes.

Chapitre 2. REVUE BIBLIOGRAPHIQUE

Ce chapitre comporte quatre parties. La première vise à expliquer le problème étudié et donner des notions qui serviront à comprendre la suite de notre mémoire. La seconde exhibe quelques applications pratiques du problème de partitionnement de graphes, ceci afin de montrer l'intérêt de ce problème dans la vie réelle. La troisième partie explique le fonctionnement général des métaheuristiques que nous allons utiliser dans le chapitre suivant. Enfin, la quatrième partie présente une large revue des algorithmes développés pour optimiser notre problème.

2.1 Notations et définition du problème

Un graphe G est un couple (V, E) où V est un ensemble fini d'objets (sommets) et E est un sous-ensemble de VxV. Les éléments de E (les arêtes) sont des paires (x, y) de sommets qui limitent les extrémités de l'arête. La figure 1(a) ci-dessous représente un graphe à 10 sommets et 16 arêtes. Les sommets et les arêtes peuvent avoir des poids différents. Nous noterons par $|v|$ le poids du sommet v et par $|e|$ le poids de l'arête e. Cependant, dans le domaine de partitionnement de graphe, les sommets et les arêtes ont souvent des poids unitaires ($|v|=1$ pour tout V et $|e|=1$ pour tout E). Ceci est ainsi vrai pour tous les graphes que nous allons expérimenter avec nos algorithmes.

Une partition P du graphe G est une subdivision de V en k sous-ensembles (classe) disjoints P_i telle que l'union des P_i donne V. Le poids d'un sous-ensemble P_i est égal à la somme des poids des sommets appartenant à cet sous-ensemble et l'ensemble d'arêtes interclasses ou arêtes coupées (c.-à-d. les sommets incidents à l'arête appartiennent à des classes différentes) est noté par E_c. L'objectif habituel du problème de partitionnement de graphes est de trouver une partition qui équilibre le poids des sommets dans chaque sous-ensemble tout en minimisant le poids des arêtes coupées $|E_c|$. Pour qu'une partition soit équilibrée, il faut que le poids de chaque sous-ensemble soit inferieur ou égal à un poids optimal égal à $\lceil |V|/k \rceil$ (La fonction $\lceil x \rceil$ retourne le plus

petit entier $\geq x$). La figure 1 illustre un exemple de graphe à 10 sommets et une partition en 4 classes de ce dernier. Cette partition comporte 9 arêtes coupées (arêtes en pointillés) et satisfait la contrainte $|P_i| \leq \lceil 10/4 \rceil = 3$.

Il est commun d'accorder un certain déséquilibre à la taille des classes. Certaines expériences ont montré que le fait d'accorder un déséquilibre peut dans certains cas améliorer la qualité des partitions (Simon et al., 1997). Le déséquilibre est défini comme un écart par rapport au poids optimal des classes.

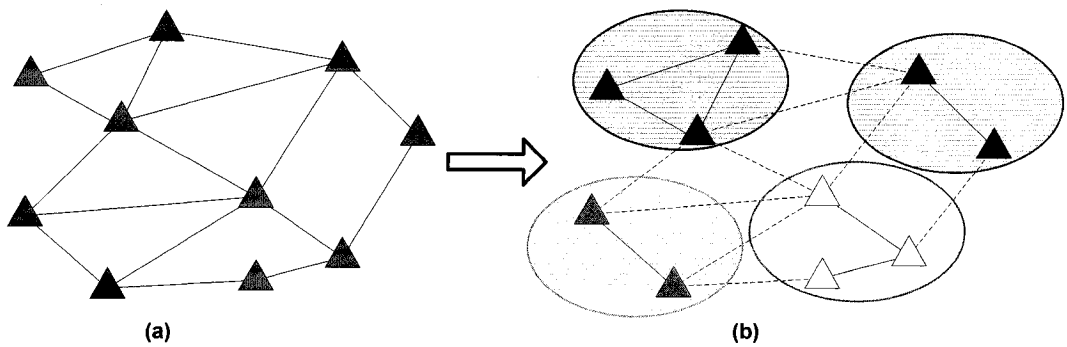

Figure 1: Représentation d'un graphe et son partition en 4 classes.

Étant donné un graphe $G= (V, E)$, un nombre de classes k et déséquilibre x, nous allons considérer, dans notre mémoire, le problème de partitionnement de graphes comme le problème qui consiste à partitionner l'ensemble des sommets V sur k classes de sorte que le nombre d'arêtes interclasses soit minimisé et que la taille de chaque classe soit strictement inferieure à $\lceil |V|/k \rceil (100+x)/100$.

2.2 Applications du problème de partitionnement de graphes

Le problème de partitionnement de graphe comporte de nombreuses applications dans différents domaines. Celles-ci incluent les domaines du calcul parallèle et distribué (Hendrickson et al, 1993), du design VLSI (Karypis et al., 1997), du stockage des données (Karypis et al., 1997), etc. Nous allons, dans le reste de cette section, présenter quelques-unes de ces applications et montrer leurs relations avec notre problème.

2.2.1 Calcul Distribué

Le calcul distribué consiste à répartir un calcul sur plusieurs ordinateurs distincts. Le but d'une telle répartition est la mise en commun de la puissance de calcul de plusieurs processeurs. Les communications entre les processeurs dépendent beaucoup de la tâche à effectuer. En effet, lorsque le traitement d'une donnée est complètement indépendant des autres, aucune communication inter-processeurs n'est alors nécessaire. Malheureusement, dans la plupart des cas, le calcul effectué sur un processeur ne peut être mené à terme que s'il utilise des résultats trouvés par d'autres processeurs. Dans ce cas, il est nécessaire d'établir des communications entre les processeurs concernés et de diffuser les résultats convoités.

Les communications jouent un rôle très important et influencent énormément les performances du calcul (Karypis et al.,1997). En effet, elles engendrent toujours des surcoûts (en termes de temps d'exécution) lorsque l'application envoie et reçoit des messages, mais surtout lorsqu'elle attend des messages en provenance d'autres processeurs. De ce fait, l'efficacité du calcul distribué nécessite une partition équitable de la charge de calcul entre les processeurs de sorte que le coût des communications soit minimisé. Autrement, aucun avantage ne peut être déduit de la mise en parallèle du calcul, et on peut ainsi se contenter d'un calcul centralisé sur une seule machine.

Pour différentes applications scientifiques, le problème de la partition de la charge de calculs sur plusieurs processeurs peut être facilement décrit en un langage de graphes (Hendrickson et al, 1993). En effet, un sommet dans le graphe peut représenter une tâche de calcul, alors qu'une arrête entre deux sommets peut indiquer la dépendance entre les données. Ainsi, pour que le calcul puisse être réalisé efficacement sur des machines parallèles, le graphe doit être reparti en domaines (portions), ayant un nombre de sommets approximativement égaux, de telles sortes que les arêtes inter-domaines soient au nombre minimum.

2.2.2 Conception VLSI

VLSI (Very-Large-Scale Integration) est une technologie de circuit intégré dont la densité d'intégration permet de supporter plus de 100,000 composants électroniques sur une même puce. Durant le processus de conception et de synthèse d'un circuit VLSI, il est important de pouvoir diviser les spécifications du système en un ensemble de classes de telle sorte que les connections interclasses soient minimisé (Karypis et al.,1997). En effet, cette étape comporte de nombreuses applications comme celles du prototypage rapide, de la synthèse HDL, de la simulation et des tests. Par ailleurs, de nombreux systèmes de prototypage rapide utilisent le partitionnement afin d'implanter des circuits complexes sur des centaines de FPGAs (puce) interconnectées (une FPGA représente une classe). Ce type d'application représente un grand défi vu les contraintes reliées à la topologie des composants électroniques. Par exemple, dans le cas où le nombre de signaux sortant d'une classe serait supérieur au nombre d'épingles disponible sur une FPGA, alors cette classe ne peut être implantée en utilisant une seule FPGA. En conséquence, le circuit doit être subdivisé d'avantage et ainsi implanté en utilisant plus d'FPGA.

Un graphe peut être utilisé pour représenter naturellement un circuit VLSI. Les sommets du graphe peuvent ainsi être utilisés pour représenter les différentes portes logiques du circuit, et les arêtes peuvent êtres utilisées pour représenter les connections entre ses portes (comme illustré à la figure 2, Banos et al 1999). Il faut noter que, dans une telle application, l'aptitude de l'algorithme de partitionnement à produire des partitions de bonne qualité affecte énormément la faisabilité, la qualité et le coût du système résultant (le circuit).

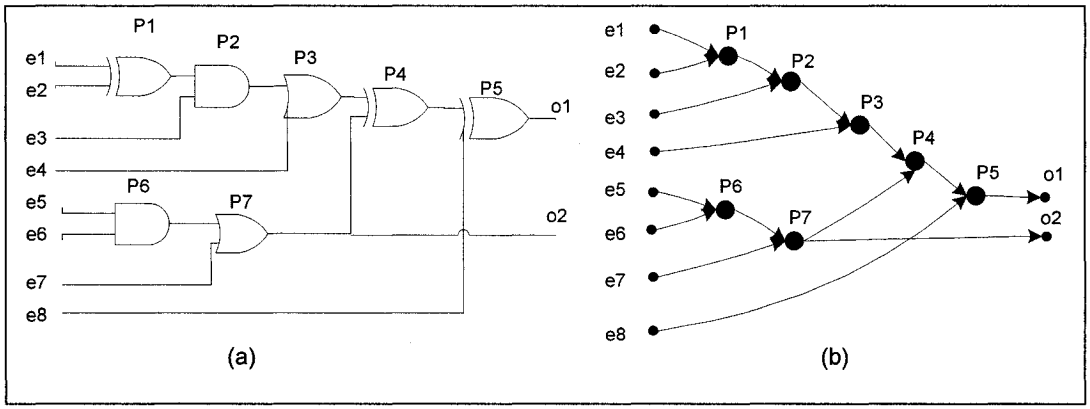

Figure 2 : Représentation d'un circuit combinatoire á 7 portes logique (a) et le graphe qui lui est associé (b).

2.2.3 Regroupement de données

L'efficacité du stockage dans les grandes bases de données exige que les informations consultées simultanément par des requêtes individuelles soient stockées sur un nombre restreint de blocs disque. Ceci améliore significativement la performance des opérations surtout quand le temps d'accès au disque représente un goulot d'étranglement.

Le partitionnement de graphe est une méthode efficace pour le regroupement des données (Karypis et al., 1997). Dans cette application, on représente les données stockées dans la base de données par des sommets. Pour accélérer l'accès à un ensemble de données, on ajoute des arêtes entre les sommets qui sont accédés par une même requête. Une base de données est ainsi modélisée par un graphe. Le partitionnement d'un tel graphe permet de regrouper les données en blocs de taille aussi petites, que la taille des secteurs mémoires dans le disque, et de minimiser les arêtes inter-blocs.

2.3 Les métaheuristiques

Les métaheuristiques forment une famille d'algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficile pour lesquels on ne connaît pas de méthodes classiques efficaces. Ils constituent des méthodes approchées qui ne garantissent pas la complétude de la résolution. Ces algorithmes sont des méthodes

génériques pouvant optimiser une large gamme de problèmes différents, sans nécessiter de changements profonds dans l'algorithme employé.

Dans cette section, nous allons présenter les algorithmes génériques que nous allons utiliser lors de la conception des algorithmes dédiés au problème de partitionnement de graphes. Ces algorithmes sont les suivants; l'algorithme *glouton*, l'algorithme *tabou* et l'algorithme *mémétique*.

2.3.1 Algorithme glouton

On appelle algorithme glouton un algorithme qui suit le principe de faire, étape par étape, un choix optimal local, dans l'espoir d'obtenir un résultat optimal global. Le schéma général de cet algorithme est basé sur un critère local de sélection des éléments pour construire une solution optimale. Ainsi, en partant d'une solution initiale, généralement vide, l'algorithme sélectionne à chaque étape tous les candidats possibles et évalue leurs scores. Le candidat ayant le meilleur score est incorporé dans la solution partielle. L'algorithme glouton se termine lorsque la solution est rendue complète (mais non nécessairement légale). L'algorithme 1, ci-dessous, récapitule les différentes étapes d'une procédure gloutonne.

```
1   Initialiser la solution partielle S (S := solution vide).
2   Tant que le critère d'arrêt n'est pas satisfait, faire :
    a. Déterminer l'ensemble de candidats et calculer le score de
       chacun d'entre eux.
    b. Choisir le candidat qui maximise (minimise) le score.
    c. Compléter la solution partielle S à l'aide du candidat
       choisi.
3   Renvoyer S.
```

Algorithme 1 : Schéma général d'une procédure gloutonne.

2.3.2 Algorithme tabou

La recherche taboue est une métaheuristique d'optimisation présentée par Fred Glover en 1986. Cette méthode est une métaheuristique itérative qualifiée de recherche locale au sens large.

Admettons que S soit un ensemble de solutions à un problème d'optimisation et que f soit une fonction qui mesure la valeur $f(s)$ de toute solution s dans S. On défini un voisinage est comme étant une fonction N qui associe un sous-ensemble de S à toute solution $s \in S$. Une solution $s' \in N(s)$ est dite voisine de s. Le principe de la recherche taboue consiste à choisir, à chaque itération, la meilleure solution $s' \in N(s)$, même si $f(s')>f(s)$ (Hertz, 2006). En effet, lorsque la recherche atteint un minimum local, par rapport au voisinage $N(s)$, elle est obligée de choisir une solution s' qui est plus mauvaise que s. Cependant, le danger est qu'à l'itération suivante, elle revienne immédiatement à s puisque s est meilleure que s'. Pour éviter de tourner en rond, on crée une liste T, nommée *Liste Tabou*, qui mémorise les dernières solutions visitées et interdit tout déplacement vers une solution de cette liste. Les solutions ne demeurent dans T que pour un nombre limité d'itérations. La liste T est donc une mémoire à court terme. Les solutions appartenant à cette liste sont nommées *solutions taboues*. De même, tout mouvement qui nous mène de la solution courante à une solution de T est appelé *mouvement tabou*. En définissant l'ensemble $N^T(s)$ comme l'ensemble de solutions voisines de s non-taboues, la recherche locale choisie, à chaque itération, la meilleure solution $s' \in N^T(s)$. L'algorithme 2, ci-dessous, résume les différentes étapes d'un algorithme Tabou.

1. Choisir une solution $s \in S$, poser $T:=\phi$ et $s^*:=s$.
2. Tant qu'aucun critère d'arrêt n'est satisfait, faire :
   a. Déterminer une solution s' qui minimise $f(s')$ dans $N^T(s)$.
   b. Si f(s') < f(s*) alors poser s*:=s'.
   c. Poser s:=s' et mettre à jour T.
3. Fin du tant que

Algorithme 2 : Schéma général d'un algorithme Tabou.

Comme critère d'arrêt on peut par exemple fixer un nombre maximum d'itérations sans amélioration de s^* ou on peut fixer un temps limite après lequel la recherche doit s'arrêter.

2.3.3 Algorithme mémétique

L'algorithme mémétique est une approche hybride qui combine plusieurs méthodes de recherches : une méthode basée sur le voisinage et une autre basée sur la population. Le principe de base consiste à générer une solution à partir de celles de la population courante et d'appliquer la recherche locale à cette solution. La solution retournée par recherche locale est ensuite utilisée pour mettre à jour la population. Ce processus se termine lorsqu'un critère d'arrêt est satisfait.

Un algorithme mémétique est inspiré du processus d'évolution naturelle. En partant d'un ensemble de solutions, qui peuvent être vues comme une population d'individus, il sélectionne à chaque itération deux d'entre elle et applique un opérateur de croisement pour en générer un nouveau. Ce nouvel individu hérite des caractéristiques des anciens. Il est appelé l'enfant des individus sélectionnés. On applique ensuite un algorithme de recherche locale sur l'enfant généré. Enfin, on utilise l'individu enfant pour remplacer un individu de la population. L'algorithme 3 décrit les différentes étapes de l'algorithme mémétique (Galinier et al., 1999).

```
1. Générer une population P d'individus
2. Tant qu'aucun critère d'arrêt n'est satisfait, faire :
      a. Sélectionner 2 individus I1 et I2 de la population P.
      b. Créer un enfant E en croisant les individus I1 et I2.
      c. Appliquer la recherche locale sur E pendant L itérations.
      d. Remplacer un individu de la population par E.
```

Algorithme 3 : Schéma général d'un algorithme mémétique.

Comme critère d'arrêt, on peut considérer le taux de similitude des individus de la population. En effet, après un certain nombre d'itérations, les individus deviennent de

plus en plus semblables. La combinaison aurait ainsi tendance à produire les mêmes enfants et la recherche locale d'effectuer les mêmes mouvements. Dans ce cas, il vaut mieux arrêter l'algorithme.

2.4 Algorithmes de partitionnement de graphes

Le problème de partitionnement de graphe est un problème NP-Difficile (Garey et al., 1976). Ces dernières années, beaucoup d'efforts ont été consacrés au développement des heuristiques pour essayer de lui trouver une approximation raisonnable. Ces heuristiques peuvent souvent manipuler des graphes dont les tailles dépassent un million de sommets et leur trouver de bonnes partitions. Contrairement au développement d'heuristiques, peu d'efforts ont été déployés pour le développement d'algorithmes exacts. En effet, à cause de la NP-difficulté du problème, seuls les graphes de taille relativement petite peuvent être résolus de façon exacte.

Dans cette section, nous allons présenter des algorithmes heuristiques, de différentes conceptions, parmi les plus connus et les plus performants. Ces heuristiques de partitionnement de graphes sont réparties en deux catégories selon les informations disponibles sur les graphes (Dehmel, 1996). Une première catégorie utilise des graphes ayant des informations sur les positions de sommets dans l'espace. Les sommets ont ainsi une position géométrique déterminée par des coordonnées (exemple, dans le cas d'un graphe planaire les sommets sont identifiés par deux coordonnées). Ce type de graphe est généralement issu de la discrétisation des objets physiques (des éléments finis) et possède des propriétés particulières telles qu'un sommet ne peut être relié qu'aux sommets qui lui sont proches dans l'espace. Cette catégorie d'heuristiques est connue sous le nom d'heuristiques *géométriques*. Il existe un second type de graphe sur lesquels on ne dispose d'aucune information concernant les positions spatiales des sommets. De plus, les arêtes de ces graphes ne possèdent pas d'interprétation simple comme celle de la représentation de proximité physique des graphes géométriques. Les seules informations qui sont utilisées dans ce cas sont les arêtes. De ce fait, le partitionnement de cette catégorie de graphes nécessite l'utilisation d'algorithmes

combinatoires (ce type d'algorithme peut aussi servir à partitionner les graphes géométriques).

Il existe une multitude d'algorithmes pour résoudre le problème du partitionnement de graphe. Parmi les plus anciens nous pouvons citer l'algorithme de recherche en largeur qui permet de trouver des bonnes solutions dans certain cas de mailles à élément fini. Cependant, l'algorithme le plus efficace, dont plusieurs variantes sont encore utilisées de nos jours, est l'algorithme de *Kernighan-Lin*. En partant d'une partition initiale, ce dernier effectue itérativement une série d'échanges entre deux groupes de sommets, et choisit, à chaque étape, la meilleure série qui permet de minimiser le nombre d'arêtes coupées. Plus récemment, le problème de partitionnement de graphe est réduit à un problème de partitionnement de vecteurs en utilisant les vecteurs propres de la matrice Laplacienne du graphe. Ce type d'algorithme est connu sous le nom d'algorithme de partitionnement *spectral*. La plupart de ces algorithmes peuvent être accélérés en utilisant le concept *multi-niveaux*. L'idée de ce concept consiste à remplacer le graphe de base par un graphe compacté qui comporte un nombre réduit de sommets, de le partitionner et par la suite de propager la solution trouvée sur le graphe de base.

Quel algorithme est-il meilleur? Dans ce qui suit, nous allons décrire plus en détail le fonctionnement de certains algorithmes les plus connus et nous allons discuter des performances de chacun d'entre eux. La majorité de ces algorithmes sont conçus pour résoudre le problème de bissection (ou bi-partitionnement) de graphe. Le *k*-partitionnement peut facilement être obtenu en appliquant récursivement un même algorithme sur chacune des classes jusqu'à atteindre le nombre de classes voulues. Pour des fins pratiques, nous allons regrouper les algorithmes en deux catégories selon les types de graphes utilisés (avec et sans informations sur les coordonnées). Cependant, nous allons passer rapidement sur les algorithmes géométriques et nous allons nous attarder sur les algorithmes combinatoires puisque, dans notre étude, nous allons développer des algorithmes de cette dernière catégorie.

2.4.1 Partitionnement des graphes avec informations sur les coordonnées

Il existe des graphes pour lesquels on dispose d'informations sur les coordonnées des sommets. Ces coordonnées sont souvent disponibles lorsque les graphes proviennent de la discrétisation d'objets physiques (Dehmel, 1996).

Dans cette section, nous allons présenter deux algorithmes de bissection récursive pour les graphes avec des coordonnées spatiales des sommets. Ces algorithmes sont ; l'algorithme récursif de bissection de coordonnées (Simon, 1994) et l'algorithme récursif de bissection inertielle (Hendrickson et al., 1995).

2.4.1.1 Algorithme récursif de bissection de coordonnées (RBC)

Cet algorithme est l'un des premiers algorithmes conçus pour la résolution des graphes avec informations sur les coordonnées spatiales (Simon, 1994). Il suppose que pour repérer un ensemble de sommet $V = \{v_1, v_2, ...v_n\}$ dans l'espace, on ne peut avoir besoin que de deux ou trois coordonnées. Cette supposition est vérifiée dans le cas des graphes à éléments finis. Ainsi, à chaque sommet $v_i \in V$ est associé un couple $v_i = (x_i, y_j)$ ou un triplet $v_i = (x_i, y_j, z_i)$, selon du nombre de dimensions du graphe.

La stratégie de partitionnement de cet algorithme, consiste simplement à déterminer la direction (la coordonnée) de la plus grande expansion du graphe (la direction des sommets les plus éloignés dans le graphe) et de subdiviser les sommets selon cette direction. Sans perte de généralité, supposons que la coordonnée x soit la direction de la plus grande expansion du graphe. Les sommets sont alors triés selon leur coordonnée en x. Ceux ayant les plus petites valeurs de la coordonnée en x formeront le premier sous-ensemble, alors que les sommets restants formeront le second sous-ensemble. Cette stratégie peut être appliquée récursivement sur chacun des sous-ensembles jusqu'à obtenir le nombre de classes voulu. L'algorithme 4, ci-dessous, récapitule les différentes étapes de la procédure récursive de bissection de coordonnées (*RBC*).

1. Déterminer la direction de la plus grande expansion du graphe (x, y ou z).

2. Trier les sommets selon la coordonnée associée à la direction choisie.

3. Former un ensemble avec les sommets qui ont les plus petites coordonnées et un autre avec les sommets qui ont les plus grandes coordonnées.

4. Répéter récursivement jusqu'à atteindre le nombre de classes voulu.

Algorithme 4 : Procédure récursive de bissection de coordonnées.

L'algorithme *RBC* présente de nombreux inconvénients. En effet, étant donné que les espacements entre les nœuds ne sont pas nécessairement constants et que l'algorithme ne tient pas compte des arêtes, alors il ne faut pas s'attendre à ce que les sommets appartenant à une même classe soient connexes. De plus, l'algorithme est très sensible à la direction de l'expansion du graphe. Une simple rotation du repère de coordonnés induit une solution complètement différente.

2.4.1.2 Algorithme récursif de bissection inertielle (RBI)

La bissection inertielle (ou bissection orthogonale) est une variante de la bissection de coordonnées (Hendrickson et al., 1995). Elle tire avantage du fait que l'orientation naturelle du graphe peut ne pas correspondre à l'un des axes des coordonnées. Ainsi, l'amélioration apportée consiste à choisir la direction d'expansion comme étant la perpendiculaire à l'axe d'inertie *I* du graphe, ceci au lieu de choisir une perpendiculaire à l'un des axes de références (x, y ou z) dans le cas de la bissection de coordonnée.

Par définition, l'axe d'inertie *I* représente l'axe autour duquel les sommets du graphe ont un moment angulaire de valeur minimale. L'astuce derrière cette approche consiste au fait que les sommets du graphe vont tous être à proximité de l'axe *I*. Ainsi en traçant un axe perpendiculaire à celui-ci, nous allons nécessairement découper un

nombre limité d'arêtes. La figure 3 illustre l'emplacement de l'axe d'inertie *I* ainsi que la droite perpendiculaire qui permet de couper le graphe en deux parties de taille égale.

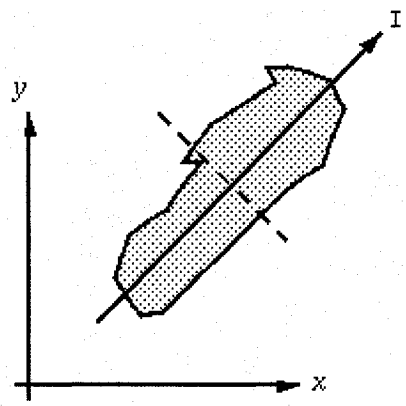

Figure 3 : Illustration de la bissection Inertielle.

L'algorithme 5 récapitule les différentes étapes de la procédure récursive de bissection inertielle (*RBI*).

1. *Déterminer* l'axe d'inertie *I* du graphe.

2. *Choisir* un axe (ou un plan) perpendiculaire à *I* de sorte qu'il découpe le graphe en deux parties de taille égale.

3. *Répéter* récursivement jusqu'à atteindre le nombre de classes voulu.

Algorithme 5 : Procédure récursive de bissection inertielle.

La faiblesse de l'algorithme *RBI*, comme celle de l'algorithme *BCR*, est qu'il ne tient pas compte des arêtes. En effet, l'algorithme ne cherche pas à minimiser les arêtes coupées. Il se contente de produire une partition initiale.

2.4.2 Partitionnement des graphes sans informations sur les coordonnées

Ces dernières années, de nombreux projets et laboratoires de recherche se sont spécialisés dans le développement d'algorithmes pour la résolution du problème de partitionnement de graphes. Certains d'entre eux développent des algorithmes pour des

applications spécifiques, tels que les algorithmes de partitionnement de circuits VLSI. D'autres développent des algorithmes pour des applications plus larges mais qui restent applicable sur des graphes particuliers (exemple, graphes issus des éléments finis où les sommets sont liés de proche en proche). Ce qui nous intéresse dans notre recherche, ce sont les algorithmes qui servent à la résolution de problèmes de différents domaines, c'est-à-dire des algorithmes qui ne se basent sur aucune spécification reliée à certaines applications, et qui sont capables de bien partitionner n'importe quel type de graphe. De ce fait, nous allons nous concentrer sur les concepts de certains algorithmes qui restent dans un cadre de résolution général.

Dans les sections suivantes de ce chapitre, nous allons analyser les concepts de quelques algorithmes parmi les plus performants, afin de bénéficier de certaines de leurs astuces dans le développement de nos propres algorithmes. Dans une seconde étape, nous faisons aussi comparer les performances de nos algorithmes par rapport aux leurs. Nous allons d'abord présenter quelques algorithmes parmi les plus anciens qui ont révolutionné le domaine de partitionnement de graphe. Ces algorithmes sont : l'algorithme récursif de bissection de graphe (Simon, 1994), l'algorithme récursif de bissection spectrale (Pothen et al., 1990), l'algorithme de Kernighan-Lin (Kernighan & Lin, 1970) et l'algorithme de bissection multi-niveaux (Hendrickson et al., 1993). Nous allons ensuite passer à la description de quelques nouveaux algorithmes. Parmi ces derniers, nous allons nous intéresser à p-METIS (Karypis et al., 1999), à k-METIS (Karypis et al., 1998), à JOSTLE (Walshaw et al., 2000), à JOSTLE Evolutionary (Soper et al., 2004), à Iterated JOSTLE (Walshaw et al., 2004) et à PMSATS de Banos et al (2004).

2.4.2.1 Algorithme récursif de bissection de graphe (RBG)

L'algorithme de bissection de graphe (Simon, 1994) se base sur une approche qui tire profit de la notion de distance entre les sommets. La distance $d(v_i, v_j)$ est définie comme étant le plus court chemin entre les sommets v_i et v_j. L'algorithme détermine d'abord deux sommets extrêmes ayant une distance maximale (c'est-à-dire deux sommets ayant le plus long des plus courts chemins) et assigne chacun des ces sommets

à un ensemble. Ensuite, il trie les restes des sommets dans l'ordre croissant de leurs distances par rapport à l'une ou l'autre des deux extrémités. Enfin, il assigne les sommets à l'un des deux ensembles selon leurs proximités des sommets extrêmes. La principale difficulté de cet algorithme réside dans le calcul du diamètre du graphe (le plus long des plus courts chemins). Il existe de nombreuses heuristiques qui peuvent être utilisées à cette fin. Par exemple, nous pouvons choisir arbitrairement un sommet *R* comme une racine potentielle du graphe. Par la suite, nous accordons aux voisins de ce sommet le niveau 1, les voisins de ses voisins le niveau 2 et ainsi de suite jusqu'à atteindre tous les sommets. Le dernier sommet marqué aura alors le niveau *m* qui détermine la distance qui le sépare de la racine. Ce dernier sommet est considéré par la suite comme nouvelle racine *R* et les mêmes étapes seront alors exécutées de nouveaux. Quant la valeur de *m* reste inchangée, *R* est alors pris comme racine et *m* comme diamètre du graphe. Dans la pratique, cette procédure converge rapidement au bout de quelques itérations. La moitié des sommets qui sont les plus proches de la racine formeront ainsi une première classe, et le reste formera la seconde. L'algorithme 6 récapitule les différentes étapes de la procédure récursive de bissection de graphe (*RBG*).

1. *Évaluer* le diamètre du graphe.
2. *Trier* les sommets selon leurs niveaux.
3. *Assigner* la moitié des sommets à chacun des sous-ensembles.
4. *Répéter* récursivement jusqu'à atteindre le nombre voulu de classes dans la partition.

Algorithme 6 : Procédure récursive pour la bissection de graphe.

L'algorithme *RBG* suppose que le graphe est connexe. Dans le cas contraire, tous les sommets ne peuvent pas être assignés à un domaine. Selon Simon (1994), l'avantage de cet algorithme est que les classes résultantes sont compactes (la distance entre les sommets les plus éloignés de la classe est réduite) et renferment un ensemble connexe de sommets (un sous-graphe induit connexe).

2.4.2.2 Algorithme récursif de bissection spectrale (RBS)

L'algorithme que nous allons présenter ici est totalement différent de ceux présentés auparavant et surtout beaucoup moins intuitif. L'algorithme récursif de bissection spectrale dérive de la stratégie de la bissection de graphe, développée par A. Pothen et al. (1990) qui se base sur le calcul spécifique d'un vecteur propre de la matrice de Laplace du graphe. La matrice de Laplace $L(G) = (l_{ij}), i, j = 1....n$ du graphe G est définie comme suit (avec n le nombre de sommets dans le graphe):

$$l_{ij} = \begin{cases} +1 & si\ (v_i, v_j) \in E \\ -\deg(v_i) & si\ i = j \\ 0 & sinon \end{cases}$$

Où deg (v_i) est le degré du sommet v_i. Par définition, le degré d'un sommet v_i est le nombre de sommets v_j tels que $(v_i, v_j) \in E$. La matrice L est symétrique, semi-définie négative et sa plus petite valeur propre est égale à zéro. À la seconde plus petite valeur propre correspond un vecteur propre appelé le vecteur de *Fiedler* (Fiedler, 1975). Ce vecteur est tel que les signes de ses composantes divise les sommets du graphe en deux parts plus ou moins égales. L'algorithme RBS consiste ainsi à partager les composantes du vecteur de Fiedler en assignant les sommets d'un certains signe à un premier sous-ensemble et les sommets de l'autre signe au second sous-ensemble. Les deux sous-ensembles résultants sont partitionnés à leur tour en deux sous-ensembles, jusqu'à ce qu'on obtienne le nombre de sous-ensembles voulu. L'algorithme 7 récapitule les différentes étapes de la procédure récursive de bissection spectrale (*RBS*).

1. *Calculer* le vecteur de *Fiedler*.
2. *Trier* les sommets selon la valeur des éléments du vecteur de *Fiedler*.
3. *Assigner* la moitié des sommets à chacun des sous-ensembles.
4. *Répéter* récursivement jusqu'à atteindre le nombre de partition voulu.

Algorithme 7 : Procédure récursive pour la bissection spectrale.

D'après différentes sources (Simon, 1994), il s'avère que cette technique génère un partitionnement de meilleure qualité comprenant moins d'arêtes coupées que les deux premières techniques de bissection récursive. De plus, les classes obtenues sont toutes connexes (referment des sous-graphes induits connexes), parfaitement arrondies et compactes (la distance entre les sommets les plus éloignés de la classe est réduite).

2.4.2.3 Algorithme de Kernighan-Lin (KL)

L'algorithme de *Kernighan-Lin* (Kernighan & Lin, 1970) est un algorithme de descente de nature itérative. Il est principalement conçu pour le problème de la bissection de graphe. Ensuite, il a été généralisé au *k*-partitionnement de graphe. L'algorithme commence avec une partition initiale d'un graphe donné. À chaque itération, il cherche deux sous-ensembles de sommets, chacun appartenant à une classe différente, de telle sorte qu'en les échangeant nous obtenons une partition avec moins d'arêtes coupées. Pour évaluer une série d'échanges entre deux classes, *KL* trie les sommets de chaque classe selon leurs gains. Le gain d'un sommet est égal à la différence entre le nombre d'arêtes coupées de la solution courante et le nombre d'arêtes coupées si le sommet est déplace vers l'autre classe. L'algorithme évalue ensuite l'échange des deux sommets de meilleurs gains et met à jour les gains des autres sommets de sorte que leur nouveau gain prenne en compte cet échange. Il faut noter que l'échange n'a pas eu encore lieu mais les gains des sommets sont évalués comme s'il l'est réellement. Ce processus est répété jusqu'à ce que tous les sommets des deux classes aient été traités. À ce moment, l'algorithme cherche le meilleur gain enregistré et effectue ensuite la série d'échange qui a abouti à ce gain. L'algorithme se termine quand aucune série d'échange ne permet d'améliorer la partition. Le temps d'exécution alloué à chaque itération est proportionnel à $O(|E| \log |E|)$.

Il existe plusieurs améliorations de l'algorithme original de *KL*. Parmi celles-ci, nous pouvons citer l'algorithme de Fiduccia et Mattheyeses (FM) (1982) qui permet de réduire la complexité d'une itération à $O(|E|)$. La différence majeure entre ces deux algorithmes est qu'à la place d'échanger deux sommets, *FM* ne déplace qu'un sommet à

la fois. Ainsi, au lieu de chercher une série d'échanges à chaque itération, *FM* essaye de trouver une série de mouvements uniques, qui permet d'améliorer la solution.

L'algorithme de KL permet de trouver un optimum local quand la partition initiale est de bonne qualité et quand la moyenne des degrés des sommets du graphe est assez grande (Bui et al., 1993).

2.4.2.4 Algorithme de bissection multi-niveaux

On peut partitionner un graphe en utilisant l'algorithme multi-niveaux (Hendrickson et al., 1993). Le principe de base de cet algorithme est simple. Il consiste à compacter (compresser) les sommets d'un graphe jusqu'à atteindre un nombre limité de sommets (moins que cent sommets), partitionner le graphe réduit et ensuite projeter la partition trouvée sur le graphe original. Cette méthode permet ainsi de réduire la taille du graphe dans le but de limiter la complexité. La compression de graphe se fait en plusieurs étapes (ou niveaux). La première étape consiste à compresser le graphe original d'un certain coefficient et obtenir un nouveau graphe de taille plus réduite. À la l'étape suivante, la compression se fait sur le nouveau graphe (réduit) et permet ainsi d'obtenir un nouveau graphe encore plus réduit. La phase de compression se termine lorsque le graphe atteint une certaine taille. Le graphe le plus réduit est ensuite partitionné et la partition trouvée est projetée sur les graphes intermédiaires jusqu'à atteindre le graphe original. La figure 4 (Karypis et al., (1999)) illustre graphiquement le processus général de la bissection multi-niveaux. Pour améliorer la partition trouvée sur le graphe le plus réduit, un opérateur de raffinement (recherche locale) de la solution peut être appliqué à la suite de chaque projection.

Plus formellement, les étapes de la bissection multi-niveaux du graphe $G_0 = (V_0, E_0)$ peuvent être décrites comme suit :

1. *Une phase de compression* pendant laquelle le graphe G_0 est transformé en une séquence de graphes compactés $G_1, G_2, \ldots\ldots, G_m$ de sorte que $|V_0| > |V_1| > |V_2| > \ldots\ldots > |V_m|$.

2. *Une phase de bissection* qui permet d'obtenir une bissection P_m du graphe $G_m=(V_m, E_m)$.

3. *Une phase de décompression* pendant la quelle la partition P_m de G_m est projetée sur le graphe G_0 en passant par l'intermédiaire des partitions $P_{m-1}, P_{m-2}, \ldots \ldots, P_1, P_0$.

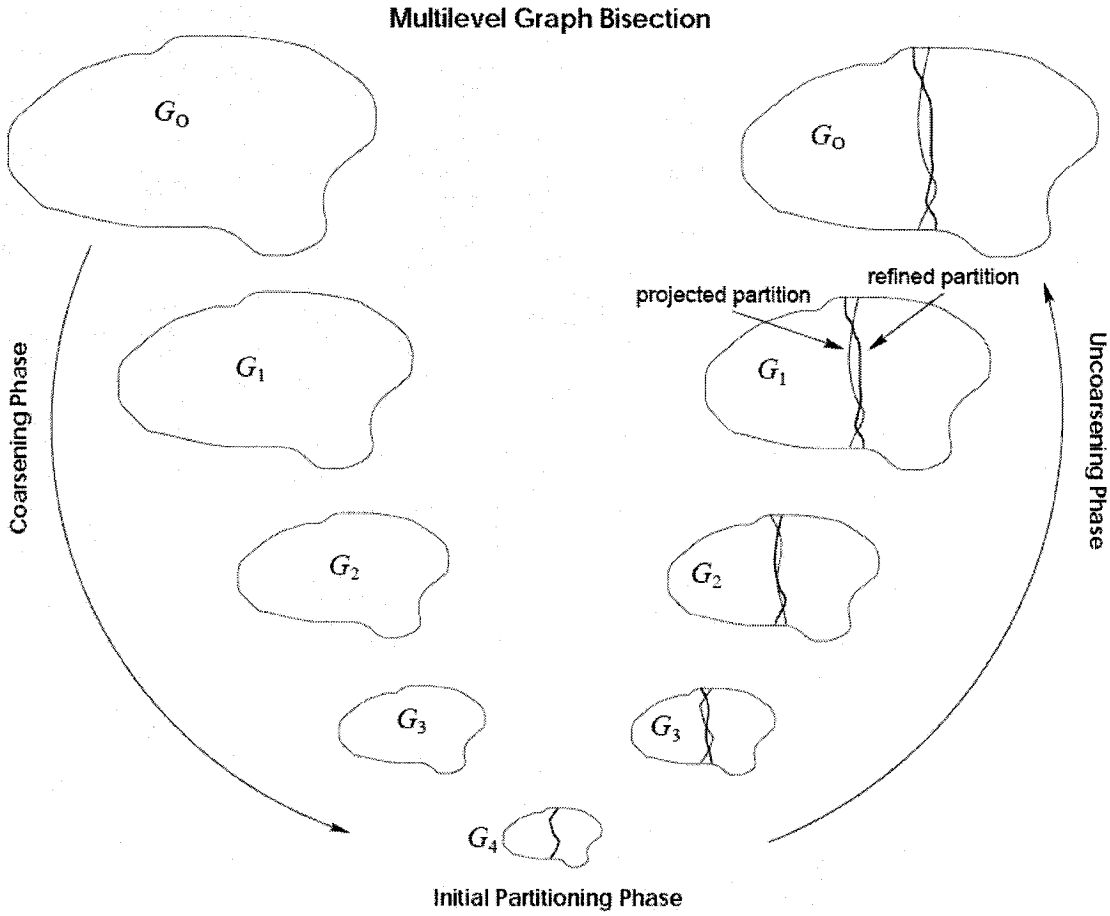

Figure 4 : Illustration des différentes phases de la bissection multi-niveaux du graphe.

L'algorithme 12 récapitule les différentes étapes de la procédure de partitionnement multi-niveaux.

```
1. Tant que le graphe n'a pas atteint la taille désirée, répéter :
        - Graphe := compacter (Graphe).
2. Partitionner le graphe.
3. Tant que le graphe n'a pas atteint la taille originale,
   répéter :
        a. Graphe := décompacter (Graphe).
        b. partition := décompacter (partition).
        c. Raffiner localement la solution.
```

Algorithme 8 : Procédure multi-niveaux.

2.4.2.4.1 La phase de compression

Durant la phase de compression, une séquence de graphes compactés est construite itérativement. Chacun de ces graphes comporte un nombre de sommets inférieur à celui du graphe qui le précède. Il existe différentes méthodes de compression de graphe. Quelques une d'entre-elles sont présentées dans la figure 5.

Dans la plupart des méthodes de compression, un ensemble de sommets V_i^v du graphe G_i est fusionné pour former un seul sommet v dans le graphe de niveau supérieur G_{i+1}. Le poids du sommet v est égal au poids total des sommets dans l'ensemble de V_i^v. Les arêtes incidentes à v sont les arêtes incidentes à l'ensemble V_i^v. Dans le cas particulier où plus d'une arête relie V_i^v à un même sommet u, alors le poids de l'arête (v, u) sera égal à la somme des poids des arêtes entre V_i^v et u. La figure 5 (Karypis & Kumar, (1999)) donne une illustration du calcul des poids des arêtes et des sommets pour différentes méthodes de compression. Il est ainsi facile de remarquer qu'à la suite de la projection d'une partition le poids des différentes classes ainsi que le poids des arêtes coupées sont maintenus constants. De ce fait, l'application de l'opérateur de raffinement sur un graphe intermédiaire permet de minimiser le nombre d'arêtes coupées du graphe original.

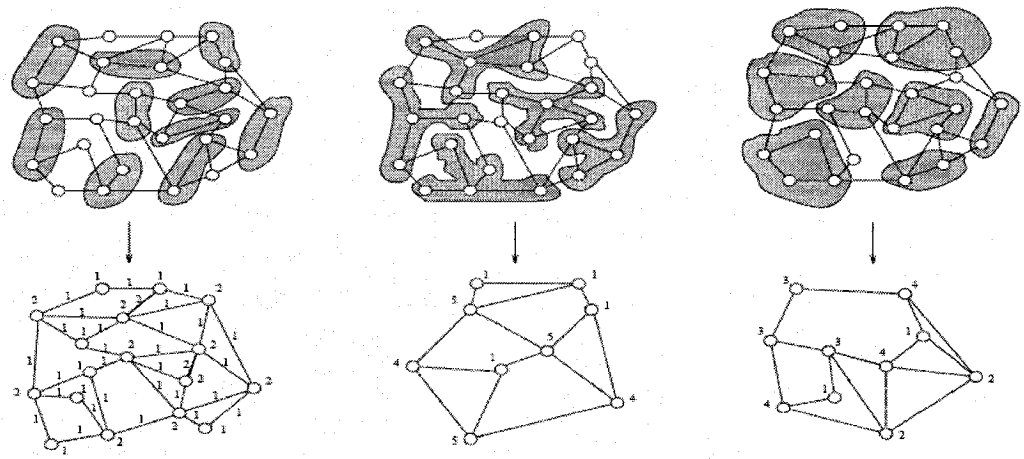

Figure 5 : Différentes méthodes de compression de graphe.

Il existe deux approches différentes permettant d'obtenir un graphe compacté. La première consiste à trouver un ensemble d'arêtes indépendantes (deux arêtes sont indépendantes si elles n'ont pas d'extrémité en commun) et de fusionner les sommets qui lui sont incidentes. Alors que la seconde cherche à former des multi-nœuds composés de groupes de sommets qui ont une grande connectivité (degré élevé). Cette dernière approche est plus avantageuse dans le cas des graphes à grande densité, alors que la première est utilisée surtout dans le cas des mailles à élément fini dont les sommets respectent un même patron de connectivité. L'algorithme 9 récapitule les différentes étapes de la phase de compression.

1. Trouver un couplage maximal dans le graphe (ensemble maximal d'arêtes indépendantes).
2. Pour chaque arête e (i, j) du couplage faire :
   a. Contracter l'arête e pour former un nouveau sommet v.
   b. Le poids de v: W(v) = W(i) + W(j).
   c. Si i et j sont adjacents au même sommet k alors :
      - Le poids de e: W_e (v, k) = W_e(i, k) + W_e(j, k).

Algorithme 9 : Procédure de compression de graphe.

2.4.2.4.2 La phase de partitionnement initial

La seconde phase de l'algorithme multi-niveaux consiste à déterminer une partition P_m le graphe compacté $G_m = (V_m, E_m)$. Cette partition doit équilibrer les tailles des classes tout en minimisant le nombre d'arêtes coupées. La partition du graphe G_m peut être obtenue en utilisant différents algorithmes de partitions tels que l'algorithme de partitionnement spectral, les algorithmes de partitionnement géométriques (dans le cas où les coordonnées sont disponibles) et les algorithmes combinatoires (ex : *Kernighan-Lin*). Étant donné que la taille du graphe G_m est assez réduite (généralement $|V_m|<100$), cette étape ne devrait pas prendre plus que quelques secondes.

2.4.2.4.3 La phase de décompression

La phase de la décompression du graphe est triviale. Elle consiste à projeter la partition P_m du graphe G_m vers le graphe original en passant par l'intermédiaire des graphes $G_{m-1}, G_{m-2}, \ldots, G_1$. Étant donné que chaque sommet (multi-nœud) du graphe G_{i+1} renferme un ensemble distinct de sommets du graphe G_i, le passage de P_{i+1} à P_i se fait simplement en remplissant chaque multi-nœud v par l'ensemble des sommets V_i^v qui lui est associé.

Le passage d'un niveau $i+1$ à un niveau inférieur i induit un dégré de liberté plus important dans la partition projetée. En effet, si la configuration P_{i+1} est un minimum local du graphe G_{i+1}, la projection P_i sur le graphe G_i n'est pas nécessairement elle aussi un minimum local. Par conséquent, il est encore possible d'améliorer la partition en utilisant un algorithme de raffinement de la solution. Plusieurs algorithmes basés sur la recherche locale peuvent ainsi être utilisés. Parmi ces algorithmes nous pouvons citer l'algorithme de *Kernighan-Lin*, l'algorithme tabou, etc.

2.4.2.5 P-METIS : Algorithme multi-niveaux à bissection récursive

L'algorithme P-METIS (Karypis et al., 1999) est basé principalement sur le concept multi-niveaux développé par Hendrickson et al., (1993). Les auteurs, Karypis et al., ont expérimenté l'effet des variations de différents paramètres de l'algorithme multi-

niveaux, et ils ont montré l'efficacité de certains de leurs choix sur la qualité de la partition obtenue. Ainsi, ils ont présenté un nouvel opérateur de compression qui permet de préserver les propriétés du graphe original de telle sorte que la partition établie sur le graphe compacté soit assez proche de la partition du graphe original. De plus, ils ont développé un opérateur de raffinement de la solution caractérisé par une grande rapidité de résolution. Dans ce qui suit, nous allons décrire les différents opérateurs qui sont à l'origine de l'efficacité de l'algorithme P-Metis.

2.4.2.5.1 Opérateur de compression : Couplage à poids maximum (CPM)

Comme nous l'avons expliqué antérieurement, la phase de compression consiste à réduire la taille du graphe dans le but de limiter la complexité du problème. L'une des méthodes la plus efficace pour compacter un graphe consiste à déterminer un couplage maximum (un couplage est un ensemble d'arrêtes non adjacentes) et à fusionner les sommets incidents aux arêtes de ce couplage. En effet, il a été démontré que la compression par couplage préserve plusieurs propriétés du graphe original (Karypis et al, 1999).

Une méthode simple pour produire un couplage maximum consiste à visiter les sommets, dans un ordre aléatoire, et de fusionner chaque sommet avec l'un de ses voisins (sommets qui lui adjacent). Cette méthode, connue sous le nom du couplage aléatoire, est une méthode simple et efficace qui permet de limiter le nombre de niveaux (le nombre de graphes intermédiaires). Cependant, le but ultime du problème est de produire une partition qui minimise le nombre d'arêtes coupées (et non pas de réduire le nombre de niveaux). Pour ce faire, les auteurs ont proposé de déterminer un couplage à poids maximum qui renferme les arêtes ayants les poids les plus élevés. Cette méthode permet de réduire, d'une façon significative, le poids des arêtes dans le graphe compacté. Or, dans leurs travaux antérieurs (Karypis et al., 1995), les auteurs ont démontré que le poids des arêtes coupées est proportionnel au poids total des arêtes dans le graphe. De ce fait, en supprimant les arêtes de plus grande pondération, ils réduisent

le nombre d'arêtes coupées dans la partition. Dans cette nouvelle méthode, le couplage à poids maximum est évalué en utilisant le même principe du couplage aléatoire. Les sommets sont encore visités dans un ordre aléatoire. Mais au lieu de fusionner un sommet avec l'un de ses voisins non marqués, il est fusionné avec le voisin non marqué incident à l'arête ayant le plus grand poids.

2.4.2.5.2 L'opérateur de bissection : Algorithme glouton de recherche en largeur

Une méthode simple pour bi-partitionner un graphe consiste à choisir un sommet aléatoire et d'envelopper la région à son alentour en procédant par une recherche en largeur. Cette méthode se termine quand la moitié des sommets est incluse dans la classe (ou la moitié du poids total des sommets). La qualité de la solution obtenue est très sensible au choix du sommet initial. En effet, des choix différents de ce sommet conduisent à un nombre différent d'arêtes coupées. Pour résoudre partiellement ce problème, l'algorithme est lancé pour un certain nombre de fois. La partition ayant le minimum d'arêtes coupées est choisie comme solution initiale. Dans p-Metis, les auteurs ont amélioré cette méthode en utilisant la notion de gain. Au lieu d'insérer à chaque itération aléatoirement un sommet voisin à ceux déjà dans la classe, ils insèrent le sommet qui permet de diminuer au maximum (ou d'augmenter le minimum) le nombre d'arêtes coupées. Cette façon de faire a montré son aptitude à obtenir des partitions de qualité en un laps de temps très réduit.

2.4.2.5.3 L'opérateur de raffinement : KL avec sommets frontières

L'opérateur de raffinement de la solution est appliqué pendant la phase de décompression, juste après la projection des sommets d'un niveau donné G_m à un niveau inférieur G_{m-1}. Cet opérateur permet d'améliorer la partition projetée qui est déjà d'une bonne qualité. L'algorithme de raffinement utilisé, dans le cas de p-Metis, n'est que l'algorithme KL avec réduction de l'espace de recherche. Dans cet algorithme, seuls les sommets incidents à une arête coupée (sommets frontière) sont déplacés. Ainsi,

contrairement à l'algorithme KL où les gains de tous les sommets sont calculés, dans l'algorithme *KL frontière* seuls les gains des sommets frontières sont pris en compte.

Dans p-Metis, une itération se termine dès que 50 échanges sont exécutés et qu'aucune amélioration n'est apportée sur le nombre d'arêtes coupées. Cette façon de faire permet de réduire le temps d'exécution de l'algorithme KL surtout lorsqu'il suffit d'un nombre réduit de sommets pour améliorer la solution. En effet, d'après certaines expériences, les auteurs ont remarqué que dans la majorité des cas une itération cesse d'améliorer la solution une fois que le nombre de sommets déplacés atteint environ moins de 5% de la totalité des sommets. De plus, d'autres expériences ont montré que la plus grande amélioration, en nombre d'arêtes coupées, est obtenue pendant la première itération de l'algorithme. Ainsi, si le nombre d'itérations, à chaque niveau de l'algorithme multi-niveaux, est limité à une seule série d'échanges (une seule itération), le temps d'exécution se trouvera réduit d'un facteur allant de 2 à 4 (Karypis et al., 1999).

Dans le but de limiter davantage le temps d'exécution de l'opérateur de raffinement (KL frontières), les auteurs ont eu l'idée de combiner l'opérateur de raffinement avec une seule itération (KL1) avec l'opérateur à plusieurs itérations (KL). Il s'agit d'appliquer l'opérateur normal (à plusieurs itérations) tant que le graphe est de taille réduite. Une fois que la taille du graphe s'élargit, il convient de passer à l'autre opérateur (KL1). La condition nécessaire au passage d'un opérateur à un autre est directement reliée à la taille du voisinage. Ainsi, l'opérateur KL est appliqué tant que le nombre de sommets frontières est inférieur à 2% du nombre total des sommets du graphe original. Autrement, l'opérateur KL1 est utilisé. Cette condition représente un bon compromis entre le taux d'amélioration et le temps qu'il faut y consacrer.

2.4.2.5.4 Les résultats expérimentaux

Suite aux expérimentations de l'algorithme p-Metis sur différents graphes tests (Walshaw et al., 2006), il s'est avéré que les différents opérateurs de l'algorithme multi-niveaux sont caractérisés par une grande rapidité. En analysant le temps nécessaire à p-Metis pour produire des partitions de 256 classes (machines avec 1.2GB de mémoire et

des processeurs de 200MHz), les auteurs disent que dans le cas du plus petit graphe (add32, |V|=4960, |E|=9462) le temps total ne dépasse pas 1.63 sec, alors que dans le cas du plus grand (auto, |V|=448695, |E|=3314611) le temps est de 179 sec. De plus, en comparant la qualité de la solution obtenue par p-Metis avec les meilleurs résultats connus de nos jours (Walshaw et al., 2006), nous remarquons que, pour des partitions de 64 classes, l'écart, en nombre d'arêtes coupées, ne dépasse pas 10% (en défaveur de p-Metis).

En conclusion, nous pouvons qualifier p-Metis d'algorithme rapide. Dans leur conception, les auteurs ont su trouver un bon compromis entre la qualité de la solution et le temps qu'il faut y allouer. En effet, ils ont jugé inutile de perdre davantage de temps pour essayer d'améliorer la solution de 1 ou 2%.

De plus, p-Metis a permis de montrer la capacité du concept multi-niveaux à produire des partitions de qualité en un laps de temps limité.

2.4.2.6 k-METIS : Algorithme multi-niveaux pour le k-partitionnement de graphe (k-way)

Le *k*-partitionnement de graphe est dans la majorité des cas résolu en procédant par bissection récursive. Dans de telles procédures, nous obtenons initialement un bi-partitionnement du graphe. Ensuite, en appliquant récursivement le même algorithme sur chacune des partitions, nous obtenons après *k* itérations le nombre de partitions voulues. Contrairement à leur algorithme *p*-Metis, Karypis et al., (1998) ont présenté dans *k*-Metis un algorithme qui permet directement de partitionner un graphe en *k* classes. Cette façon de procéder comporte de nombreux avantages. En effet, le graphe est maintenant compressé une seul fois (au lieu de *k* fois), réduisant ainsi la complexité de $O(|E| \log(k))$, à $O(|E|)$. De plus, il est connu qu'un bi-partitionnement récursif produit, dans les meilleurs cas, des partitions d'aussi bonne qualité qu'un *k*-partitionnement direct (Karypis et al., 1998). En effet, le bi-partitionnement récursif se limite à résoudre une réduction du problème et non pas à la résolution du problème global.

L'algorithme *k*-Metis est largement inspiré de son prédécesseur *p*-Metis. En effet, à l'exception des opérateurs : partitionnement initial et raffinement de la solution, le reste des opérateurs n'ont pas connu de changements majeurs.

2.4.2.6.1 Opérateur de compression : Couplage à poids maximum* (CPM*)

D'après les tests effectués sur *p*-Metis, les auteurs se sont aperçus qu'une bonne partition se transmet des graphes compactés au graphe original, lorsque la moyenne de degrés des graphes compactés est assez petite. L'opérateur de compression est une modification de l'opérateur de couplage à poids maximum (*CPM*). Cet opérateur essaye de réduire les degrés des graphes compactés.

L'algorithme de compression *CPM** est basé sur une procédure randomisée similaire à celle de *CPM*. En effet, les sommets sont encore visités dans un ordre aléatoire. Un sommet visité *u* est fusionné avec un sommet *v* lorsque l'arête (*u*, *v*) possède le poids maximum parmi toutes les arêtes incidentes à *u*. La seule modification apportée concerne le cas particulier où plusieurs arêtes incidentes au sommet visité *u*, possèdent un même poids maximum. Dans ce cas, *u* est fusionné avec le sommet *v* qui partagent avec lui le plus grand nombre de voisins.

D'après différentes expériences, l'algorithme *CPM** a monté son aptitude à produire de meilleures compressions que *CPM* et ceci pour une même complexité $O(|E|)$. La moyenne des degrés des graphes compressés par *CPM** est moins élevé que celle obtenue par *CPM*.

2.4.2.6.2 L'opérateur de partitionnement initial

Dans *k*-METIS, les auteurs ont utilisé l'algorithme multi-niveaux de *bissection* récursive *p*-METIS comme opérateur de partitionnement initial.

2.4.2.6.3 L'opérateur de raffinement : algorithme global de Kernighan-Lin

L'algorithme global de Kernighan-Lin est une variante de l'algorithme *KL* adapté pour le problème du *k*-partitionnement de graphes. Cet algorithme utilise une queue de priorité qui contient des sommets triés selon leurs gains. Le gain du déplacement d'un sommet *u* de la classe *i* à la classe *j* est la différence entre le nombre de ses voisins (sommets qui lui sont adjacents) dans *i* et le nombre de ses voisins dans *j*. À chaque déplacement d'un sommet, l'algorithme sélectionne le sommet de plus grand gain et vérifie ensuite si le déplacement de ce sommet ne viole pas deux conditions sur la taille des classes. La première condition porte sur la taille maximale qui ne doit pas dépasser $C|V_0|/k$ (C est le déséquilibre des classes). La seconde condition porte sur la taille minimale des classes qui ne doit être inferieure à $0.9|V_0|/k$. Dans le cas où les conditions sont respectées, le sommet est déplacé vers la nouvelle classe même si son gain est positif (n'améliore pas la fonction objective). L'algorithme continue ainsi à déplacer des sommets jusqu'à atteindre *x* déplacements sans amélioration de la fonction du coût. Dans ce cas, les derniers *x* mouvements seront ignorés. Une fois qu'un sommet est déplacé, ce dernier n'est plus considéré dans la même itération. Ce processus peut être répété pour un certain nombre d'itérations ou jusqu'à ce qu'il y ait convergence.

2.4.2.6.4 Les résultats expérimentaux

En comparant les performances de *k*-Metis et *p*-Metis, il s'est avéré que la méthode de *k*-partitionnement direct est beaucoup plus rapide que celle de la bissection récursive. A titre d'exemple, dans le cas du graphe auto (|V|=448.695, |E|=3.314.611), *k*-Metis n'a besoin que de 39.67 sec pour partitionner le graphe en 256 classes, alors que *p*-Metis met 179.15 sec (1.2GB et 200MHz). La qualité des partitions produites par l'algorithme du *k*-partitionnement est comparable à celle obtenue par l'algorithme récursif, voire meilleure.

L'efficacité de *k*-Metis est reliée à un certain nombre de facteurs. En effet, en testant l'algorithme avec les différentes procédures de compression *(CPM ou CPM*)*, il

s'est avéré que, dans la majorité des cas, il existe un écart de 30% entre la partition initiale et la partition finale. Cela s'explique par le fait que la procédure de compression produit une excellente réplique réduite du graphe original et l'algorithme de bissection récursif produit une partition initiale de bonne qualité.

2.4.2.7 L'algorithme JOSTLE

Lors de la conception de cet algorithme, les auteurs Walshaw et al., (2000) se sont basés sur deux observations qui ont pour effet d'améliorer la qualité de la solution. La première consiste à accorder un léger déséquilibre à la taille des classes et la seconde consiste à améliorer la qualité de la solution de façon progressive de façon à ce que la partition produite à un niveau donné (du processus multi-niveaux) soit meilleure que celle qui la précède (parce qu'elle possède plus de degrés de liberté). L'idée est qu'en accordant un certain déséquilibre au graphe le plus compact, la partition sera probablement meilleure que si le déséquilibre ait été durement respecté. Ceci est possible à condition que la valeur de déséquilibre soit dégradée progressivement durant la phase de décompression de telle sorte que la partition finale ne soit pas affectée (déséquilibrée). Cependant, il faut noter que si le graphe le plus réduit commence avec une partition de bonne qualité mais avec un déséquilibre élevé, alors la qualité de la partition serait probablement dégradée pendant qu'on essaye d'équilibrer les classes.

JOSTLE est un algorithme multi-niveaux qui utilise l'algorithme bissection récursive de Kernighan-Lin (KL) comme opérateur de raffinement. Conformément à KL, ce dernier possède deux boucles imbriquées, une externe et une interne. La boucle externe se termine quant aucun déplacement n'est effectué pendant la boucle interne. Les sommets étant classés selon leurs gains, la boucle interne examine chacun de ces sommets en commençant par ceux ayant les gains les plus élevés. Si la condition sur la taille des classes est respectée et si la valeur de la fonction du coût est améliorée, les sommets seront alors transférés de leurs classes vers d'autres.

Pour comprimer un graphe et réduire ainsi sa taille, les auteurs ont choisi d'utiliser le même opérateur de compression que celui de p-Metis. Ainsi, en parcourant

les sommets en ordre aléatoire et en fusionnant les sommets non-marqués avec leurs voisins ayant les arêtes les plus pondérées, cet opérateur a montré son efficacité à produire des graphes compacts qui préservent le mieux les caractéristiques du graphe original. La phase de compression se poursuit jusqu'à ce que le nombre de sommets dans le graphe final soit aussi petit que le nombre de classes voulues dans la partition. Le partitionnement initial se résume ainsi à l'assignation de chaque sommet à une classe différente.

Dans le but de profiter d'une complexité linéaire, les auteurs ont choisi d'utiliser le tri par paquet (bucket sorting). Ce type de tri représente un mécanisme essentiel pour ajuster les sommets selon leurs gains de façon efficace et rapide. L'idée principale consiste à rassembler tous les sommets ayant un même gain dans un seul paquet. Le choix d'un sommet avec un gain maximal consiste simplement à trouver le paquet, non vide, ayant le plus grand gain et de sélectionner l'un de ses sommets. Suite au déplacement d'un sommet d'une classe à une autre, les gains des sommets voisins sont ajustés et la liste des sommets est triée de nouveau. Cette opération consiste simplement à recalculer les gains des sommets affectés et à les changer de paquets si leurs nouveaux gains diffèrent des anciens.

Comme nous l'avons expliqué, JOSTLE essaye d'exploiter l'observation expérimentale selon laquelle, en accordant un certain déséquilibre au graphe le plus compact et en dégradant par la suite ce déséquilibre au fur et à mesure des étapes de décompression, la partition finale serait préalablement meilleure que si le déséquilibre a été durement respecté. La mise au point de ce concept nécessite le développement d'un mécanisme qui permet d'équilibrer la charge entre les classes de façon équitable. Ceci faisant, il faut essayer de ne déplacer que les sommets entre classes adjacentes de façon à ne pas trop dégrader la qualité de la partition. L'algorithme choisi par les auteurs est un algorithme diffusif qui dérive de la méthode de minimisation de la norme euclidienne des poids transférés. Elle consiste à résoudre le système $Lx = b$, où L est la matrice de Laplace et $b_i = |P_i| - \lceil |V| / k \rceil$ est l'écart entre le poids de l'ensemble P_i et le poids optimal des classes. Cette méthode permet ainsi d'évaluer le poids à transférer entre les

classes adjacentes et la technique d'optimisation (algorithme KL) permet par la suite de déterminer les sommets à déplacer.

Dans le but d'appliquer la deuxième observation qui consiste à améliorer progressivement la partition durant les étapes de compression, les auteurs ont conçu une fonction cible qui décroît graduellement en fonction du nombre de sommets dans le graphe compacté. Si les poids de toutes les classes de la partition sont inférieurs aux poids projeté par la fonction, alors on dira que la partition est équilibrée. Dans ce cas, l'opérateur d'optimisation peut se concentrer exclusivement sur le raffinement de la solution. Par contre, si le poids d'une des classes de la partition est supérieur au poids projeté, alors l'opérateur devra se concentrer sur l'équilibrage des classes (avec un certain respect du raffinage). Il faut noter que la fonction de calcul du poids maximum est une fonction arbitraire qui doit être déterminée de façon rigoureuse. En effet, si elle décroît rapidement, l'équilibrage de la partition peut provoquer une dégradation de la qualité de la solution. Par contre, si elle décroît lentement, il se peut qu'on n'observe jamais les avantages associés à une tolérance au déséquilibre élevée.

Résultats expérimentaux

Dans le but de montrer la qualité des partitions obtenues par leur algorithme, les auteurs C. Walshaw et M. Cross ont choisi de comparer leurs résultats à ceux de Metis (k-Metis version 2.0.6).

En négligeant les détailles d'implémtation, les différences majeures entre ces deux algorithmes résident dans le fait que Metis compacte le graphe, durant le processus multi-niveaux, jusqu'à atteindre un nombre de sommets égale à 2000. Par la suite, il procède à sa partition de façon équilibrée. Par contre, Jostle compacte le graphe en P sommets (P est égale au nombre de classes dans la partition) et utilise ensuite ses opérateurs de balancement (équilibrage) et de raffinement progressif.

Ceci étant, il s'avère que Jostle permet d'améliorer les partitions de 6,2% en moyenne. Cependant, le temps d'exécution est beaucoup plus élevé. En effet, le temps nécessaire pour partitionner les graphes est multiplié d'un facteur allant de 1 à 10,

excepté certains cas particuliers.

2.4.2.8 L'algorithme JOSTLE Evolutionary

Dans cet algorithme les auteurs Soper et al., (2004) ont combiné la technique de recherche évolutionnaire avec la méthode multi-niveaux. La technique évolutionnaire est employée en construisant une population de graphes qui ne différent du graphe original que par les pondérations sur les arêtes et les sommets. Alors que l'opérateur multi-niveaux (*JOSTLE*) est utilisé comme une boite noire, afin de partitionner les graphes modifiés. Les partitions trouvées sur les graphes modifiés devraient être des bonnes partitions du graphe original. L'évolution de la population est assurée par l'application de deux opérateurs : un opérateur de croisement qui combine les informations de deux ou plusieurs partitions sélectionnés aléatoirement et un opérateur de mutation qui ne permet de modifier qu'une seule partition à la fois.

L'idée de base consiste à accorder à chaque sommet du graphe une fraction supplémentaire (ou biais) à son poids (≥ 0) et à ajouter à chaque arête la somme des fractions des sommets qui lui sont incidents. Ces fractions auront un grand effet sur les résultats de l'opérateur de compression, puisque les arêtes les plus pondérées seront supprimées en premier lieu. Les sommets ayant les plus petites fractions auront ainsi beaucoup plus de chance d'avoir des arêtes coupées.

Afin de générer un nouveau graphe, l'opérateur de croisement sélectionne un certain nombre de partitions de la population (trois ou quatre). Ensuite, pour chaque sommet du graphe, il examine si ce dernier possède des arêtes coupées (sommet frontière) dans au moins une des partitions considérées. Dans le cas affirmatif, il assigne à ce sommet, dans le nouveau graphe, une fraction choisie aléatoirement entre 0 et 0.01. Dans le cas contraire, il lui assigne la valeur de 0.1 plus une valeur aléatoire du même intervalle. Cette méthode représente une technique d'incitation qui encourage l'opérateur de partition à couper les arêtes qui ont été coupées dans les partitions parentes. De ce fait, les sommets qui ont des arêtes coupées, dans les partitions parentes, vont probablement se trouvera aussi avec des arêtes coupées dans la nouvelle partition.

Il faut noter que l'intervalle des fractions est choisi de sorte qu'il ne renferme que des valeurs très petites, ceci afin de ne pas trop modifier la propriété du graphe original et ainsi s'assurer que la partition générée sera une bonne partition du graphe original.

Un graphe est peut-être aussi généré par l'application de l'opérateur de mutation. Cet opérateur choisit une partition de la population. Ensuite, pour chaque sommet du graphe, il vérifie si ce dernier est un sommet frontière (sommet avec au moins une arête coupée), un voisin d'un sommet frontière ou un voisin d'un voisin d'un sommet frontière. Si c'est le cas, il lui accorde une fraction choisie aléatoirement du même intervalle [0, 0.01]. Sinon, il lui accorde une valeur de 0.2 plus une valeur aléatoire. Cette technique incite ainsi l'opérateur de partition à couper, non seulement les arêtes qui ont déjà été coupées auparavant mais aussi à couper les arêtes qui sont à proximité des sommets frontières.

Dans leur algorithme, les auteurs ont choisi de fixer le nombre de partitions dans la population à 50. À chaque génération, un ensemble de 50 nouvelles partitions est généré en procédant soit par croisement soit par mutation. Les candidats pour le croisement et la mutation sont choisis aléatoirement à partir de la population courante. Lors du remplacement des partitions parentes (population), les anciennes et les nouvelles partitions sont triées selon les valeurs de leurs fonctions du coût. Les 50 meilleures partitions sont ensuite sélectionnées pour former la nouvelle génération. Il faut noter que les individus récemment générés ne sont intégrés dans la population que si leurs coûts sont au moins meilleurs que ceux déjà dans la mémoire. Ce mode de remplacement élitiste est le plus approprié selon les auteurs car un grand nombre des partitions générées ne sont pas de très bonne qualité.

Pour générer la population initiale, l'algorithme accorde à chaque sommet un biais choisi aléatoirement de l'intervalle [0, 0.1] et utilise par la suite l'opérateur *Jostle* pour trouver une partition au graphe.

Résultats expérimentaux

Dans leurs expériences, les auteurs ont fixé le nombre de générations de l'algorithme évolutionnaire à 1000, soit 50000 exécutions de l'algorithme Jostle (puisque à chaque génération, 50 nouvelles partitions sont générés). Malheureusement, dans leur article, les auteurs n'ont pas présenté les durées d'exécution sous prétexte que les tests ont été effectués sur des machines différentes. Ils mentionnent seulement que pour les plus grands graphes le temps d'exécution peut prendre plusieurs jours.

En ce qui concerne la qualité des partitions obtenues, la plupart des résultats sont meilleurs que ceux obtenus par Jostle. L'écart varie de 10% à 26% et peut même atteindre 73% pour certains graphes.

2.4.2.9 L'algorithme Iterated JOSTLE

L'algorithme Iterated JOSTLE (Walshaw, 2004) est une version itérée de l'algorithme JOSTLE (Walshaw et al., 2000). Son principe de base consiste à partir d'une solution initiale et à fusionner, à chaque étape de la compression multi-niveau, un sommet avec un de ses voisins appartenant à une même classe. Lorsque le processus de compression est achevé, la partition du graphe compacté aurait ainsi le même coût que la partition du graphe original. En utilisant un algorithme de raffinement qui assure l'amélioration d'une partition initiale, la solution résultante de la procédure de raffinement multi-niveaux est nécessairement meilleure que l'initiale. Ainsi, en itérant les étapes de compression et de décompression et en utilisant à chaque itération la partition trouvée précédemment (à l'itération précédente), l'algorithme devrait ainsi converger vers une meilleure partition que l'initiale.

Le problème de cette procédure est que l'algorithme de raffinement n'est pas toujours assuré d'améliorer une partition initiale. Comme solution à ce problème, l'auteur utilise une procédure de compression qui se base sur un facteur aléatoire. Donc, à chaque lancement de cette procédure, avec une même partition, on obtient un graphe compacté différent. Lorsque l'opérateur de raffinement n'arrive pas à améliorer une

partition, cette même partition est réutilisée encore une fois comme partition initiale lors de la prochaine itération. L'opérateur de compression produit ainsi un nouveau graphe compacté, différent de celui trouvé à l'itération précédente, et l'opérateur de raffinement essayera encore une fois d'améliorer cette même solution initiale. Le nombre de fois qu'une solution initiale est réutilisée est utilisé comme critère d'arrêt de l'algorithme (c.-à-d., le nombre de fois que l'algorithme n'arrive pas à améliorer une solution).

Il faut bien remarquer qu'une itération de l'algorithme Iterated JOSTLE est une exécution complète de l'algorithme multi-niveaux Jostle (Walshaw et al., 2000). Ainsi, en partant d'une solution initiale, l'algorithme commence par compacter le graphe jusqu'à atteindre un certain seuil (un nombre fixé de sommets). Cette procédure crée ainsi une série hiérarchique de graphes dont chacun est une réduction du précédent. À l'étape suivante, l'algorithme décompresse les graphes compactés, en passant par la série hiérarchique dans le sens inverse. À chaque niveau de décompression, un algorithme de raffinement est appliqué pour améliorer la partition courante. Dans le cas où la partition du graphe original est meilleure que la partition initiale, celle-ci est utilisée comme solution initiale à l'itération suivante. Autrement, la même solution initiale de l'itération courante sera utilisée à l'itération suivante.

Résultats expérimentaux

Afin d'expérimenter les performances de son algorithme, Walshaw a lancé des tests sur les graphes connus de l'archive (Walshaw et al., 2006). En ne tenant pas compte des résultats de Jostle Evolutionary qui lui nécessite de nombreuses heures d'exécutions, l'auteur dit qu'il est arrivé à améliorer la plupart des partitions. Cependant, il ne précise pas le nombre d'exécutions qui lui ont permis de trouver ses résultats.

Pour ce qui est du temps d'exécution, nous n'avons que des informations sur les temps nécessaires aux partitionnements des graphes 4elt, DSJC1000.1, DSJC1000.5 et DSJC1000.9. Ces temps d'exécutions sont respectivement 4, 2, 11, 26 secondes.

2.4.2.10 L'algorithme PMSATS: Parallel multilevel simulated annealing and Tabu search

Cet algorithme développé par Banos et al (2004) est aussi une procédure multi-niveaux. Cependant, au contraire des précédentes, elle se base sur un algorithme de raffinement autre que l'algorithme de Kernighan-Lin. De plus, le mode d'exécution de cet algorithme se fait en parallèle. Le but de cette parallélisation n'est pas de réduire le temps d'exécution de la version en série mais plutôt d'optimiser le plus possible la qualité des partitions.

Comme toute procédure multi-niveaux, PMSATS comprend les trois phases: une phase de compression, une phase de partitionnement initial et une phase de décompression. Afin de réduire la taille des graphes, les auteurs ont utilisé la procédure de compression basée sur la notion du couplage à poids maximum mise au point par Karypis et al., (1999). Cet algorithme commence par déterminer un couplage maximal à poids maximum. Par la suite, il fusionne les sommets incidents aux arêtes de ce couplage. Après que la taille du graphe ait atteint un certain seuil, la phase suivante consiste à produire une partition initiale du graphe le plus réduit. Pour ce faire, les auteurs se sont aussi basés sur un autre algorithme de Karypis et al., (1999). Cet algorithme commence par assigner aléatoirement un sommet à une classe. Il procède ensuite, itérativement, en ajoutant à chaque classe de la partition les voisins du sommet qui est déjà dans la classe. Quand tous les voisins de ce sommet sont injectés, il passe aux voisins de ses voisins et ainsi de suite jusqu'à ce que la classe ait atteint la taille $|V|/K$. Une fois que la construction d'une classe est terminée, l'algorithme assigne aléatoirement un nouveau sommet à une nouvelle classe et répète le même processus. Lorsque la partition initiale est établie, l'algorithme passe à l'étape de décompression et de raffinement de la solution. Ainsi, comme nous l'avons déjà expliqué, cette étape représente une série de projections et d'améliorations successives. En commençant par le graphe le plus compact, l'algorithme améliore la partition (avec une procédure de recherche locale) et la projette ensuite sur le graphe de niveau supérieur. De même, la

partition projetée est elle aussi améliorée et projetée de nouveau sur le graphe de niveaux suivant. Ce processus continue jusqu'à ce que le graphe original soit obtenu.

La procédure de recherche locale de PMSATS est une combinaison des algorithmes recuit simulé et tabou. Les auteurs ont décidé d'incorporer au recuit simulé une liste de mouvements tabous afin de lui permettre d'échapper aux optimas locaux. Les mouvements pris en compte dans la recherche sont les mouvements qui impliquent des sommets ayant des arêtes coupées.

Comme nous l'avons déjà précisé, PMSATS est un algorithme qui s'exécute en mode parallèle. Le principe de base de cette parallélisation consiste à construire un ensemble de p solutions et d'appliquer (parallèlement) de différente façon l'algorithme multi-niveaux décrit précédemment. En utilisant des graines aléatoires différentes, chaque instance de l'algorithme produit un graphe compacté différent. La procédure de partitionnement initiale appliqué sur ces graphes produit aussi des partitions différentes puisque les sommets initiaux choisis aléatoirement sont différents. Chaque instance de l'algorithme commence ainsi avec sa propre solution initiale. De plus, pendant les phases de raffinements des solutions, chaque partition est améliorée en utilisant différents paramètres de la procédure de recherche locale. Ce qui fait que chaque processeur améliore sa propre solution. À certaines itérations de la phase de raffinement, un mécanisme de sélection élitiste est appliqué dans le but de continuer la recherche sur les meilleures partitions. Ce mécanisme sélectionne les meilleures solutions trouvées, à une même itération, et continue la recherche là-dessus.

Résultats expérimentaux

En comparant les résultats des exécutions de PMSATS sur les différents graphes de l'archive (Walshaw et al., 2006), les auteurs disent qu'ils ont arrivé à améliorer 40% des résultats de l'archive et ils ont obtenu 12% de solutions égales. Cependant, ils ne précisent pas, non plus, le nombre d'exécutions qui ont permis d'obtenir de telles solutions.

Au point de vue du temps d'exécution, les auteurs ne donnent pas de chiffres précis. Ils mentionnent seulement que les temps d'exécution peuvent varier de quelques secondes à plusieurs heures, selon les graphes testés.

Chapitre 3. ALGORITHMES PROPOSÉS

Pour essayer de trouver des bonnes approximations au problème de partitionnement de graphe, nous avons choisi de développer des algorithmes heuristiques. Pour choisir une catégorie d'algorithme parmi tant d'autres, nous nous sommes inspirés du problème de coloriage de graphe. En effet, il existe une grande similarité entre le problème du partitionnement et celui du coloriage (dans les deux problèmes les solutions sont des partitions). La catégorie d'algorithme qui s'est démarquée par son efficacité à obtenir de bonnes solutions dans le coloriage est celle des algorithmes hybrides (Galinier et al., 1999). L'idée essentielle de cette catégorie d'algorithme consiste à exploiter pleinement la puissance de recherche des méthodes de voisinage et de la recombinaison des algorithmes évolutifs sur une population de solutions (hybridation de deux types d'algorithme). Ainsi, dans notre étude, nous avons établi comme objectif principal de développer un algorithme hybride pour le problème de partitionnement de graphe.

Dans ce chapitre, nous allons présenter quatre différents algorithmes : un algorithme glouton CoverGreedily, un algorithme de recherche locale TabuGP et deux algorithmes, TabuGPG et MA-GP, qui incorporent des mécanismes de guidage de la recherche. L'algorithme CoverGreedily servira à produire les partitions initiales des trois autres algorithmes. L'algorithme TabuGP servira d'opérateur de recherche locale dans les algorithmes TabuGPG et MA-GP. L'algorithme TabuGPG est un algorithme à deux phases de recherche : Une phase de raffinement et une autre de guidage. L'algorithme MA-GP est un algorithme hybride (mémétique) qui se base sur un opérateur de recombinaison pour produire des partitions avec de nouvelles potentialités.

3.1 Algorithme glouton de couverture de graphe : CoverGreedily

Un algorithme glouton est, par définition, une procédure constructive qui commence avec une solution partielle (ou vide) et qui à chaque étape étend la solution. Les différents éléments possibles pour compléter la solution (candidats) sont évalués par un score. Le candidat ayant le meilleur score est choisi. La procédure continue jusqu'à ce que la solution soit complète. Dans notre algorithme, nous cherchons à concevoir une procédure qui permet de produire une partition dont les classes renferment des ensembles de sommets connexes. Ainsi, au contraire de la définition d'un algorithme glouton, qui stipule qu'il faut choisir le meilleur candidat à chaque étape, nous allons ajouter des sommets dans les classes en nous basant uniquement sur leur connectivité. Notre algorithme est donc une procédure simplifiée d'un algorithme glouton qui n'utilise aucune fonction de score. Pour construire une partition, nous allons simplement ajouter aux classes des sommets adjacents à ceux appartenant déjà à la partition.

Le principe général de notre procédure "*semi-gloutonne*" consiste à partir d'une solution initiale partielle qui couvre un certain nombre de sommets. Ces sommets peuvent être considérés comme les centres ou les noyaux des classes puisque nous allons ajouter aux classes les sommets qui se trouvent à leurs alentours. Nous allons ensuite étendre la partition de sorte qu'elle couvre tous les sommets du graphe. Pour ce faire, nous allons insérer dans chacune des classes des sommets adjacents à ceux se trouvant déjà dans la classe. La solution résultante de ce processus est une solution complète mais non nécessairement légale. En effet, certaines classes peuvent avoir des tailles supérieures à la taille maximale permise. En conséquence, pour rendre légale la partition, nous allons dans l'étape subséquente transférer des sommets des classes excédant la taille maximale vers ceux que ne le sont pas. L'algorithme 10, ci-dessous, donne un aperçu sur les différentes étapes de notre procédure semi-gloutonne.

```
CoverGreedily (P)

Entrée : Un k-partitionnement partiel P = {P1,..., Pk}.
Sortie : Un k-partitionnement complet et légal.

   1. Sélectionner les centres (noyaux) des classes.
   2. Étendre la partition P sur tous les sommets du graphe.
   3. Rendre légale la partition P.
```

Algorithme 10: Schéma général de la procédure gloutonne de couverture de graphe.

Comme solution initiale, cet algorithme peut commencer avec une solution P, vide ou partielle. Dans le cas où la solution est vide, nous affectons à chacune des classes un sommet choisi aléatoirement. Afin de mémoriser l'ordre des insertions, nous allons marquer ces sommets comme les premiers sommets insérés dans leurs classes. Cependant, dans le cas où nous disposons d'une solution partielle, les sommets faisant partie de la partition vont tous être considérés comme les centres (ou noyaux) de leurs classes. Nous n'avons pas alors à ajouter des sommets aléatoirement. Nous allons seulement leurs attribuer des numéros que nous les considérerons ensuite comme un ordre d'insertion ou de priorité. Une fois tous les sommets du graphe couverts, nous nous baserons sur cet ordre pour ajouter les sommets aux classes. Nous allons ainsi commencer par insérer les voisins des sommets ayant les plus petits numéros avant de passer à ceux ayant des numéros plus grands. Cette façon de faire nous assure, d'une part, un parcours en largeur des sommets. D'autre part, elle nous permet de limiter le nombre d'arêtes coupées puisque nous ajoutons tous les voisins d'un sommet avant de passer à un autre. L'algorithme 11, ci dessous, récapitule les étapes de la sélection des centres des classes.

```
GetClassCenter (P)

Entrée : Un k-partitionnement P={P1,..., Pk} complet et illégal.
Sortie : Un k-partitionnement complet et illégal.

   Pour i = 1 à k, faire: // k égal au nombre de classes
      1. Si |Pi| = 0 alors :
         a. Choisir aléatoirement un sommet v∉P.
         b. Pi = Pi ∪{v}.
         c. Accorder à v le numéro 1 dans Pi.
      2. Sinon
         - Numéroter les sommets de Pi dans un ordre quelconque.
```

Algorithme 11: Procédure de sélection des centres des classes.

Après avoir fixé des sommets dans chacune des classes, nous allons maintenant étendre la partition de sorte qu'elle puisse couvrir tous les sommets du graphe. Pour ce faire, nous allons ajouter les sommets adjacents à ceux faisant déjà parti de la partition. En parcourant les sommets dans l'ordre de leur insertion, nous allons ajouter à chaque classe un sommet adjacent à un sommet faisant déjà partie de la classe. Lorsque tous les sommets adjacents à un même sommet dans une classe sont insérés, on passe au sommet qui suit dans l'ordre d'insertion. Cette procédure continue jusqu'à ce que tous les sommets appartiennent à une des classes de la partition.

Dans le cas des graphes non-connexes, il se peut qu'un certain nombre de sommets restent non-couverts à la suite de l'application des étapes énoncées jusqu'ici. En effet, puisque l'extension de la couverture se fait de proche en proche, les sommets non-connexes au graphe principal (la plus grande partie connexe du graphe) ne peuvent pas être inclus dans la partition. Afin de les annexer à la solution, nous allons procéder d'une façon aléatoire. Nous allons ainsi affecter chacun d'entre eux aléatoirement à une classe. L'algorithme 12, ci-dessous, décrit en détail les étapes que nous venons d'expliquer. Dans cet algorithme, la fonction *sommet_courant* (t_i, P_i) retourne le $t_i^{ème}$ sommet inséré dans la classe P_i.

ExtendPartition (P, G)

Entrée : Un graphe G=(V,E) et un k-partitionnement partiel $P=\{P_1,\ldots, P_k\}$.

Sortie : Un k-partitionnement complet non nécessairement légal.

1. Pour i = 1 à k, faire :
   $t_i := 1$.
2. Tant qu'il existe (v, v')∈E tels que v'∈P et v∉P, faire :
   - Pour $i = 1$ à k, faire :
     - Tant qu'aucun sommet n'est inséré dans P_i et qu'il existe (v, v')∈E tels que v'∈P_i et v∉P, faire :
       a. v = sommet_courant (t_i, P_i).
       b. S'il existe (v, v')∈E tels que v∈P_i et v'∉P, alors :
          i. $P_i = P_i \cup \{v'\}$.
          ii. Marquer v' comme le $|P_i|^{ème}$ élément inséré dans P_i.
       c. Sinon Si $t_i < |P_i|$ alors :
          - Incrémenter t_i.
3. Assigner aléatoirement tout v∈V et v∉P.

Algorithme 12: Procédure d'extension de la couverture.

Comme on peut le remarquer de la procédure d'extension de la couverture, nous n'appliquons aucune contrainte sur la taille des classes. En effet, puisque nous ajoutons un sommet à la fois dans chacune des classes, la partition résultante devrait être, dans le meilleur des cas, complète et légale. Cependant, ce n'est malheureusement pas toujours le cas puisque les sommets voisins à certaines classes peuvent tous se trouver appropriés par d'autres classes. Ainsi, lorsqu'on veut étendre la couverture, certaines classes ne peuvent plus prendre d'autres sommets et leur taille se trouve ainsi réduite à un nombre limité de sommets. La partition résultante se trouve, quant à elle, déséquilibrée avec des classes de petites tailles et d'autres avec des tailles plus grandes. Afin de l'équilibrer et la rendre légale, nous allons appliquer l'algorithme 13 ci-dessous. Dans ce dernier, nous parcourons toutes les classes excédant la taille maximale permise. Pour chacune d'entre

elles, nous déplaçons un certain nombre de leurs sommets, vers les petites classes, de sorte que leur taille soit égale à la taille maximale. Afin dégrader le moins possible la partition, nous allons évaluons le coût des déplacements des sommets et nous choisissons à chaque fois le sommet ayant le meilleur coût. Lorsqu'une classe est rendue légale, on passe à la suivante. L'algorithme se termine lorsque toute la partition est rendue légale.

LegalizePartition (P)

Entrée : Une partition $P=\{P_1,\ldots,P_k\}$ complète et illégale.
Sortie : Une partition complète et légale.

Pour i allant de 1 à k, faire :

Tant que $|P_i| < max\_size$, faire :

a. Pour *tout* sommet $v \in P_i$, faire :

- Pour j allant de 1 à k, faire :
  - Si $|P_j| < max\_size$ et $i \neq j$, alors :
    - Évaluer le coût du déplacement de v de i vers j.

b. Effectuer aléatoirement un mouvement parmi ceux de meilleur coût.

Algorithme 13 : Procédure de légalisation des solutions.

3.2 Algorithme Tabou: Tabu-GP (Tabu Graph Partitioning)

Comme mentionné dans le chapitre précédent, la recherche Tabou est une métaheuristique itérative qualifiée de *recherche locale* au sens large. Étant donné un voisinage $N(s)$ et une fonction d'évaluation f, le principe de cette métaheuristique consiste à choisir, à chaque itération, la meilleure solution $s' \in N(s)$, même si $f(s') > f(s)$. Pour éviter de revenir immédiatement à s (puisque s est meilleure que s'), elle utilise une liste T, nommée *Liste Tabou*, qui mémorise les dernières solutions visitées et interdit tout déplacement vers une solution de cette liste. Les solutions ne demeurent dans T que pour un nombre limité d'itérations. Dans cette partie du rapport, nous allons appliquer l'algorithme tabou au problème de partitionnement de graphe. Nous allons commencer par définir ses concepts et décrire ses différentes étapes. Nous allons ensuite présenter les aspects les plus importants de ses structures de données. Dans une dernière section, nous allons présenter des techniques additionnelles permettant d'améliorer ses performances.

3.2.1 Définition de l'algorithme Tabu-GP

Un algorithme tabou est une procédure générique qui peut être adaptée à différents problèmes. Afin de concevoir un algorithme tabou dédié à un problème particulier, il faut ainsi l'adapter au problème en question. Pour ce faire, il faut définir chacun des concepts suivants : l'espace des configurations, la fonction d'évaluation des configurations, la fonction de voisinage et le mécanisme de mise à tabou des configurations visitées. Dans cette section, nous allons définir les concepts de notre algorithme tabou pour le problème de partitionnement de graphe. Nous allons ainsi définir les configurations qu'il manipule, la fonction d'évaluation de ces configurations, les mouvements qui peut effectuer à chaque itération et les mouvements qui sont interdits.

3.2.1.1 Définition d'une configuration

Les configurations permises par notre algorithme sont les partitions légales du problème. Une partition est dite légale lorsque tous les sommets d'un graphe sont répartis sur des ensembles disjoints qui ont des tailles inférieures à une valeur fixée. Plus formellement, étant donné un graphe $G = (V, E)$, un entier k ($k \geq 2$) et un réel x ($x \geq 0$), une configuration est définie comme une subdivision de V sur k sous-ensembles $P_1, P_2, \ldots, P_k$ qui satisfait les contraintes suivantes :

(1) $P_i \cap P_j = \varnothing \qquad \forall i, j \in [1 \ldots k] \; et \; i \neq j.$

(2) $\bigcup_{i=1}^{k} P_i = V.$

(3) $|P_i| \leq \left(1 + \frac{x}{100}\right) \left\lceil \frac{|V|}{k} \right\rceil \qquad \forall i \in [1 \ldots k].$

La contrainte (1) signifie que toutes les classes doivent être disjointes deux à deux. La contrainte (2) signifie que tout sommet du graphe doit appartenir à une classe P_i. La contrainte (3) impose une taille maximale aux classes qui doit être inférieure ou égale $(1 + x\%) \lceil |V|/k \rceil$ (La fonction $\lceil a \rceil$ retourne le plus petit entier $\geq a$). Le réel x de la contrainte (3), nommée *" déséquilibre des classes"*, indique un écart par rapport à la taille optimale $\lceil |V|/k \rceil$.

3.2.1.2 Définition de la fonction d'évaluation

Une partition P définit aussi un ensemble d'arêtes $E_c = \{(u, v) \in E \mid u \in P_x, v \in P_y, x \neq y\}$. Une arête e appartient à cet ensemble lorsque ses extrémités n'appartiennent pas à un même sous-ensemble de la solution. Une telle arête est dite coupée. La fonction objectif f de notre algorithme est la même que celle du problème. Elle est égale au nombre d'arêtes coupées $|E_c|$.

3.2.1.3 Définition du voisinage d'une configuration

Le voisinage d'une configuration P est l'ensemble des configurations pouvant être obtenues par l'application de modifications particulières (mouvements) sur P. Dans *TabuGP*, nous définissons un mouvement comme le déplacement d'un sommet v, d'une classe P_i, vers une autre classe P_j qui contient au moins un sommet v' adjacent à v. Il

faut noter que deux sommets adjacents appartenant à deux classes différentes sont nécessairement liés par une arête coupée. Dans les parties suivantes de notre description, nous allons noter le déplacement d'un sommet v vers une classe i par m (v, i) et la configuration voisine à P, résultante du mouvement m (v, i), par $P \oplus m$ (v, i).

Nous avons choisi d'effectuer les mouvements m (v, i) parce qu'ils représentent les seuls mouvements élémentaires (mouvements composés d'un seul sommet) pouvant diminuer la fonction du coût. Tout autre mouvement élémentaire ne peut que créer de nouvelles arêtes coupées et augmenter ainsi la fonction du coût. De plus, le nombre des mouvements m (v, i) est très réduit par rapport au nombre de mouvements élémentaires possibles dans une partition. En se limitant aux mouvements m (v, i), nous réduisons ainsi le nombre de mouvements testés à chaque itération.

3.2.1.4 Définition de la liste Tabou

La liste Tabou est un mécanisme qui interdit de revenir sur les dernières configurations explorées. Cette interdiction demeure pour un certain nombre d'itérations *It_Tabu.* Après l'application d'un mouvement m (v, i), nous allons interdire (d'où le nom de *tabou*) le mouvement inverse qui consiste à remettre le sommet v dans sa classe initiale. Supposant j la classe initiale de v, nous allons ainsi rendre tabou le mouvement m (v, j).

3.2.2 Description de l'algorithme Tabu-GP

Pour partitionner un graphe donné $G = (V, E)$, l'algorithme *TabuGP* a besoin d'avoir en entrée : le graphe lui-même, le nombre k de classes voulu dans la partition, la taille maximale (*P_max*) d'une classe et une partition initiale P de G. La partition initiale P doit faire partie de l'ensemble des configurations permises par l'algorithme (section 3.2.1.1). En sortie, l'algorithme renvoie la meilleure partition P^* trouvée au cours de la recherche.

En partant de la partition initiale comme configuration courante, l'algorithme parcourt, à chaque itération, toutes les classes et vérifie la taille de chacune d'entreelles. Dans le cas où la taille d'une classe i est strictement inférieure à la taille maximale

P_max, il évalue les gains des mouvements non-tabous. Par définition, le gain d'un mouvement $m(v, i)$ vaut $\delta(v, i) = f(P \oplus m(v, i)) - f(P)$. Une valeur strictement négative (respectivement positive) de ce gain indique que le mouvement $m(v, i)$ permet de diminuer (respectivement augmenter) la fonction du coût. Dans le cas où la taille d'une classe est égale à la taille maximale, l'algorithme ignore cette classe et n'évalue pas les mouvements qui lui sont associés. En effet, le déplacement d'un sommet v vers une classe i de taille *P_max* causerait la violation de la contrainte imposée sur la taille des classes (la taille de la classe i devient égale à *P_max* + 1). Une fois que les gains de tous les mouvements permis sont évalués, l'algorithme effectue aléatoirement un mouvement $m(v^*, i^*)$ parmi ceux du meilleur gain. Ensuite, il rend tabou le mouvement inverse $m(v^*, P(v^*))$ pour *It_tabu* itérations ($P(v^*)$ retourne la classe du sommet v dans la partition P). En dernière étape, l'algorithme effectue le mouvement choisi $m(v^*, i^*)$ et la configuration $P \oplus m(v^*, i^*)$ devient donc la solution courante. De plus, dans le cas où le coût $f(P \oplus m(v^*, i^*))$ de cette configuration est strictement inférieur au coût $f(P^*)$ de la meilleure solution P^*, l'algorithme mémorise cette configuration comme la meilleure solution trouvée au cours de la recherche. Lorsque l'algorithme finit de parcourir *It_max* itérations, il retourne la meilleure partition trouvée.

```
Entrée : Un graphe G = (V, E), un entier k > 1, un entier P_max,
         une partition initiale P = (P1,…, Pk) de G.

Sortie : Une partition P*.

Paramètres : It_max, It_tabu.

      Poser P* := P.
      Pour Iter = 0 à It_max, faire :
         Pour i = 1 à k, faire :
            Si |Pi| < P_max, alors:
                 Évaluer les gains δ(v, i) des mouvements m(v, i)
                 non-tabous.
      Choisir un mouvement m(v*, i*) de gain maximum.
      Rendre tabou le mouvement m(v*, P(v*)) pour It_tabu
      itérations.
      Poser P: = P ⊕ m(v*, i*).
      Si f(P) < f(P*), alors :
         P*:= P.
```

Algorithme 14 : Tabu-GP.

L'algorithme *TabuGP* comporte deux paramètres *It_max* et *It_tabu*. *It_max* est le nombre d'itérations. *It_tabu* est le nombre d'itérations pendant lesquelles un mouvement reste tabou. Ce nombre peut avoir une valeur constante qui est réglée pour chaque graphe. Cependant, le fait de varier *It_tabu* peut aider la recherche à trouver de meilleures solutions. Dans notre algorithme, *It_tabu* est variable. Nous allons dans le chapitre suivant expliquer la méthode utilisée pour fixer ses différentes valeurs.

3.2.3 Structures de données

La plus grande part du temps de calcul de *TabuGP* est consacrée, à chaque itération, à la recherche du meilleur mouvement. Une technique naïve, pour ce faire, consiste à évaluer les gains $\delta(v, i)$ de tous les mouvements $m(v, i)$ vers les classes P_i de taille $|P_i| < P\_max$ et de choisir le mouvement frontière de meilleur gain. Il faut noter

que pour un k-partitionnement $P = (P_1, ..., P_k)$, le gain $\delta(v, i)$ est égal à la différence entre le nombre de sommets adjacents à v dans $P(v)$ et le nombre de sommets adjacents à v dans P_i. Le gain $\delta(v, i)$ peut ainsi être calculé en $O(|\Gamma(v)|)$, avec $\Gamma(v)$ l'ensemble des sommets voisins à v. Cependant, il existe des implémentations beaucoup plus efficaces permettant de réduire la complexité du calcul. Notons $\gamma(v, i)$ le nombre de sommets appartenant à la classe i et adjacents à v, et admettons que de telles valeur soient conservées dans une matrice de dimension $|V|$ x k que nous appelons la matrice *gamma*. La matrice *gamma* peut être initialisée au début du processus de recherche et mise-à-jour, à chaque itération, après qu'un mouvement $m(v, i)$ est appliqué à la solution courante. Le gain du mouvement $m(v, i)$ peut être ainsi calculé en un temps constant en évaluant $\delta(v, i) = \gamma(v, i) - \gamma(v, P(v))$ et la recherche du meilleur mouvement peut être alors effectuée en $O(k\ |V|)$. De plus, il est aussi possible de déterminer, en temps constant, si un sommet v est adjacent à une classe i autre que $P(v)$ en vérifiant juste si $\gamma(v, i) > 0$. Une fois un mouvement effectué, la matrice *gamma* peut être mise-à-jour en $O(|\Gamma(v)|)$. Il suffit ainsi d'incrémenter d'une unité $\gamma(w, i)$ et de décrémenter d'une unité $\gamma(w, P(v))$, pour chaque sommet $w \in \Gamma(v)$.

Les mouvements tabous sont sauvegardés dans une matrice de deux dimensions $|V|$ x k que nous appelons la matrice T. Un élément $T(v, i)$ donne l'itération à laquelle $m(v, i)$ cesse d'être tabou (c.-à-d., $m(v, i)$ est tabou si et seulement si $T(v, i) \geq iter$). Il est ainsi possible de savoir en temps constant si un mouvement est tabou. De plus, lorsque un mouvement $m(v, i)$ est appliqué à la configuration courante, la mise à jour de T se fait aussi en temps constant en posant $T(v, i) = iter + It\_tabu$.

3.2.4 Technique d'intensification et de diversification de la recherche

Maintenant que les concepts de notre algorithme tabou sont établis, nous allons lui incorporer des techniques spéciales qui vont le rendre plus robuste et plus apte à trouver de bonnes solutions. Les techniques les plus utilisées pour de telles fins sont l'intensification et la diversification. L'idée à la base de l'intensification est qu'on

devrait explorer de façon plus approfondie les régions qui semblent les plus prometteuses. Complémentairement à cette idée, la diversification consiste à inciter l'algorithme à se diriger vers des régions qui n'ont pas encore été visitées. Il existe différents mécanismes et techniques permettant de simuler ces deux concepts. Cependant, le plus difficile est de trouver un équilibre entre les deux de sorte qu'ils s'aident mutuellement à trouver de bonnes solutions.

Dans un algorithme tabou, la longueur de liste Tabou est l'un des paramètres le plus critique à fixer. En lui attribuant une grande valeur, on prive la recherche d'un grand nombre de mouvements. L'algorithme est donc obligé de visiter de nouvelles régions de l'espace de recherche mais sans pour autant explorer en profondeur chacune d'entre-elles. Toutefois, en attribuant à la liste Tabou une longueur réduite, on incite l'algorithme à explorer en profondeur une certaine région de l'espace de recherche, mais une fois qu'un optimum local est atteint, la recherche risque de se bloquer aux alentours ce minimum. Ainsi, selon la valeur accordée au paramètre, l'algorithme peut avoir un comportement d'intensification ou de diversification mais pas les deux à la fois.

Afin de remédier à ce problème, nous allons alterner périodiquement ces deux techniques. En utilisant le paramètre tabou comme seul mécanisme d'intensification et de diversification, nous allons périodiquement lui attribuer des grandes valeurs pour inciter une diversification et des petites valeurs pour inciter une intensification. Les phases de diversification permettent ainsi de visiter de nouvelles régions de l'espace de recherche et les phases d'intensification d'explorer en profondeur chacune d'entre-elles. La figure 6 ci-dessous illustre notre procédure de variation de la liste Tabou. Cette procédure varie selon une fonction périodique du nombre d'itérations. Au cours de la recherche, elle permet d'attribuer périodiquement différentes valeurs à *It_tabu*. Dans la procédure de la figure 6, *It_tabu* peut prendre cinq différentes valeurs $T_1 < T_2 < T_3 < T_4 < T_5$. La plus grande valeur T_5 est fixée à une valeur T_{max}. Les autres valeurs T_i (i<5) sont fixées de sorte que chaque valeur soit à un multiple α (α=0.5) de la valeur qui lui est directement supérieure ($T_i = \alpha\ T_{i+1}$). Le nombre d'itérations, pendant lesquelles une valeur T_i est appliquée est un multiple β de T_{max} ($4xT_{max}$). Une période (ou un cycle) se

termine lorsque chaque valeur est appliquée pendant $\beta x T_{max}$ itérations. La durée d'une période est donc aussi un multiple de T_{max} ($\alpha\beta T_{max}=20xT_{max}$). De ce fait, T_{max} représente le seul paramètre de notre procédure.

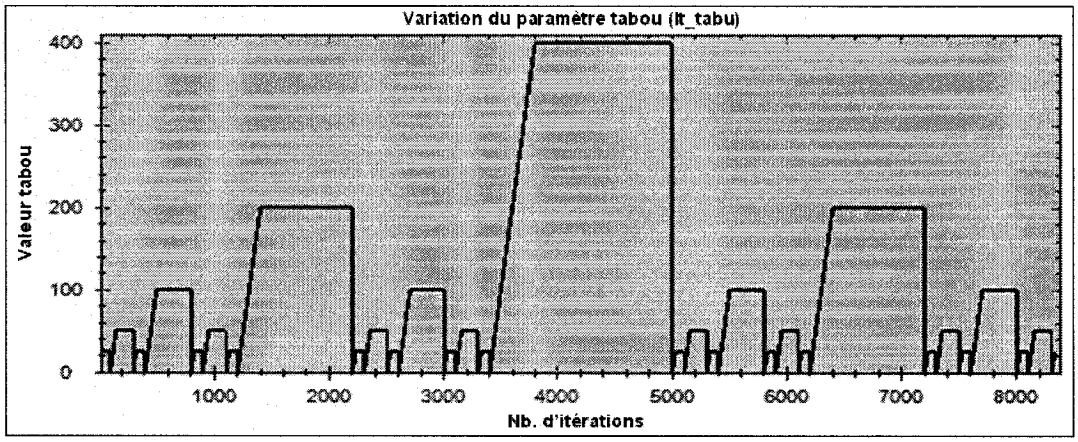

Figure 6 : Variation du paramètre Tabou (It_tabu)

La procédure de variation de la figure 6 est obtenue en faisant alterner les différentes valeurs de *It_tabu* à la manière d'une fractale (une courbe qui possède des détails similaires à des échelles arbitrairement petites ou grandes). Une méthode simple pour implémenter une telle procédure consiste à utiliser une fonction récursive. L'algorithme 15, nommé *Phase*, illustre notre fonction récursive pour la variation du paramètre tabou. Pour avoir le cycle de variation de la figure 6, on exécute l'algorithme *Phase* avec i=5 (5 phases). Le cycle peut être répété autant de fois jusqu'à ce qu'un nombre fixé d'itérations de recherche locale soit atteint.

```
Phase (i, P, T)
   Si i = 1, faire :
      - Exécuter Tabu-GP sur P avec It_max = βT et It_tabu = T.

   Sinon, faire:
      - Phase (i-1, P, αT).
      - Exécuter Tabu-GP sur P avec It_max = βT et It_tabu = T.
      - Phase (i-1, P, αT).
```

Algorithme 15 : Procédure de variation du paramétre Tabou.

Dans notre procédure, le nombre d'itérations accordés aux différentes valeurs *It_tabu* est proportionnel à la valeur elle-même. Les petites valeurs sont appliquées pendant un nombre réduit d'itérations, alors que les valeurs les plus grandes sont appliquées pour un nombre d'itérations plus important. Cette façon de procéder évite de piéger l'algorithme dans les optima locaux et permet de diversifier suffisamment la recherche. De plus, les petites valeurs sont toujours appliquées après les valeurs les plus grandes, ceci pour intensifier la recherche sur les nouvelles régions visitées (par les grandes valeurs). La seule difficulté dans notre procédure consiste à régler le paramètre T_{max}. En lui accordant une grande valeur, on risque de perturber la recherche. Alors qu'en lui attribuant des petites valeurs, on risque de restreindre la recherche à quelques régions de l'espace de recherche. Pour régler T_{max}, nous avons procédé d'une façon expérimentale (voir chapitre 4).

Le schéma de variation de la liste taboue proposé (figure 6) a permis d'obtenir de meilleurs résultats que d'autres schémas de variation testés (voir chapitre 4). Nous avons choisi ce schéma de variation car il représente une procédure simple (1 seul paramètre T_{max}) qui permet de varier systématiquement la longueur taboue pendant la recherche.

3.4.1 Algorithme Tabu-GP orienté: Tabu-*GPG* (Guided Tabu Graph Partitioning)

Tabu-GPG est un algorithme qui effectue une longue chaine de recherche locale mais qui n'utilise pas toujours une même fonction d'évaluation. Dans cet algorithme, on fait alterner deux types de phases de recherche : des phases durant lesquelles on utilise la fonction normale du problème (qui consiste à minimiser le nombre d'arêtes coupées) et d'autres durant lesquelles on utilise une fonction modifiée. La fonction modifiée a pour rôle de guider la configuration vers des zones favorables de l'espace de recherche : ceci sera précisé dans la suite.

L'algorithme 16, ci-dessous, illustre les différentes étapes de l'algorithme *Tabu-GPG*.

1. P := {}.
2. Construire une partition initiale P du graphe G.
3. P' := Local_search_operator (P, L, f_1)
4. répéter jusqu'à atteindre un nombre fixé d'itérations de recherche locale :
   (a) P := Local_search_operator (P', L, f_2)
   (b) P' := Local_search_operator (P, L, f_1).

Algorithme 16 : Les étapes de l'algorithme Tabu-GPG.

En première étape, Tabu-GPG commence par construire une solution initiale P. Cette solution est obtenue en appliquant la procédure gloutonne *CoverGreedily*. Il applique ensuite, sur P, un opérateur de recherche locale avec la fonction normale du problème f_1 (minimiser le nombre d'arrêtes coupées). L'opérateur de recherche locale utilisé est une version de Tabu-GP légèrement modifiée de celle présentée dans la section précédente. Au lieu de retourner la meilleure solution de la recherche, Tabu-GP retourne la dernière solution. L'algorithme alterne ensuite les deux phases de recherche :

Il applique sur P', pendant L itérations, la phase de recherche avec la fonction modifié f_2 et il applique sur P, pendant L itérations, la phase de recherche avec la fonction normale f_1. L'algorithme s'arrête lorsque le nombre total d'itérations de recherche locale (dans les deux phases) ait atteint une valeur fixée.

Nous faisons l'hypothèse que, compte tenu du type de graphe considéré dans nos jeux de donnée (graphes peu denses), une partition ne devrait pas contenir des classes composées de nombreuses composantes connexes. De même, une partition devrait généralement éviter des classes de forme allongée, c'est-à-dire dont le diamètre est trop important (le diamètre désigne la distance maximum entre deux sommets dans le sous-graphe induit par la classe). Au contraire, les classes devraient être connexes et de «forme compacte», c'est-à-dire telles que la distance entre les sommets les plus éloignés de la classe soit réduite. La fonction de coût modifiée devrait encourager la construction de classes possédant les propriétés désirées.

Cependant, plutôt que de définir explicitement une telle fonction, nous avons employé une technique beaucoup plus simple et qui nous semble revenir un peu au même – bien que ce ne soit pas réellement évident. Cette technique consiste à modifier le graphe du problème en lui ajoutant les arêtes à distance de deux. On ajoute ainsi une arête entre deux sommets non adjacents entre eux et adjacents à un même sommet tiers. Lors de l'évaluation d'un mouvement, l'algorithme tiendra ainsi compte, en plus des sommets à distance de un, des sommets à distance de deux dans le graphe original. Dans les deux phases de recherche, nous appliquons ainsi le même opérateur de recherche locale mais sur deux graphes différents. Lorsqu'on veut guider la recherche, on applique l'opérateur de recherche sur le graphe modifié. Lorsqu'on veut raffiner une partition, on applique l'opérateur de recherche sur le graphe de base.

Suite à des tests expérimentaux, nous avons observé que Tabu-GPG a permis d'améliorer les résultats de beaucoup de graphes. Nous avons été étonnés de la qualité des résultats obtenus. Nous nous sommes demandés si l'amélioration n'était pas plutôt dû à une autre cause, par exemple l'effet de diversification occasionné par l'utilisation de la fonction de coût auxiliaire. Pour tester si c'est le cas, nous avons, au lieu d'ajouter

les arêtes à distance de deux, ajouté le même nombre d'arêtes aléatoirement. Les résultats obtenus ont été dans les meilleurs des cas proches de ceux de l'algorithme Tabu-GP. L'ajout aléatoire des arêtes ne permet donc pas d'améliorer les solutions. Nous tendons à en déduire que la technique présentée possède effectivement la propriété de mieux guider la recherche. Dans nos expériences réalisées sur les graphes de l'archive, ceci est particulièrement vrai pour les graphes les moins denses (voir le chapitre 4).

3.4 Algorithme mémétique : MA-GP (Memetic Algorithm Graph Partitioning)

Un algorithme mémétique est une heuristique hybride qui fait évoluer un ensemble de solutions (individus), appelé population. Cet algorithme utilise deux opérateurs : un opérateur de recombinaison qui génère une nouvelle solution à partir de celles de la population et un opérateur de recherche locale qui améliore la solution générée. Dans MA-GP, on utilise Tabu-GP comme opérateur de recherche locale.

Le schéma général d'un algorithme mémétique est présenté dans l'algorithme 17. La première étape de l'algorithme consiste à initialiser une population $\mathcal{P}$ avec un ensemble d'individus (partitions). Pour un nombre fixé de générations, il répète ensuite les étapes suivantes : il sélectionne deux individus P' et P'' de la population (appelés individus parents). À partir de ces deux individus, il génère un nouvel individu P (individu enfant). Il applique sur P un opérateur de recherche locale, pendant L itérations. Enfin, il utilise P pour mettre à jour la population.

1. Initialiser la population $\mathcal{P}$ avec un ensemble d'individus.
2. Pour un nombre fixé de générations, répéter :
   (a) Sélectionner deux parents P' et P'' de la population.
   (b) Appliquer un opérateur de recombinaison sur P' et P'' pour générer un enfant P.
   (c) Appliquer sur P un opérateur de recherche locale pendant L itérations
   (d) Remplacer par P un individu de la population (mettre à jour la population).

Algorithme 17: Schéma général de l'algorithme mémétique.

Dans les sections ultérieures, nous allons compléter la description de MA-GP. Nous allons présenter chacune des procédures suivantes : L'initialisation de la population, la sélection des individus de la population, la recombinaison des individus et

la mise à jour de la population. Dans une dernière section, nous allons également présenter un opérateur d'évaluation de la diversité qui permet de calculer une mesure de similitude entre les partitions continues dans la population. Cet opérateur servira à fixer le nombre de générations de notre algorithme.

3.4.1 Initialisation de la population

Comme mentionné auparavant, la population $\mathcal{P}$ contient un ensemble d'individus. Chacun de ces individus est une partition légale du graphe étudié. Le nombre d'individus dans la population est un paramètre de notre algorithme. Pour initialiser la population, nous allons utiliser notre procédure gloutonne *CoverGreedly*. Afin d'essayer de trouver les meilleures partitions initiales, nous allons, pour chaque individu, exécuter N fois (N vaut 5) l'algorithme glouton et choisir la meilleure partition trouvée. Pour initialiser toute la population, nous allons ainsi exécuter N x $\mid \mathcal{P} \mid$ fois l'algorithme glouton.

3.4.2 Sélection des parents

Le rôle de la sélection consiste à choisir deux individus de la population pour devenir parents. Dans MA-GP, on procède de façon uniforme (aléatoire) de sorte que tous les individus ont la même chance d'être sélectionnés.

3.4.3 Opérateur de recombinaison

Dans un algorithme mémétique, l'opérateur de recombinaison cherche à combiner les caractéristiques des individus parents pour créer des individus enfants avec de nouvelles potentialités dans la génération future. Dans cette section, nous allons expliquer l'idée à l'origine de la recombinaison de MA-GP. Ensuite, nous allons présenter une façon simplifiée de l'implémenter.

L'idée principale de la recombinaison de MA-GP consiste à déterminer des ensembles de sommets communs à deux classes de deux parents différents. Ces ensembles de sommets représentent une caractéristique commune entre les deux parents

qu'on va transmettre à l'enfant. Le nombre de ces ensembles est égale k (le nombre de classes dans une partition). Comme critère d'appariement des classes, nous allons maximiser le nombre total de sommets dans les différentes intersections. Le fait de trouver k intersections qui maximisent le nombre de sommets revient à résoudre le problème du couplage parfait de poids maximum dans un graphe biparti complet $G_a(V_a, E_a)$. Les sommets de ce graphe sont répartis en deux ensembles disjoints V_1 et V_2 de k sommets chacun, tel que les sommets de V_1 sont reliés à tous les sommets de V_2. Les sommets de V_1 représentent les classes de P' et ceux de V_2 représentent les classes de P''. Une arête $(v_1, v_2) \in E_a$ représente l'intersection de deux classes (les classes représentées par v_1 et v_2) et le poids de cette arête est égale au nombre de sommets dans l'intersection. Résoudre le problème du couplage parfait de poids maximum dans G_a revient à choisir k arêtes non adjacentes dont la somme des poids est maximale. Les arêtes de ce couplage représentent les intersections entre les classes qui regroupent le plus grand nombre de sommets.

La figure 7 illustre les étapes de l'appariement des classes d'un exemple de deux partitions à 3 classes. Dans cet exemple, on a deux partitions d'un graphe de 12 sommets : $A = (A_1, A_2, A_3)$ et $B = (B_1, B_2, B_3)$. La figure 7(a) présente les 9 intersections entre les classes des deux partitions (deux de ces intersections sont vides). La première étape de l'appariement consiste à construire le graphe biparti présenté dans la figure 7(b). Les ensembles de sommets $V_1 = \{A_1, A_2, A_3\}$ et $V_2 = \{B_1, B_2, B_3\}$ de ce graphe représentent respectivement les classes des partitions A et B. Les sommets de V_1 sont tous liés aux sommets de V_2. Les arêtes du graphe biparti représentent les intersections des classes des deux partitions. L'arête (A_1, B_1) (respectivement (A_1, B_2), (A_1, B_3) ...) représente ainsi l'intersection des classes A_1 et B_1 (respectivement A_1 et B_2; A_1 et B_3); ...). De plus, le poids d'une arête est égal au nombre de sommets dans l'intersection. De ce fait, le poids de l'arête (A_1, B_1) (respectivement (A_1, B_2), (A_1, B_3) ...) est égal ainsi à 1 (respectivement 2,1, ...). La deuxième étape de l'appariement consiste à choisir 3 arêtes non adjacentes de poids maximum. La figure 7(c) présente les 3 arêtes de poids total maximum égal à 7. Pour construire une nouvelle partition, on forme avec les sommets

de chaque intersection choisie (représenté par une arête dans figure 7(c)) une nouvelle classe (figure 7(d)).

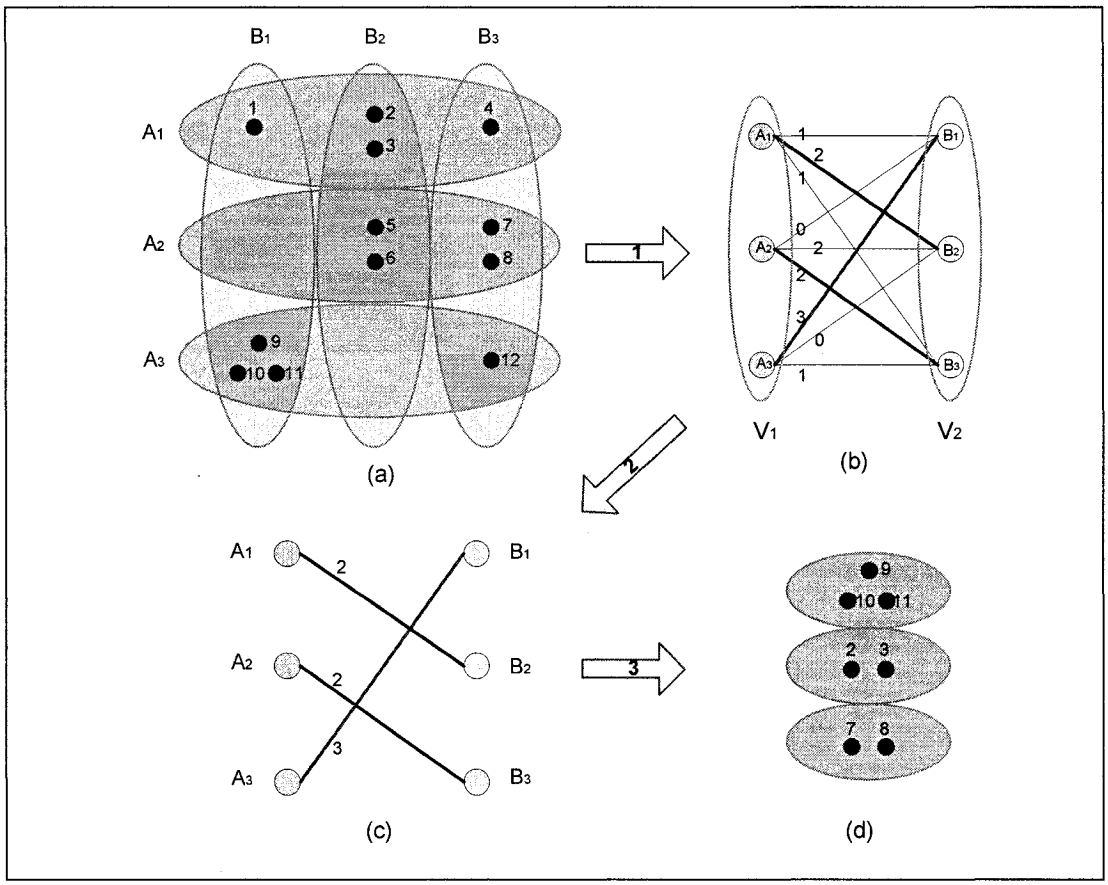

Figure 7 : Exemple illustratif de l'appariment: (a) Les intersections entre deux partitions à trois classes, (b) Le graphe biparti associé aux deux partitions, (c) Le couplage parfait de poids maximum dans le graphe biparti, (d) La nouvelle partition générée.

Dans *MA-GP*, l'idée à la base de la recombinaison repose sur le principe de l'appariement de classes présentées ci-dessous. Cependant, au lieu de déterminer le couplage parfait à poids maximum, nous allons nous contenter d'une procédure gloutonne plus simple. Cette procédure est décrite formellement dans l'algorithme 18.

L'opérateur de recombinaison de MA-GP prend en entrée deux individus parents $P' = (P_1', \ldots, P_k')$ et $P'' = (P_1'', \ldots, P_k'')$ et retourne en sortie le nouvel individu généré P. Pour k itérations, l'opérateur répète les étapes suivantes : Il choisit deux classes P_q' et P_r'' qui ont la plus grande intersection. Dans le cas où l'intersection est non

vide, il forme une nouvelle classe P_i avec les sommets de l'intersection. Dans le cas contraire (l'intersection est vide), il choisit aléatoirement l'une des deux classes P_q' ou P_r'' et forme avec ses sommets la classe P_i. L'opérateur supprime ensuite les classes P_q' et P_r'' et passe à l'itération suivante. Lorsque les k itérations auront été terminées, la solution résultante est une partition partielle de k classes qui couvre certains sommets. À l'étape 2 de la recombinaison, on complète la partition de sorte qu'elle couvre tous les sommets du graphe et ensuite on la rend légale de sorte que les classes ne dépassent pas la taille maximale permise. Pour compléter la solution, on ajoute aux classes les sommets qui ne font pas parties de la partition. Pour la légaliser, on équilibre les tailles des classes. Les deux procédures utilisées dans cette dernière étape sont respectivement les procédures *ExtendPartition* et *LegalizePartition* de notre algorithme glouton *CoverGreedly*.

- Entrée : Deux partitions $P' = (P_1', \ldots, P_k')$ et $P'' = (P_1'', \ldots, P_k'')$.
- Sortie : La partition P.

$P := \{\}$.

1. Construction d'un k-partitionnement partiel P :

Pour $i = 1 \ldots k$, faire :

a. Sélectionner deux classes $P_q' \in P'$ et $P_r'' \in P''$ de sorte que $|P_q' \cap P_r''|$ soit maximale.

b. Si $|P_q' \cap P_r''| > 0$, alors : //Intersection non vide
- $P_i := P_q' \cap P_r''$.

c. Sinon
- Affecter aléatoirement à P_i la classe P_q' ou P_r''.

d. Supprimer P_q' de P' et P_r'' de P''.

2. Étendre la partition P sur tous les sommets du graphe.

Algorithme 18 : L'opérateur de recombinaison de *MA-GP*.

Pour mieux illustrer le fonctionnement de notre recombinaison, nous allons l'appliquer sur l'exemple des partitions de la figure 7 : A = (A_1, A_2, A_3) = ({1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10 11, 12}) et B = (B_1, B_2, B_3) = ({1, 9, 10, 11}, {2, 3, 5, 6}, {4, 7, 8, 12}). On observe que la plus grande intersection est formée des classes A_3 et B_1. Avec les sommets 9, 10, 11 de cette intersection, on forme la classe C_1 et on supprime ensuite les classes A_3 et B_1. Les partitions A et B deviennent respectivement égales à ({1, 2, 3, 4}, {5, 6, 7, 8}) et ({2, 3, 5, 6}, {4, 7, 8, 12}). On observe trois intersections de taille maximale égale à 2. On choisit aléatoirement l'intersection des classes A_1 et B_2 et on forme avec les sommets 2 et 3 la classe C_2. Une fois les classes A_1 et B_2 supprimé, les partitions A et B devinent respectivement égale ({5, 6, 7, 8}) et ({4, 7, 8, 12}). De l'intersection de ces deux classes, on obtient la classe C_3 formée des sommets 7 et 8. La partition partielle C résultante est ainsi égale à ({9, 10 11}, {2, 3}, (7, 8)}).

3.4.4 Mise à jour de la population

Une fois la partition P rendue légale, notre procédure de recherche locale *TabuGP* est appliquée à l'étape (c) de l'algorithme 18. La solution résultante de cette procédure est utilisée pour mettre à jour la population. Elle remplace la solution de moindre qualité des deux solutions parentes P' et P'' .

3.4.5 Évaluation de la diversité

L'épuisement de la diversité de la population est un phénomène bien connu des algorithmes évolutionnaires. Il se produit lorsque les solutions de la population deviennent tous similaires. Une fois la diversité épuisée, il est inutile de poursuivre la recherche car l'algorithme aurait tendance à produire les mêmes partitions à chaque génération. Pour évaluer la diversité dans *MA-GP*, nous exploitons le même principe que celui de la recombinaison. Nous définissons ainsi une mesure de similarité entre deux individus $P' = (P_1', \ldots, P_k')$ et $P'' = (P_1'', \ldots, P_k'')$ comme le nombre total de sommets dans les k intersections maximales des classes P_q' et P_r''. Notons un tel nombre par $|P' \cap P''|$. Pour mesurer la similitude entre toutes les partitions de $\mathcal{P}$, nous évaluons

les intersections maximales entre les classes de chaque paire de partitions (il existe $|\mathcal{P}|^2$-$|\mathcal{P}|$ paires). On obtient ainsi le nombre total de sommets dans les intersections maximales des paires de partitions. Pour obtenir le nombre moyen de sommets par intersection de deux partitions, on divise le nombre total de sommets dans les intersections par $|\mathcal{P}|^2$-$|\mathcal{P}|$. La valeur obtenue représente une mesure de similitude entre les partitions de la population. Afin de normaliser cette mesure, on la divise par $|V|$. L'équation suivante donne ainsi une mesure de similitude (ou de diversité) de la population.

$$D(\mathcal{P}) = 1 - \frac{\sum_{P' \in \mathcal{P}, P'' \in \mathcal{P}, P' \neq P''} |P' \cap P''|}{(|\mathcal{P}|^2 - |\mathcal{P}|)\,|V|}$$

Équation 1: Évaluation de la diversité

Il faut noter que $D(\mathcal{P}) = 0$, lorsque la population est formée de $|\mathcal{P}|$ individus identiques, alors que $D(\mathcal{P}) = 1$, lorsque tous les individus de la population sont différents. De ce fait, nous pouvons fixer une petite valeur ε $(0 < \varepsilon << 1)$ de sorte que lorsque $D(\mathcal{P}) \leq \varepsilon$ nous arrêtons l'exécution de l'algorithme.

Chapitre 4. RÉSULTATS EXPÉRIMENTAUX

Dans le chapitre précédent, nous avons proposé trois algorithmes : L'algorithme Tabu-GP qui utilise une liste taboue variable, l'algorithme Tabu-GPG qui utilise deux phases de recherche avec deux fonctions d'évaluations différente et l'algorithme MA-GP qui utilise un opérateur de recombinaison.

Dans ce chapitre, nous présentons les résultats expérimentaux de nos algorithmes et nous effectuons des comparaisons avec d'autres algorithmes de la littérature. Les graphes que nous utilisons dans nos expériences sont issus de l'archive de partitionnement de graphe (Walshaw, 2006). Nous sommes intéressés par ces graphes parce qu'ils ont été largement étudiés dans la littérature et constituent une bonne référence pour les comparaisons.

L'archive de partitionnement de graphe (Walshaw, 2006) rassemble aussi les meilleurs résultats trouvés sur les graphes de références avec différentes valeurs de déséquilibre et différents nombre de classes. Ces résultats sont obtenus par une vingtaine d'algorithmes tels que P-Metis (Karypis et al., 1999), k-Metis (Karypis et al., 1998), JOSTLE Evolutionary (Soper et al., 2004), Iterated JOSTLE (Walshaw, 2004), etc. Cependant, l'archive de partitionnement de graphe ne tient pas compte des résultats de l'algorithme PMSATS (Banos et al., 2004) qui a permis d'améliorer beaucoup de ces résultats. Dans nos comparaisons, nous allons nous comparer aux meilleurs résultats connus sur les graphes tests. Nous allons ainsi nous comparer aux meilleurs résultats entre les résultats de l'archive et de l'algorithme PMSATS. Dans ce chapitre, nous appelons ces résultats les *"records"*.

Les algorithmes auxquels nous devons nous comparer, au point de vu temps d'exécution, sont les algorithmes PMSATS (Banos et al., 2004) et JOSTLE Evolutionary (Soper et al., 2004). Malheureusement, on ne dispose pas d'informations précises sur les temps d'exécutions de ces algorithmes pour effectuer des comparaisons détaillées. Dans l'article de PMSATS (Banos et al., 2004), les auteurs indiquent

seulement que, selon le graphe, le temps d'exécution varie de quelques secondes à plusieurs heures. De même, les auteurs de JOSTLE Evolutionary (Soper et al., 2004) mentionnent seulement que, pour les plus grands graphes, le temps d'exécution peut prendre plusieurs jours. De ce fait, dans les comparaisons, nous allons principalement nous baser sur les meilleurs résultats obtenus (records) plutôt que sur les temps d'exécutions.

Tous les résultats que nous présentons dans ce chapitre sont obtenus sur un processeur MD Opteron 2.0 Ghz avec 5G octets de mémoire.

4.1 Jeux de données utilisés pour nos tests

Les graphes qu'on utilise dans nos tests proviennent de l'archive de partitionnement de graphe (Walshaw, 2006). Pour chacun de ces graphes, le tableau 1 ci-dessous indique : le nom du graphe, le nombre de sommets (|V|), le nombre d'arêtes (|E|), la connectivité (degré) minimum des sommets (min), la connectivité maximum (max), connectivité moyenne (moyenne) et la connexité du graphe.

Tableau 1: Caractéristiques des graphes tests.

			Connectivité			
Graphe	**|V|**	**|E|**	**min**	**max**	**moyenne**	**connexe**
add20	2395	7462	1	123	6,231	-
data	2851	15093	3	17	10,588	-
3elt	4720	13722	3	9	5,814	-
uk	4824	6837	1	3	2,835	-
add32	4960	9462	1	31	3,815	-
bcsstk33	8738	291583	19	140	66,739	-
whitaker3	9800	28989	3	8	5,916	-
crack	10240	30380	3	9	5,934	-
wing_nodal	10937	75488	5	28	13,804	-
fe_4elt2	11143	32818	3	12	5,89	-
vibrobox	12328	165250	8	120	26,809	-
bcsstk29	13992	302748	4	70	43,274	non
4elt	15606	45878	3	10	5,88	-
fe_sphere	16386	49152	4	6	5,999	-
cti	16840	48232	3	6	5,728	-
memplus	17758	54196	1	573	6,104	-
cs4	22499	43858	2	4	3,899	-
bcsstk30	28924	1007284	3	218	69,65	-
bcsstk31	35588	572914	1	188	32,197	non
fe_pwt	36519	144794	0	15	7,93	non
bcsstk32	44609	985046	1	215	44,164	-
fe_body	45087	163734	0	28	7,263	non
t60k	60005	89440	2	3	2,981	-
wing	62032	121544	2	4	3,919	-
brack2	62631	366559	3	32	11,705	-
finan512	74752	261120	2	54	6,986	-
fe_tooth	78136	452591	3	39	11,585	-
fe_rotor	99617	662431	5	125	13,3	-
598a	110971	741934	5	26	13,372	-
fe_ocean	143437	409593	1	6	5,711	-
144	144649	1074393	4	26	14,855	-
wave	156317	1059331	3	44	13,554	-
m14b	214765	1679018	4	40	15,636	-
auto	448695	3314611	4	37	14,775	-

4.2 Construction de la configuration initiale de Tabu-GP

Un algorithme de recherche locale (par exemple, un algorithme tabou) commence sa recherche à partir d'une configuration. Pour construire une configuration initiale, plusieurs techniques sont traditionnellement envisagées. L'une d'entre elles consiste à construire la configuration de manière totalement aléatoire. Une autre approche consiste à utiliser un algorithme glouton. Le but des expériences relatées dans cette section est d'évaluer le mérite de ces deux options.

Un algorithme glouton construit en général une solution sensiblement meilleure qu'une solution aléatoire. L'algorithme glouton risque d'être beaucoup plus lent, mais ceci est sans réelle importance car le temps consommé par l'algorithme glouton pour construire une solution est néanmoins très court, en comparaison du temps total accordé à l'algorithme. A priori, il se pourrait que l'utilisation d'un algorithme glouton aide la recherche et permette de trouver de meilleures solutions ou d'accélérer la découverte de bonnes solutions : ceci justifierait de faire appel à l'algorithme glouton. Mais c'est n'est pas nécessairement le cas. Inversement, il se pourrait aussi que l'algorithme tabou initialisé avec une solution aléatoire puisse, en un temps très court, atteindre une solution aussi bonne que celle qu'on aurait obtenue avec l'algorithme glouton – et, ainsi, «rattraper» très rapidement l'algorithme glouton. Dans ce cas, l'utilisation d'un algorithme glouton représenterait une complication inutile.

Pour comparer les deux options (construction aléatoire et construction gloutonne), nous avons effectué des tests sur quelques graphes. Par exemple, la figure 8 illustre les résultats obtenus sur les graphes *t60k* et *598a*. L'évolution de la meilleure valeur de la fonction de coût f* de Tabu-GP est présentée en pointillé (TS+random) lorsque l'algorithme est initialisé avec une partition aléatoire et en trait plein (TS+Greedy) lorsqu'il est initialisé avec une partition gloutonne. Pour les deux graphes, on observe que les courbes en pointillés commencent avec des valeurs beaucoup plus élevées que les courbes en trait plein. Ceci n'est pas surprenant puisque on s'attendait à ce que le nombre d'arêtes coupées dans les partitions aléatoires soit plus élevé que celui

des partitions gloutonnes. En effet, la méthode gloutonne ajoute à une classe les sommets adjacents aux sommets déjà dans la classe. Plusieurs arêtes vont ainsi se trouver à l'intérieur d'une même classe, ce qui en conséquence limite le nombre d'arêtes interclasse. Contrairement à cela, la méthode aléatoire ajoute les sommets aléatoirement aux classes.

On peut aussi observer de ces mêmes courbes que selon, le graphe testé et la partition initiale utilisée, la recherche peut avoir différents comportements. Pour un nombre de classes k=16 et un déséquilibre Imb=5%, la recherche sur la partition aléatoire du graphe *t60k* converge vers une solution de moindre qualité que la recherche sur la partition gloutonne du même graphe. Cependant pour le graphe *598a*, la recherche converge vers les mêmes solutions, indépendamment de la partition initiale utilisée. On a observé des constations analogue sur d'autre graphe. De ce fait, nous pouvons conclure qu'une partition gloutonne permet, dans certains cas, d'améliorer la partition trouvée par la recherche locale (Ceci est en comparaison avec une partition aléatoire).

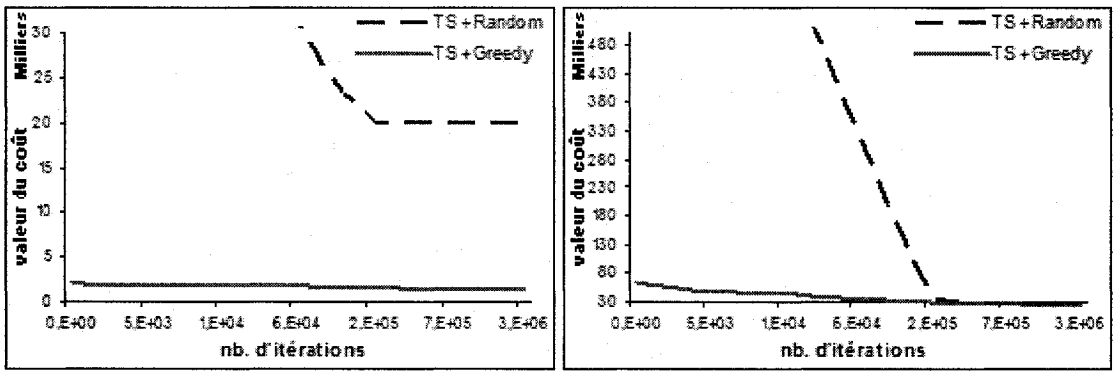

Figure 8 : Les profiles d'éxécututions de Tabu-GP initialié avec différentes partitions initiales (à gauche le graphe t60k et à droite 598a).

Sur le graphe 598a la recherche converge vers les mêmes partitions, indépendamment de la partition initiale utilisée (courbes à droite), on remarque que, TS+Random rattrape TS+Greedy seulement au bout de 25 millions.

En analysant le temps d'exécution de Tabu-GP, on s'aperçoit que pour 3 millions d'itérations, l'algorithme termine le partitionnent du graphe t60k en 16 classes au bout de 127 secs lorsqu'il est initialisé avec une partition gloutonne. Cependant, lorsqu'il est initialisé avec une partition aléatoire, il se termine au bout de 988 secs. Dans le cas du

graphe 598a, le temps d'exécution de l'algorithme avec une partition initiale gloutonne est de 555 secs. Alors qu'avec une partition initiale aléatoire, il est de 2331 secs. La différence entre les temps d'exécutions des deux instances de Tabu-GP, est due au nombre de mouvements testés à chaque itération. En effet, dans Tabu-GP, le nombre de mouvements possibles à une itération augmente avec les arêtes coupées. Étant donné que le nombre d'arêtes coupées dans la partition gloutonne est inférieur à celui de la partition aléatoire, la recherche locale passe ainsi moins de temps à évaluer les mouvements dans la partition gloutonne qu'à évaluer les mouvements de la partition aléatoire.

En se basant sur les observations effectuées, nous pouvons conclure que les partitions gloutonnes de CoverGreedily permettent de mieux préserver les performances de la recherche locale que les partitions aléatoires. De ce fait, dans les expériences suivantes, nous allons toujours utiliser des partitions gloutonnes.

4.3 Effet de la variation de la liste taboue

Pour fixer la longueur de la liste taboue, une option habituelle consiste à utiliser une liste taboue de longueur constante *It_tabu* (où *It_tabu* un paramètre). Une méthode un peu plus compliquée consiste à choisir le nombre d'itérations pendant lequel un attribut reste tabou en effectuant un choix aléatoire dans un intervalle $[T_{max}, T_{min}]$ (où T_{max} et T_{min} sont des paramètres). C'est ce genre de méthode que nous avions choisie initialement pour notre algorithme Tabu-GP. Cependant, nous nous sommes rendu compte qu'une liste constante (ou qui varie dans un intervalle) présentait des inconvénients majeurs. C'est pourquoi nous utilisons dans Tabu-GP une liste dont la longueur varie systématiquement selon une forme analogue à une fractale, (voir section 3.2.4). Dans cette section, nous présentons des expériences qui comparent le comportement et les résultats obtenus avec Tabu-GP selon qu'on utilise une liste constante ou une liste fractale.

Les inconvénients d'une liste constante sont illustrés par l'expérience suivante : Pour une valeur *It_tabu* fixée, nous vidons, à un moment de l'exécution, la liste taboue de façon brusque et permanente. La figure 9 illustre le résultat obtenu sur le graphe

memplus (*It_tabu*=8400) lorsqu'on vide la liste à l'itération 100000. Pour fixer *It_tabu*, nous avons testé quelques valeurs taboues avec un nombre réduit d'itérations (k=16 et imb=5%). Nous avons ensuite fixé *It_tabu* à la valeur qui permet d'obtenir les meilleurs résultats par rapport aux autres valeurs testées.

On remarque à partir de la figure 9, que juste avant l'itération 100000, la valeur du coût diminue lentement, alors que juste après cette même itération, la valeur du coût diminue de façon brusque. Ainsi, le fait de vider la liste a permis, juste après 20000 itérations (à l'itération 120000), de diminuer de 1000 la valeur de coût. Pour, d'autres graphes testés, bcsstk31 (*It_tabu*=6000) et vibrobox (*It_tabu*=7800) les diminutions enregistrées après 20000 itérations du vidage de la liste, sont respectivement 1700 et 3000. Ce phénomène s'explique par le fait qu'avant l'itération 100000 la liste taboue interdisait en permanence de nombreux mouvements qui permettraient d'améliorer le coût. En vidant complètement la liste, les mouvements qui étaient auparavant interdits sont devenus permis ce qui permet de réduire la fonction de coût au cours des itérations suivantes.

Il faut remarquer que les valeurs fixées de la liste taboue sont très grandes (exemple, *It_tabu*= 8400 pour memplus). Un grand nombre de mouvements se trouvent ainsi en permanence interdits. Cependant, nous avons observé que le fait d'utiliser des listes plus petites ne résoudrait pas le problème car la recherche risquerait dans ce cas de se bloquer très rapidement dans les optima locaux (ou des bassins d'attraction pu profonds).

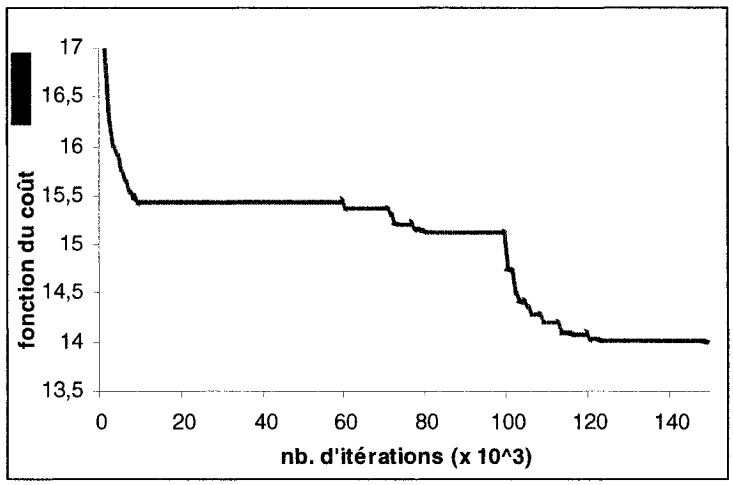

Figure 9 : Effet du vidage de la liste taboue (graphe memplus).

L'expérience que nous venons d'effectuer suggère de vider la liste constante (const+vidage) plusieurs fois au cours de la recherche. Une façon simple de procéder consiste à vider instantanément la liste à chaque *It_per* itérations. Une autre méthode de variation plausible serait de varier de façon systématique la longueur de la liste (fractale).

Afin de fixer la forme de la liste dans Tabu-GP, nous allons comparer les résultats obtenus avec les listes suivantes : constante, const+vidage et fractale. Pour ce faire, nous avons commencé par fixer la longueur de la liste constante pour quelques graphes (avec k=16 et imb=5%). Nous avons ensuite fixé la longueur de la liste const+vidage à *It_tabu* et la durée d'une période *It_per* à 4x *It_tabu*. Dans le cas de la fractale, nous avons fixé la valeur maximale de la liste taboue T_{max} (voir section 3.2.4). Par exemple, la figure 10 illustre la moyenne (10 exécutions) des résultats de Tabu-GP sur les graphes fe_pwt (droite) et 4elt (gauche) avec un nombre d'itérations de 5 millions, k=16 et *imb*=5%. D'après cette figure, nous remarquons que la liste const+vidage améliore considérablement le résultat de la liste constante. Mais, en faisant varier la longueur de la liste de manière plus systématique (fractale), le résultat est encore meilleur. De ce fait, dans la suite des expériences, nous utiliserons la forme fractale pour varier la liste de Tabu-GP.

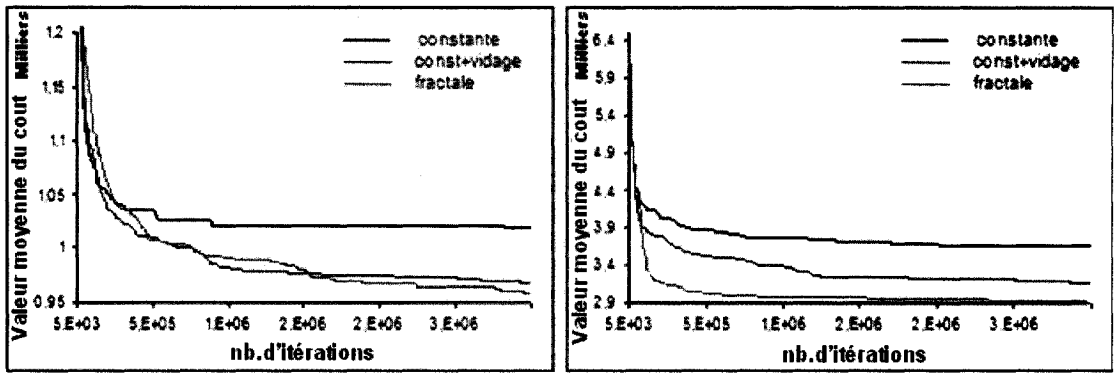

Figure 10 : Les traces d'exécutions de l'algorithme pour différentes variations de la liste taboue (Le graphe 4elt à droite et graphe fe_pwt à gauche).

4.4 Réglage du paramètre T_{max} dans Tabu-GP

L'algorithme TabuGP comporte deux paramètres : Le nombre d'itérations de recherche locale *It_max* et la longueur maximale T_{max} de la liste taboue fractale.

Pour fixer la valeur maximale T_{max} de la liste fractale, nous avons effectué des tests en procédant exemplaire par exemplaire (un exemplaire est constitué d'un graphe, d'une valeur *k* et d'une valeur *Imb*). Le principe de ces tests consiste à lancer des séries d'exécutions avec différentes valeurs du paramètre afin de tenter d'identifier une valeur aussi bonne que possible pour celui-ci. Afin de réduire la durée de ces tests préliminaires, nous choisissons d'effectuer un nombre réduit d'exécutions par série et un nombre réduit d'itérations de l'algorithme.

La figure 11 illustre les résultats de Tabu-GP, dans le cas de l'exemplaire *uk*, *k*=16 et *Imb*=5%, avec 10 valeurs T_{max} entre 25 et 800. Pour chaque valeur de T_{max}, nous avons effectué 10 exécutions de 5 millions d'itérations et nous avons représenté les résultats sous forme d'une boîte à moustaches (Box & Whiskers Plot). Les boîtes à moustaches rassemblent les informations suivantes:

- La valeur du 1er quartile Q_1 (25% des résultats), correspondant au trait inférieur de la boîte.
- La valeur du 2ème quartile Q_2 (50% des résultats), représentée par un trait horizontal à l'intérieur de la boîte.

- La valeur du 3ème quartile Q_3 (75% des résultats), correspondant au trait supérieur de la boîte.
- Les deux moustaches inférieure et supérieure qui délimitent respectivement la valeur minimum dans les données qui est supérieure à Q_1-1,5(Q_3- Q_1) et la valeur maximum qui est inferieure à Q_1+1,5(Q_3- Q_1).
- Les valeurs extrêmes, exceptionnelles, situées au-delà des deux moustaches. Ces valeurs sont représentées par des marques (+).

Sur la figure, on observe que les valeurs de f* (par exemple, la médiane ou le minimum) présentent une forme en U : La valeur de f* décroit au début jusqu'à attendre un minimum, ensuite elles augmentent au fur et à mesure que les valeurs T_{max} augmentent. Ceci n'est guère étonnant car on peut supposer qu'il existe une valeur idéale pour le paramètre (celle qui correspond à la partie centrale du U) et que la performance se dégrade si on s'éloigne de cette valeur idéale. Dans l'exemplaire de la figure 11, la valeur idéale devrait être dans l'intervalle [150 ... 400]. En effet, on observe que la performance de Tabu-GP est meilleure avec les valeurs T_{max} de cet intervalle qu'avec les autres valeurs testées. Toutefois, il n'est pas évident de déterminer la valeur idéale parce que la performance de Tabu-GP est assez stable pour T_{max} =[150 ... 400]. Toutes les valeurs de cet intervalle semblent donc adéquates comme valeur du paramètre. Dans le cas de cet exemplaire (Uk, k=16, Imb=5%), nous avons fixé la valeur de T_{max} à 300.

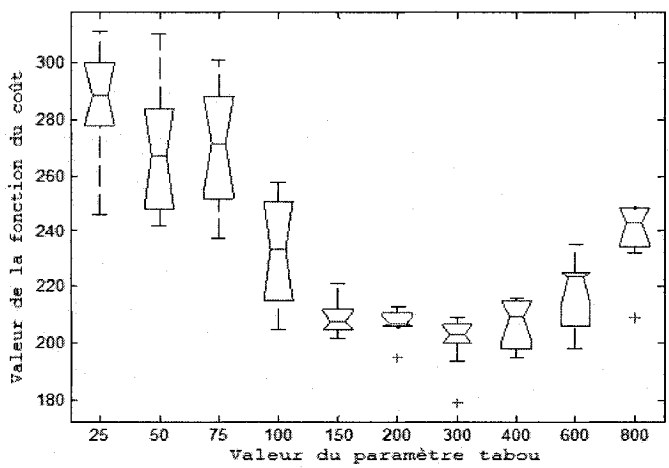

Figure 11 : Distribution des résultats de TabuGP sur le graphe uk.

Nous avons effectué le même genre de tests pour tous les exemplaires testés. Les valeurs trouvées pour la liste taboue sont indiquées dans les tableaux présentés dans les sections suivantes.

4.5 Analyse de la diversité dans MA-GP

Dans le cas des algorithmes mémétiques, il est connu que la diversité de la population a une grande influence sur la performance. Un épuisement rapide de la diversité de la population conduit la recherche à une convergence prématurée (une stagnation de la recherche). Dans cette section, nous allons étudier l'évolution de la diversité au cours de la recherche en variant le nombre d'itérations de la recherche locale L. Pour évaluer la diversité de la population, nous allons utiliser la mesure de la diversité présentée dans la section 3.4.5 du chapitre 3.

Par exemple, la figure 12 présente l'évolution de la diversité de population pour le graphe bcsstk32 (k=16, Imb=5%) pour L=5000, L=25000 et L=125000. Le nombre total d'itérations de recherche locale It_max est 5000000. La première chose qu'on observe sur la figure est que la diversité s'épuise plus rapidement pour les plus petites valeur de L. Pour L = 5000, la diversité s'est épuisée au bout de 1000000 itérations. MA-GP a effectué ainsi 200 générations avant l'épuisement de la diversité. Pour L=25000 la diversité s'épuise au bout de 5000000 itérations. MA-GP a donc aussi effectué 200 générations avant l'épuisement de la diversité.

Nous avons effectué la même expérience sur d'autres exemplaires et nous nous somme aperçus que la diversité s'épuise toujours au bout d'environ 200 générations. Ainsi, dans MA-GP, il semble que la diversité s'épuise au bout 200 générations, indépendamment de l'exemplaire er de la valeur de L.

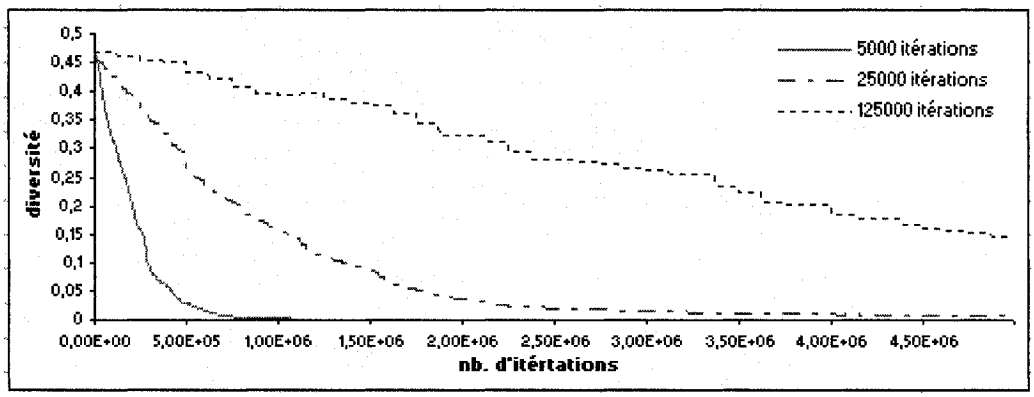

Figure 12 : Courbes d'évolution de la diversité (bcsstk32, k=16, Imb=5%).

La figure 13 illustre l'évolution de la meilleure valeur de la fonction du coût (f*) obtenue sur le graphe bcsstk32 (k=16, Imb=5%) pour L=5000, L=25000 et L=125000. On observe que pour L = 5000, MA-GP arrête d'améliorer f* au bout de 100000 itérations, ce qui coïncide exactement au moment où la diversité s'est épuisée. Lorsqu'on observe l'évolution de f* pour L=25000, on s'aperçoit aussi que l'amélioration s'arrête au même moment où la diversité s'est épuisée (c'est-à-dire à 500000 itérations). En effectuant cette même expérience avec d'autres graphes, nous avons observé le même cas de figure. L'amélioration de f* s'arrête lorsque la diversité s'épuise. Il vaut mieux arrêter la recherche lorsque la diversité de la population est épuisée.

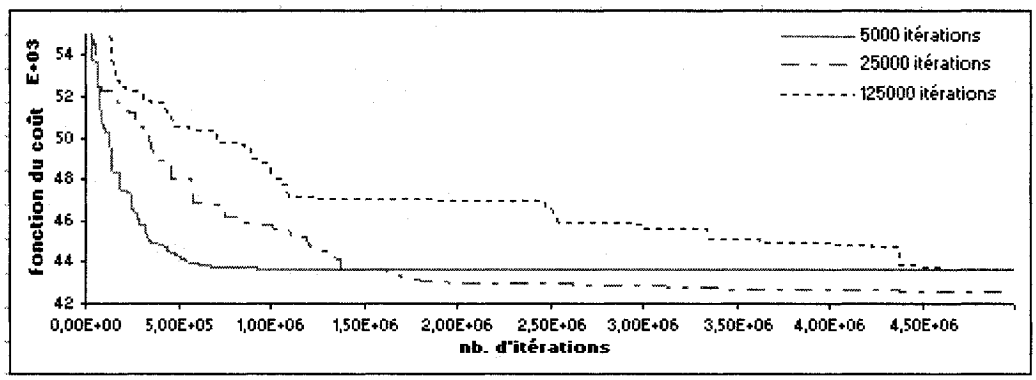

Figure 13 : Courbes d'évolution de la fonction de coût (bcsstk32, k=16, Imb=5%).

On remarque aussi que plus L est grande, plus f* décroit lentement. Cependant, une grande valeur de L vaut mieux à long terme qu'une valeur plus petite puisque pour les petites valeurs de L la diversité s'épuise plus rapidement. Ainsi, pour un même

nombre de générations ($It\_\max/L$), il vaut mieux accorder les plus grandes valeurs possibles à L. Toutefois, en faisant de la sorte, on rend l'algorithme de plus en plus long.

Dans MA-GP, le nombre total d'itérations *It_max* est fixé à 30 millions d'itérations. Étant donné que l'amélioration de f* s'arrête lorsque la diversité s'est épuisée, il vaut mieux ainsi que le nombre d'itérations *It_max* se termine au moment où la diversité s'épuise. De ce fait, on fixe L à 150000 itérations (30000000 itérations/200 générations).

4.6 Résultats de Tabu-GP

Dans les sections suivantes, nous allons présenter des tests approfondis réalisés avec nos trois algorithmes (Tabu-GP, Tabu-GPG et MA-GP). Comme le jeu de données de l'archive comprend un nombre d'exemplaires beaucoup trop grand (4×6×34=816 exemplaires en tout), nous avons sélectionné un jeu de tests réduit constitué de 10 exemplaires seulement. Ceux-ci correspondent à 10 graphes de taille et de densité variées (voir tableau 2) avec k=16 et *imb*=5%.

Le tableau 2 présente les résultats de Tabu-GP obtenus sur le jeu de tests réduit. Pour obtenir ces résultats, nous avons effectué pour chaque graphe une série de 10 exécutions de 30 millions d'itérations chacune. Le paramètre tabou T_{max} a été fixé selon la méthode décrite dans la section 4.4. La valeur fixée du paramètre est présentée, pour chaque graphe, dans la colonne 2 (tableau 2). Les résultats f* (meilleure valeur de la fonction du coût) de 10 exécutions sont présentés dans les colonnes 3 à 6 sous la forme suivante : La valeur minimale f^*_{min}, la valeur maximale f^*_{max}, la moyenne f^*_{avg} et l'écart type f^*_{std}. La colonne CPU indique le temps d'exécution moyen (en seconde) de Tabu-GP. La colonne Record (f^*_{min}) donne les meilleurs résultats connus des exemplaires testés et la colonne Record (gain(%)) donne les gains de Tabu-GP par rapport à ces meilleurs résultats. Le gain est calculé selon la formule suivante [Tabu-GP(f^*_{min})/ Record (f^*_{min})] – 1. Un gain négatif indique une amélioration par rapport au record connu alors qu'un gain positif indique une dégradation.

Les quatre dernières lignes du tableau indiquent respectivement : la moyenne des gains par graphe, le nombre de records battus, le nombre des égalités avec le record et le nombre de records non battus.

Tableau 2 : Résultats de l'algorithme Tabu-GP (k=16 et Imb=5%) obtenus avec 10 exécutions de 30 millions d'itérations.

Graphe	T_{max}	f^*				CPU (sec)	Record	
		f^*_{min}	f^*_{max}	f^*_{avg}	f^*_{std}		f^*_{min}	gain(%)
Data	230	1074	1080	1077,1	1,70	218,95	1147	-6,36
Uk	300	179	203	194,4	7,50	391,27	154	16,23
Crack	750	1075	1088	1079,8	3,87	423,51	1082	-0,65
vibrobox	7800	31323	31412	31369,9	26,72	2003,09	31695	-1,17
4elt	1580	916	943	930,6	7,06	507,99	916	0,00
memplus	8400	13164	13705	13338,3	163,17	3770,51	13279	-0,87
Bcsstk31	6000	26749	33445	30920,1	2131,38	2250,81	24179	10,63
fe_rotor	16500	20597	21759	21114,9	363,96	4942,04	20674	-0,37
598a	8800	25378	25658	25525,6	78,75	4448,11	25775	-1,54
Wave	12700	42921	44275	43726,7	448,16	5813,50	43501	-1,33
Gain moyen(%)								1,46
Nombre de records battus								7
Nombre des égalités avec le record								1
Nombre de records non battus								2

On observe à partir du tableau que Tabu-GP améliore 7 records et obtient un résultat à égalité avec un record. La meilleure amélioration est obtenue avec le graphe data (-6,36%) alors que la plus mauvaise est obtenue avec le graphe uk (16.23%). En moyenne Tabu-GP n'a pas pu améliorer les records des 10 graphes testés : il dégrade en moyenne les records connus de 1,46%.

Le plus souvent Tabu-GP obtient des résultats un peu mieux que les meilleurs résultats connus (records). En effet, il améliore les résultats de 7 graphes de 0 à 1.33%. Pour le graphe *data*, Tabu-GP obtient un résultat sensiblement meilleur que le meilleur résultat connu (6,36 %). Cependant, pour les graphes uk et bcsstk31, il obtient des résultats sensiblement moins biens. Il dégrade les recordes connus respectivement de 16.23% et 10,36%.

4.7 Résultats de Tabu-GPG

Le tableau 3 présente les résultats de Tabu-GPG pour k=16 et imb=5%. Les jeux de données et le nombre d'itérations de recherche locale *It_max* sont les mêmes que ceux utilisés pour Tabu-GP (*It_max*= 30 millions).

Dans Tabu-GPG, il y a deux paramètres : La valeur taboue T_{max} et le nombre d'itérations L appliqués à chaque phase de recherche locale. Pour fixer T_{max}, nous avons utilisé une procédure analogue à celle utilisée pour Tabu-GP (section 4.4). Les valeurs fixées de T_{max} sont présentées dans la colonne 3 (tableau 3). De même, pour fixer L, nous avons utilisé la même procédure que celle de la section 4.4, mais au lieu de varier T_{max}, nous avons varié L. Les valeurs de L testés sont 186335, 370370, 731707 et 1428570 qui correspondent respectivement à 161, 81, 41 et 21 phases de recherche locale. Les valeurs de L fixées pour chaque graphe sont présenté dans la colonne 2 (tableau 3).

Comme est le cas dans Tabu-GP, les colonnes f^*_{min}, f^*_{max}, f^*_{avg} et f^*_{std} indiquent respectivement la valeur minimale, la valeur maximale, la valeur moyenne et l'écart type des résultats de 10 exécutions de Tabu-GPG.

En plus des records, nous avons reporté aussi dans le tableau 3 les résultats de Tabu-GP et nous avons calculé les gains de Tabu-GPG par rapport à ces derniers. Le calcul de gain par rapport à Tabu-GP est fait selon la formule [Tabu-GPG(f^*_{min})/Tabu-GP(f^*_{min})] – 1 et par rapport au record selon la formule [Tabu-GPG(f^*_{min})/ Records(f^*_{min})] – 1. De plus, dans la colonne Tabu-GP(CPU), nous avons calculé le surcoût du temps d'exécution de Tabu-GPG par rapport à Tabu-GP selon la formule [Tabu-GPG(CPU)/Tabu-GP(CPU)] – 1.

Les quatre dernières lignes du tableau indiquent respectivement : la moyenne des gains par graphe, le nombre de records battus, le nombre des égalités avec le record et le nombre de records non battus.

Tableau 3 : Résultats de l'algorithme Tabu-GPG (k=16 et Imb=5%) obtenus avec 10 exécutions de 30 millions d'itérations.

Graphe	L	T_{max}	f^*				CPU (sec)	Tabu-GP		Records	
			f^*_{min}	f^*_{max}	f^*_{avg}	f^*_{std}		gain (%)	CPU	f^*_{min}	gain(%)
Data	370370	85	1074	1101	1078	7,8	339,9	0,00	55,3	1147	-6,36
Uk	1428570	470	147	159	154,1	3,86	564	-17,9	44,2	154	-4,55
Crack	186335	250	1074	1083	1078,5	2,69	648,7	-0,1	53,2	1082	-0,74
Vibrobox	1428570	3900	31322	32585	31868,6	370,2	1820,1	-0,003	-9,13	31695	-1,18
4elt	186335	590	895	936	912,2	13,83	672,6	-2,29	32,4	916	-2,29
Memplus	370370	3150	13111	13406	13237,8	101,6	1194	-0,4	68,3	13279	-1,27
Bcsstk31	370370	3000	23280	25024	24234,5	504,1	3276,4	-12,8	45,6	24179	-3,72
fe_rotor	370370	6880	19823	20836	20425,9	299	4676,6	-3,76	-5,37	20674	-4,12
598a	186335	1460	25360	26611	25869,7	391	4658,6	-0,07	4,73	25775	-1,61
Wave	1428570	9520	42852	44127	43448,6	401,9	6022,1	-0,16	3,59	43501	-1,49
Gain moyen(%)								-3,76	15,6		-2,73
Nombre de records battus								9			10
Nombre des égalités avec le record								1			0
Nombre de records non battus								0			0

On observe à partir du tableau 3 que Tabu-GPG a battu tous les records connus des 10 graphes testés. La plus grande amélioration est effectuée sur le graphe data (-6,36%) alors que la plus faible est effectuée sur le graphe crack (-0,74%). Le gain moyen par graphe, de Tabu-GPG par rapport au record connus, est de -2,73%.

On observe aussi que Tabu-GPG a amélioré 9 des meilleurs résultats de Tabu-GP et obtenu un résultat égal. La plus grande amélioration est effectuée sur le graphe uk (-17,9%). Le gain moyen par graphe, de Tabu-GPG par apport à Tabu-GP, est de -3,76%.

Lorsqu'on observe le surcoût du temps d'exécution, on remarque qu'en moyenne Tabu-GPG n'a besoin que de 15,6% de temps de plus que Tabu-GP.

De ce fait, on peut conclure que le guidage de la recherche a permis d'améliorer les résultats obtenus par la recherche locale pure et a permis ainsi de battre tous les records des 10 graphes testés.

4.8 Résultats de MA-GP

Le tableau 4 présente les résultats de MA-GP pour k=16 et imb=5%. Les jeux de données et le nombre d'itérations de recherche locale *It_max* sont les mêmes que ceux utilisés pour Tabu-GP (*It_max*= 30 millions).

Dans MA-GP, il y a trois paramètres : la taille de la population $|\mathcal{P}|$ et le nombre d'itérations L de la recherche locale appliqué après le croisement et le paramètre tabou T_{max}. Nous avons fixé la taille de la population $|\mathcal{P}|$ à 10, comme c'est souvent le cas dans les algorithmes hybrides (Galinier & Hao, 1999). Pour le paramètre L, nous avons utilisé la valeur 150000 fixée dans la section 4.5.

Pour fixer T_{max}, nous avons utilisé la même procédure que pour Tabu-GPG : Nous avons testé quelques valeurs du paramètre (3 à 5), avec *It_max* =30 millions, ceci en utilisant la procédure présentée dans la section 4.4. Les valeurs fixées de T_{max} sont présentées dans la colonne 2 (tableau 4).

Les informations présentées dans le tableau 4 sont les mêmes que celles présentées dans le tableau 3 (mise à part la valeur de L que nous ne présentons pas dans le tableau 4 puisqu'elle est constante dans MA-GP).

Tableau 4 : Résultats de l'algorithme MA-GP (k=16 et Imb=5%) obtenus avec 10 exécutions de 30 millions d'itérations.

Graphe	T_{max}	f^*				CPU (sec)	Tabu-GP		Records	
		f^*_{min}	f^*_{max}	f^*_{avg}	f^*_{std}		Gain (%)	CPU	f^*_{min}	Gain(%)
Data	180	1072	1074	1072,9	0,94	263,4	-0,19	20,30	1147	-6,54
Uk	150	164	181	175,3	4,61	811,4	-8,38	107,3	154	6,49
Crack	570	1067	1096	1075,6	7,86	520,2	-0,74	22,83	1082	-1,39
vibrobox	2600	31236	31732	31412,8	139,7	1087,7	-0,28	-45,7	31695	-1,45
4elt	1185	892	916	901,3	7,07	555,7	-2,62	9,40	916	-2,62
memplus	3150	13329	13450	13410,9	40,78	1505,5	1,25	-60,0	13279	0,38
Bcsstk31	6000	22891	24852	23533,6	586,9	2343,5	-14,4	4,12	24179	-5,33
fe_rotor	3660	19887	20836	20343,4	267,0	3865,0	-3,45	-21,7	20674	-3,81
598a	3300	25374	25772	25494	134,4	5230,3	-0,02	17,58	25775	-1,56
Wave	4760	42229	43863	43153	494,5	5151,7	-1,61	-11,3	43501	-2,92
Gain moyen(%)							-3,05	4,27		-1,87
Nombre de records battus							9			8
Nombre des égalités avec le record							0			0
Nombre de records non battus							1			2

On observe à partir du tableau 4 que MA-GP a amélioré 8 records parmi 10. La plus grande amélioration est de -6,54% alors que la plus faible est de -1,39%. Dans le cas des graphes *uk* et *memplus*, MA-GP ne parvient pas à battre les records. Il dégrade

les records connus de 6,49% (*uk*) et de 0,38% (*memplus*). Le gain moyen par graphe de MA-GP par rapport aux records connus est de -1,87%. Cependant, lorsqu'on ne tient pas compte des deux graphes pour lesquels MA-GP dégrade le record (*uk*, *memplus*), le gain moyen par graphe devient égal à -3,2%.

Lorsqu'on compare les résultats de MA-GP à ceux de Tabu-GP, on s'aperçoit que MA-GP améliore 9 résultats sur 10. La meilleure amélioration est obtenue dans le cas du graphe bcsstk31 (14,4%). Cependant dans le cas du graphe *memplus*, MA-GP dégrade de 1,25%. Le gain moyen par graphe de MA-GP par rapport aux résultats de Tabu-GP est égal à -3,05%.

Au point de vue de temps d'exécution, on remarque qu'en moyenne MA-GP n'a besoin que de 4,27% de plus de temps que Tabu-GP.

Étant donné que MA-GP a permis d'améliorer 8 records de 10 graphes testés de -3,2% et a permis d'améliorer les résultats de Tabu-GP de -3,05%, nous pouvons conclure que la recombinaison a été capable de produire des partitions qui ont permis à la recherche d'être plus efficace.

Dans tous les tests expérimentaux que nous avons montré jusqu'à présent, nous nous sommes limités à k=16 classes et au déséquilibre de 5% (*Imb*=5%). Dans l'archive de partitionnement de graphes (Walshaw et al., 2006), les résultats sont présentés pour différentes valeurs de k (2, 4, 8, 16, 32, 64) et aussi pour différents déséquilibres (0%, 1%, 3%, 5%). Dans le tableau 5, nous présentons les résultats de MA-GP pour tous les graphes de l'archive et les différentes valeurs de k généralement utilisées. Cependant, compte tenu du nombre élevé des jeux de données, nous nous sommes limités à une seule valeur de déséquilibre (5%).

Dans le tableau 5, nous présentons pour chaque graphe et pour chaque valeur de k (2, 4, 8, 16, 32, 64) le meilleur résultat trouvé (f^*_{min}) par MA-GP sur un nombre total de 10 exécutions. Les colonnes intitulées *gain* donnent l'écart des meilleurs résultats de MA-GP par rapport aux records connus. Pour ce qui est des paramètres d'exécution, nous avons gardé les mêmes que ceux des expériences précédentes sur MA-GP (*It_max* = 30 millions, L=150 milles). Cependant, pour le paramètre tabou T_{max}, nous avons

utilisé des valeurs qui ont été fixées avant d'intégrer la liste taboue variable. Faute de temps, nous n'avons pas fixé les nouvelles valeurs pour la liste fractale (certains résultats du tableau 4 sont meilleurs que ceux du tableau 5).

Dans le tableau 5, nous avons utilisé différentes couleurs pour mieux distinguer les améliorations (gain négatif) des dégradations (gain positif). Un gain coloré en vert indique que MA-GP améliore le record connu de moins de 5%. Cependant, un gain coloré en rouge indique que MA-GP dégrade le record connu de moins de 5%. Un gain coloré en bleu indique une égalité avec le record. Dans le cas où l'amélioration par rapport au record est supérieure à 5%, nous avons ajouté au gain un fond vert. Cependant, dans le cas où la dégradation par rapport au record est supérieure à 5%, nous avons ajouté au gain un fond rouge.

On observe à partir du tableau que la couleur verte (les améliorations) est beaucoup plus dominante que les autres couleurs. Cependant, on observe que pour k=2, les couleurs bleu et rouge dominent plus que le vert. En effet, pour les 34 graphes testés avec k=2, on observe 17 cas d'égalité et 13 cas de dégradation par rapport au record. Ainsi, pour k=2, on conclut que MA-GP est moins performant que pour les autres valeurs de k.

On observe aussi que, pour les graphes uk et t60k, les résultats de MA-GP sont supérieurs de plus que 5% par rapport aux records connus, ceci est vrai pour la plupart des valeurs de *k*. De plus, on observe que, dans le cas des graphes cs4 et wing, MA-GP n'obtient pas des résultats uniformes. En effet, pour le graphe cs4, MA-GP n'améliore le résultat que pour k=32 et k=64 et, pour le graphe wing, MA-GP améliore pour k={8, 32, 64}. Lorsqu'on observe les caractéristiques de ces graphes (tableau 1), on s'aperçoit que ces graphes sont très peu denses (la densité moyenne varie de 2,835 à 3,919). De ceci, nous pouvons conclure que MA-GP performe moins dans le cas des graphes moins denses.

Pour les valeurs de $k \neq 2$ et excepté les graphes peu denses MA-GP arrive à battre la majorité des records. De plus, pour 28 exemplaires, il arrive à améliorer de plus de -5% (il atteint même -9,9%). En analysant globalement les résultats du tableau 5, nous

remarquons que MA-GP arrive dans 72% des cas à améliorer un record et dans 12% des cas à atteindre le record établi. Ainsi, MA-GP dégrade seulement 16% des records.

Tableau 5 : Résultats de l'algorithme MA-GP (k= 2, 4, 8, 16, 32, 64 et Imb=5%) obtenus avec 10 exécutions de 30 millions d'itérations.

Graphe	2		4		8		16		32		64	
	f^*_{min}	gain	f^*_{min}	gain	f^*_{min}	gain	f^*_{min}	gain	f^*_{min}	gain	f^*_{min}	gain
Add20	558	1,27	1143	-3,46	1670	-2,05	2036	-3,37	2350	-9,0	2913	-8,5
Data	189	4,42	368	-2,65	628	-7,8	1072	-6,5	1746	-3,80	2754	-1,75
3elt	87	0,00	197	-1,01	331	-0,90	559	-1,24	934	-2,30	1498	-2,41
Uk	22	22,22	41	0,00	99	20,73	164	6,49	292	10,19	458	5,05
Add32	10	0,00	33	0,00	63	-5,97	118	0,85	222	4,72	559	-6,4
bcsstk33	9914	0,00	20158	-5,01	33908	-0,61	54119	-2,57	76087	-3,47	104949	-3,05
whitaker3	126	0,00	377	-0,79	649	-1,37	1077	-1,37	1655	-1,84	2460	-2,96
Crack	182	0,00	361	0,28	668	-1,18	1068	-1,29	1650	-1,73	2478	-2,44
wing_nodal	1668	0,00	3522	-1,18	5340	-0,71	8181	-1,62	11579	-1,99	15489	-1,90
fe_4elt2	130	0,00	336	-3,72	582	-2,51	991	-1,59	1591	-3,05	2436	-3,18
Vibrobox	10310	0,00	18690	-2,88	23924	-1,0	31251	-1,40	39369	-4,39	46289	-1,82
bcsstk29	2818	0,00	8259	2,11	13552	-9,9	20935	-9,9	33544	-3,73	54163	-3,49
4elt	137	0,00	317	-0,63	516	-2,09	892	-2,62	1515	-1,43	2508	-2,83
fe_sphere	384	0,00	762	-0,52	1152	0,00	1682	-0,59	2439	-1,53	3479	-1,92
Cti	318	0,00	889	0,00	1701	-0,87	2729	-1,76	3960	-1,83	5635	-1,80
Memplus	5279	-1,01	9301	-0,98	11580	-2,55	13371	0,69	14084	-2,09	16276	-2,89
cs4	357	0,28	958	2,35	1480	0,54	2145	0,89	3035	-1,46	4148	-1,14
bcsstk30	6251	0,00	16189	-2,49	34071	-1,41	69071	-2,40	111563	-4,84	167994	-4,70
bcsstk31	2664	-0,45	7216	-8,41	13400	-1,19	22891	-5,33	37293	-3,32	56419	-4,63
fe_pwt	340	0,00	713	1,28	1407	-2,02	2755	-1,04	5380	-4,03	8044	-3,20
bcsstk32	4938	5,81	10640	9,38	19575	-8,13	36009	-6,03	60521	-3,88	90913	-5,46
fe_body	399	52,29	641	-8,82	1099	-4,02	1893	3,27	2985	-7,38	4838	-7,30
t60k	69	6,15	209	-0,95	527	12,85	916	7,51	1533	7,96	2446	10,13
wing	805	4,55	1695	3,61	2526	-0,98	4149	3,34	5945	-0,59	8016	-1,78
brack2	660	0,00	2735	-0,51	6621	-6,48	11237	-5,24	17055	-5,00	25291	-2,46
finan512	162	0,00	324	0,00	648	0,00	1296	0,00	2592	0,00	10732	-0,82
fe_tooth	3781	0,21	6718	-3,23	11279	-2,5	17128	-3,62	24758	-3,02	33839	-2,75
fe_rotor	1940	-0,82	7124	-8,2	12483	-5,32	20148	-2,54	30688	-5,91	44827	-3,32
598a	2340	0,17	7741	-2,97	15527	-1,01	25198	-2,24	37767	-3,40	54947	-3,40
fe_ocean	311	0,00	1698	-0,35	3940	-1,97	7475	-4,63	12385	-2,83	20120	-2,84
144	6358	-0,06	15574	2,12	24316	-5,06	36750	-4,49	54959	-5,47	76501	-4,90
wave	8581	0,21	16682	0,01	29069	-0,76	42376	-1,52	60557	-3,24	82971	-1,72
m14b	3804	0,05	13011	-2,91	25636	-6,67	41772	-3,97	64400	-3,80	94688	-2,53
Auto	9478	0,30	26309	-9,77	47004	-2,52	74954	-9,59	120923	-5,11	171241	-4,58
Gain moyen		2,81		-1,48		-1,63		-1,92		-2,43		-2,64

4.9 Analyse des résultats

Dans ce chapitre, nous avons testé les trois algorithmes avec un nombre réduit d'exemplaires (10 graphes, k=16 et *imb*=5%) et nous avons présenté les résultats obtenus avec 10 exécutions et 30 millions d'itérations (par exécution).

De ces tests, nous avons observé que Tabu-GP a battu 7 parmi les 10 records établis (algorithmes de l'archive et PMSATS). Ainsi, avec un simple algorithme de recherche locale, nous arrivons déjà à battre plusieurs records.

Suite aux mêmes tests effectués sur MA-GP, nous avons observé que l'algorithme a battu 8 des 10 records de la littérature et a amélioré en moyenne les records de -1,39%. De plus, MA-GP a amélioré les résultats de Tabu-GP de -3,05%. De ce fait, on a conclu que la recombinaison a été capable de guider la recherche. Cependant, on a observé aussi que MA-GP (comme Tabu-GP) a obtenu un résultat médiocre sur l'exemplaire du graphe uk (une dégradation du record de 6,49 %). Donc, il existe quelques graphes pour lesquels MA-GP n'obtient pas des bons résultats.

Dans le cas de Tabu-GPG, on a observé que l'algorithme a battu les 10 records de la littérature et a amélioré en moyenne les records de -2,73%. Il a été capable aussi d'améliorer le résultat de l'exemplaire du graphe uk de -4,55% pour lequel MA-GP et Tabu-GP ont obtenu des résultats médiocres.

En plus des tests sur l'ensemble réduit d'exemplaires, nous avons effectué des tests systématiques avec MA-GP pour les exemplaires formés de 34 graphes, k = [2, 4, 8, 16, 32, 64] et *Imb*=5%, donc 34x6=204 exemplaires en tout. Avec 10 exécutions et 30 millions d'itérations par exemplaire, MA-GP a amélioré 72% des records. Il a obtenu 12% de résultats à égalité avec le record et n'a pas pu améliorer 16 % des records. Dans 13,7% des cas (28 exemplaires parmi 204), MA-GP a amélioré de plus de 5% un record. Alors que dans 5,8% des cas, il dégrade de plus de 5% un record.

Lorsqu'on observe les résultats de MA-GP selon les valeurs de k, on remarque que l'amélioration moyenne par graphe augmente lorsque k augmente. En effet, pour k=2, MA-GP n'arrive pas en moyenne à améliorer les records (les résultats sont plus mauvais que les records de 2,81%). Cependant, lorsque k est supérieur à 2, MA-GP

obtient respectivement pour k=4, k=8, k=16, k=32 et k=64 les améliorations moyennes suivantes: -1,48%, -1,63%, -1,92%, -2,43% et -2,64%. On peut suspecter ainsi que l'apport des croisements est plus efficace pour un nombre élevé de classes. Pour le savoir précisément, il faudrait avoir les résultats de Tabu-GP avec les différentes valeurs de k.

Lorsqu'on observe les résultats selon les graphes, on remarque que les mauvais résultats sont concentrés sur un nombre réduit de graphes (uk, cs4, t60k et wing). Ces graphes ont une caractéristique commune : Il s'agit des 4 graphes qui pour lesquels le degré moyen des sommets est le plus faible (de 2,835 à 3,919). De ce fait, on conclut que la performance de MA-GP est moins important dans le cas des graphes peu denses.

Au point de vue du temps d'exécution, on observe que sur le jeu de tests réduit nos algorithmes mettent en moyenne entre 3,5 minutes (data, k=16, Imb=5%) et 1,6 heure (wave, k=16, Imb=5%), ceci pour effectuer une exécution de 30 millions d'itérations. Sur l'ensemble des exemplaires (34 graphes, k = [2, 4, 8, 16, 32, 64], Imb=5%), le temps d'exécution de MA-GP varie selon la valeur de k : Pour le graphe *data* le temps d'exécution varie de 2 minutes (k=2) à 27 minutes (k=64). Pour ces mêmes valeurs de k (2 et 64), le temps d'exécution de MA-GP varie pour le graphe *m14b* de 1 heure à 5,5 heures. Cependant, le graphe pour lequel le temps d'exécution est le plus élevé est *auto.* Pour ce graphe, le temps d'une exécution varie de 2,7 heures (k=2) à 13,8 heures (k=64).

Il faut noter que, dans nos tests, les résultats présentés sont été obtenus en réalisant seulement 10 exécutions de 30 millions d'itérations. Compte tenu de la variance des résultats, il serait possible de trouver des résultats encore sensiblement meilleurs simplement en effectuant un plus grand nombre d'exécutions. En revanche, ce que nous avons appelés les records sont les meilleurs résultats obtenus avec différents algorithmes pour lesquels conditions expérimentales ne sont pas clairement présentées. En particulier, les temps de calcul et/ou nombre d'exécutions réalisés ne sont généralement pas indiqués. Ceci désavantage nettement nos algorithmes. Malgré cela,

MA-GP améliore environ les trois quarts des résultats connus. MA-GP est donc clairement plus efficace que tous les algorithmes connus pour le problème.

Chapitre 5. CONCLUSION

Dans le cadre de notre recherche, nous avons proposé trois nouvelles heuristiques pour le problème de partitionnement de graphes. La première est un algorithme tabou (Tabu-GP). Une caractéristique de cet algorithme est la liste taboue dont la longueur varie de manière systématique entre de petites et des grandes valeurs. Les petites valeurs permettent d'intensifier la recherche alors que les plus grandes permettent de diversifier la recherche. La deuxième heuristique (Tabu-GPG) est un algorithme qui utilise une longue chaine de recherche locale mais qui n'utilise pas toujours une même fonction d'évaluation. Dans cet algorithme, on fait alterner deux types de phases de recherche : des phases durant lesquelles on utilise la fonction normale du problème (qui consiste à minimiser le nombre d'arêtes coupées) et d'autres durant lesquelles on utilise une fonction modifiée. La fonction modifiée a pour rôle de guider la configuration vers des zones favorables de l'espace de recherche. La troisième heuristique est un algorithme mémétique (MA-GP) qui fait évoluer un ensemble (population) de solutions. Il utilise un opérateur de recombinaison spécialisé pour le problème de partitionnement de graphe. Cet opérateur génère des nouvelles partitions à partir des intersections entre les classes de deux partitions. Sur les partitions générées, MA-GP applique un opérateur de recherche locale.

Nous avons testé les trois algorithmes à l'aide des jeux de données les plus couramment utilisés dans la littérature scientifique (Walshaw, 2006). Dans une première série de tests, nous avons testé nos trois algorithmes en utilisant un nombre réduit d'exemplaires (10 graphes, k=16, Imb=5%). Nous avons ensuite analysé les résultats obtenus et comparé ceux-ci aux meilleurs résultats connus pour ces exemplaires. Dans une seconde série de tests, nous avons testé MA-GP sur l'ensemble graphes de la littérature (Walshaw et al., 2006) avec un nombre de classes k = [2, 4, 8, 16, 32, 64] et un déséquilibre de 5%. Bien que nous n'ayons réalisé qu'un nombre réduit d'exécutions (10 exécutions), MA-GP a permis d'améliorer plus de 70 % des meilleurs résultats

jamais trouvés auparavant. De plus, pour 13,72% des exemplaires, MA-GP a amélioré de plus de 5% un record. MA-GP est donc sensiblement plus efficace que tous les algorithmes publiés à ce jour.

D'après les tests effectués sur le nombre réduit d'exemplaires, Tabu-GP a amélioré 7 records sur 10. De plus, d'autres résultats qui n'ont pas été présentés dans ce mémoire montrent que Tabu-GP est capable d'atteindre ou d'améliorer un grand nombre d'exemplaires de l'archive. Ainsi, avec un simple algorithme de recherche locale, on a amélioré les résultats des algorithmes multi-niveaux (PMSATS (banos et al., 200), Jostle Evolutionary (Soper et al., 2004), etc.). Ceci contredit l'idée selon laquelle on ne peut pas explorer efficacement l'espace de recherche.

De plus, Tabu-GPG a montré une grande facilité à battre les records. En effet, sur le nombre réduit d'exemplaires, il a amélioré tous les résultats de la littérature et il a même amélioré sensiblement certains des résultats MA-GP. Tabu-GPG est donc un algorithme prometteur qu'il faut étudier davantage.

Comme perspective de recherche, nous proposons d'effectuer des tests plus systématiques avec les algorithmes TabuGP et Tabu-GPG. En effet, les deux algorithmes ont montré une grande facilité à atteindre et à battre les meilleurs résultats sur un ensemble réduit d'exemplaires. Il faut donc étendre les tests expérimentaux sur tous les exemplaires et analyser en détail la performance des deux algorithmes.

De plus, il serait également intéressant de mettre en œuvre des techniques permettant de faciliter le réglage du paramètre de nos algorithmes (longueur de la liste taboue) pour éviter de devoir régler celui-ci exemplaire par exemplaire.

Une autre perspective de recherche consiste à hybrider MA-GP et Tabu-GPG. Cette hybridation consiste à remplacer la recherche locale de MA-GP avec Tabu-GPG. D'après des tests limités sur cet algorithme, on est aperçu que dans certains cas il permet de faire mieux que MA-GP et Tabu-GPG utilisés isolément.

Références

BANOS, R., GIL, C., ORTEGA, J., MONTOYA, F.G. (1999). «A Parallel Evolutionary Algorithm for Circuit Partitioning». Computer Society, p. 365-371.

BANOS, R., GIL, C., ORTEGA, J., MONTOYA, F.G (2004). «A Parallel Multilevel Metaheuristic for Graph partitioning ». Journal of heuristics, 10, p. 315-336.

BUI, T., JONES, C. (1993). «A heuristic for reducing fill in sparse matrix factorization». Proc. Of the 6th SIAM Conf. Parallel Processing for Scientific Computing, pp. 445-452.

DEHMEL, R.C. (1996). «Graph Partitioning». Consulté le 12 octobre 2006, tiré de http://www.cs.berkeley.edu/~demmel/cs267/lecture18/lecture18.html

FEILDER, C. (1975). «A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory». Czechoslovak Math. J., 25(100):633.

FIDUCCIA, C.M., MATTHEYSES, R.M. (1982). «A linear time heuristic for improving network partitions». Proc 19th IEEE Design Automation Conference, IEEE, pp.175-181.

GALINIER, P., HERTZ, A., ZUFFEREY. (2002). «Adaptative Memory Algorithms for Graph Coloring ». Computational Symposium on Graph Coloring and Generalizations (COLOR02), p. 75-82.

GALINIER, P., HAO, J.-K. (1999). «Hybrid evolutionary algorithms for graph coloring». Journal of Combinatorial Optimization, 3, 379-397.

GAREY, M.R., JOHNSON, D.S., STOCKMEYER. (1976). «Some simplified NP-complete graph problems». Theoret. Comput. Sci., 1, p. 237 –267.

HENDRICKSON, B., LELAND, R. (1995). «The Chaco User's Guide». Technical Report SAND93-2339, Sandia National Laboratories.

HENDRICKSON, B., LELAND, R. (1993). «A multilevel algorithm for partitioning graphs». Technical Report SAND93-1301, Sandia National Laboratories.

HERTZ, A. «Métaheuristiques : notes de cours: Optimisation Combinatoire - MTH6311». Consulté le 20 septembre 2006, tiré de http://www.gerad.ca/~alainh/Metaheuristiques.pdf .

KARYPIS, G., KUMAR, V. (1999). «A fast and high quality multilevel scheme for partitioning irregular graphs». SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359-392.

KARYPIS, G., AGGARWAL, R., KUMAR, V., Shekhar, S. (1997). «Hypergraph partitioning: Applications in VLSI domain ». Design and Automation Conference, pp. 526-529.

KARYPIS, G., KUMAR, V. (1995). «Analysis of Multilevel Graph Partitioning». Supercomputing.

KARYPIS, G., KUMAR, V. (1998). «Multilevel k-way Partitioning Scheme for Irregular Graphs ». J. Parallel Distrib. Comput. 48(1): 96-129.

KERNIGHAN, B., LIN, S. (1970). «An efficient heuristics procedure for graph partitioning graphs».Bell System Technical Journal, pp. 291-307.

MOHAR, B. (1988). «The Laplacian Spectrum of Graph». Technical Report, Dept of mathematics, Univ. of Ljubljana, 61111 Ljubljana, Yugoslavia.

PARLETT, B., SIMON, H., STRINGER, L. (1982). «Estimating the largest eigenvalue with the lanczos algorithm». Math. Comp. 38:135-165.

POTHEN, A., SIMON, H.D., Liou, K-P. (1990). «Partitioning sparse matrices with eigenvectors of graphs». SIAM Journal on Matrix Analysis and Applications, v.11 n.3, p.430-452.

SIMON, H.D., TENG, S.-H. (1997). «How Good is Recursive Bisection? » SIAM J. Sci. Comput., 18(5):1436-1445.

SIMON, H.D. (1994). «Partitioning of Unstructured Problems for Parallel Processing» Computing Systems in Eng, pp. 135-148.

SOPER, J., WALSHAW, C., CROSS, M. (2004). «A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning ». J. Global Optimization, 29(2), p. 225-241.

WALSHAW, C., CROSS, M. «The Graph Partitioning Archive». Consultée le 28 août 2006, tiré de http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/.

WALSHAW, C. (2004). «Multilevel Refinement for Combinatorial Optimisation Problems ». Annals Oper. Res., 131:325-372.

WALSHAW, C., CROSS, M. (2000). «Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm ». SIAM J. Sci. Comput., 22(1):63-80.