RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ SAAD DAHLEB DE BLIDA
FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

MÉMOIRE DE MASTER

Spécialité : Génie des Systèmes Informatiques

Intitulé :

Placement optimal ou pré-optimal, assujetti à des contraintes de temps et d'énergie, d'IPs dans un système VLSI

Présenté par : **STAMBOULI** A*bdelkarim*
REDAOUIA W*assim*

Promoteur : Dr. **MAHDOUM** A*li*

Organisme d'accueil : CDTA (Centre de Développement des Technologies Avancées)
Division de Microélectronique & Nanotechnologies

Devant le jury composé de :

Président : Mme. **REZZOUG** N*adia*

Examinateur1 : Mme. **ZAHRA** F.*Zahra*

Examinateur2 : Mme. **TOUBALINE** N*esrine*

- 2013/2014 -

Résumé

Avec l'avènement du développement technologique, il est actuellement possible d'intégrer des millions de transistors sur la même puce. Outre cet avantage, les délais des portes logiques deviennent de plus en plus courts à cause de la commutation rapide des transistors due à des tensions de seuil des transistors de plus en plus réduites. A côté de ces avantages, il existe malheureusement de nombreux problèmes auxquels les concepteurs doivent y faire face. Nous nous contentons de citer uniquement le problème en relation avec le contexte de ce projet. Les courbes de l'ITRS (*International Technology Roadmap for Semiconductors*) concernant les délais des portes logiques et des interconnexions montrent clairement que le premier type de délais ne cesse de s'améliorer au fur et à mesure que la longueur du canal du transistor se réduit. Malheureusement, ceci n'est pas le cas pour les interconnexions : la communication devient de plus en plus lente avec le développement technologique. Bien que des solutions partielles aient été trouvées telles que l'insertion d'amplificateurs, le remplacement de l'Aluminium par du Cuivre (IBM 2001), le problème se pose toujours. De plus, avec les systèmes actuels devenant de plus en plus complexes et se caractérisant par un fort degré de communications, il est devenu impossible d'utiliser un bus classique du fait que celui-ci est une ressource critique et partagée, ce qui considérablement réduit la bande passante. Une solution intermédiaire a été trouvée (utilisation de barres croisées -*cross-bars*-) mais reste néanmoins insuffisante. Il a donc fallu opter pour une meilleure solution, celle consistant à concevoir un réseau spécifique à l'application, répondant aux contraintes de largeur de bande, de consommation de puissance et de surface. Plusieurs méthodologies ont été proposées, dont les principales sont celles reposant sur les *Fat Trees*, les *Mesh Networks* et d'autres variantes de ces méthodologies. Dans le cadre d'un projet mené au CDTA, une autre méthodologie a été proposée et pour laquelle des tâches la réalisant sont en cours de développement.

Parmi ces tâches, une méthode de placement des IPs (composants) (Intellectual Property) du système a été réalisée. C'est une méthode exhaustive qui a pour avantage d'être exacte, mais ne peut être utilisée pour un nombre appréciable d'IPs du fait du temps CPU qu'elle engendre.

Afin que cette tâche de placement soit valable pour un système comportant un nombre quelconque de composants, ce travail consiste à développer une heuristique qui résou ce problème en un temps polynomiale.

Mots Clés : heuristique, kernighan-lin, VLSI, optimisation combinatoire, partitionnement de graphes

Abstract :

With the advent of technological development, it is now possible to integrate millions of transistors on the same chip. Besides this advantage, the time of the logic gates are becoming shorter due to the fast switching of the transistors due to threshold voltages of the transistors become smaller. Besides these advantages, there are unfortunately many issues that designers must cope. We just mention only the problem in relation to the context of this project. The curves of the ITRS (International Technology Roadmap for Semiconductors) on the duration of logic gates and interconnections clearly show that the first type of time continues to improve gradually as the channel length of the transistor is reduced. Unfortunately, this is not the case for the connections: the communication becomes increasingly slower with the technological development. While partial solutions have been found, such as inserting amplifiers, replacing the aluminum by Copper (IBM 2001), the problem still arises. In addition, with current systems becoming increasingly complex and characterized by a high degree of communication, it has become impossible to use a conventional bus because it is a critical and shared resource, which significantly reduces bandwidth. An interim solution was found (using -Cross-bars- crossbar) but remains insufficient. It was therefore necessary to choose a better solution, which is to design a specific network application, meeting the constraints of bandwidth, power consumption and area. Several methodologies have been proposed, of which the main ones are those based on the Fat Trees, the Mesh Networks and other variants of these methodologies. As part of a project conducted at CDTA, another methodology was proposed and for which the tasks are making in development.

Among these tasks, method of placement of IPs (components) of the system was performed. It is a comprehensive approach that has the advantage of being accurate, but can not be used for a significant number of IPs because the CPU time it generates. So that the task of placement is valid for a system with any number of components, this work is to develop a heuristic algorithm using partitioning Kernignan-Lin.

Keywords: heuristic Kernighan-lin, VLSI, combinatorial optimization, graph partitioning

Table des matières

Table des figures :

Table des Tableau :

Introduction générale

Le présent sujet a été défini dans le cadre d'un projet mené au CDTA. Parmi les tâches réalisant une nouvelle méthodologie de conception de réseaux sur puce, il en est une qui porte sur le placement des IPs (composants) du système. Une méthode exhaustive, donnant donc la solution exacte a été réalisée. Toutefois, du fait de la complexité algorithmique non polynomiale de ce problème, cette méthode ne peut faire face à un nombre considérable d'IPs du fait du temps CPU qu'elle engendre. Il est donc nécessaire de développer une heuristique ayant la double difficulté d'être à la fois polynomiale et générant une solution intéressante (pré-optimale, voire optimale).

Ce mémoire est organisé de la manière suivante. Dans le premier chapitre, nous définirons de manière plus précise la problématique et nous donnerons des généralités sur les circuits intégrés. Au deuxième chapitre nous expliquerons quelque notions de partitionnement de graphe suivra au troisième chapitre la description technique de l'heuristique développée (algorithmes, structures de données et formats d'enregistrement des fichiers). Les résultats seront présentés au quatrième chapitre et nous terminerons par une conclusion et des perspectives.

Chapitre 1

Problématique et généralité sur les circuits intégrés

1.1 Introduction :

Ce chapitre a pour objet de définir la problématique afin de mieux situer notre travail dans le contexte. Nous verrons qu'il s'agit d'un problème d'optimisation combinatoire pour lequel un algorithme à base d'une heuristique ou d'une méta-heuristique doit être développé.

1.2 Problématique :

Comme il a été souligné dans le résumé et l'introduction générale, le délai et la consommation de puissance dans les systèmes intégrés actuels sont largement dominés par les interconnexions. En effet, le développement des technologies des semi-conducteurs a rendu les transistors très rapides à cause de la diminution sans cesse croissante du canal du transistor. Ceci n'est malheureusement pas le cas pour les interconnexions où le temps de transfert des données devient plus important avec ce développement technologique. De plus, ces interconnexions engendrent, avec ces nouvelles technologies, une consommation de puissance supplémentaire (à cause de paramètres parasites) de loin non négligeable par rapport à celle induite dans les anciennes technologies. Un intérêt particulier doit être donc donné aux interconnexions pour la conception de systèmes intégrés avec les nouvelles technologies, surtout que les systèmes actuels sont très complexes et se caractérisent par un fort degré de communications.

Une interconnexion est implémentée par un métal (cuivre, aluminium, ...) et est caractérisée par une résistance, une capacité et une inductance. Le temps de transfert d'une donnée sur cette interconnexion est proportionnel au produit de ces paramètres. A leur tour, ces trois paramètres dépendent de la topologie de cette interconnexion (longueur, largeur), de sa nature (aluminium, cuivre, etc.) et du procédé technologique utilisé pour l'implémentation. La consommation de la puissance induite par un transfert de données sur une interconnexion dépendra aussi de ces trois paramètres.

En supposant donnés le procédé technologique et la nature de l'interconnexion, le délai de transfert de données sur une interconnexion et la consommation de la puissance induite par cette interconnexion dépendront fortement de la topologie (longueur et largeur) de l'interconnexion. En effet, la capacité de l'interconnexion

sera tout simplement la somme d'une capacité dépendant de la longueur et de celle dépendant de la surface de l'interconnexion.

La capacité due à la longueur de l'interconnexion est le produit de la capacité par unité de longueur (valeur dépendant du procédé technologique utilisé) par la longueur de l'interconnexion.

La capacité due à la surface de l'interconnexion est le produit de la capacité par unité de surface (valeur dépendant aussi du procédé technologique utilisé) par la surface de l'interconnexion.

Il en est de même pour les résistances et les inductances. Ceci revient à dire que, pour implémenter un système complexe dans une technologie donnée, le concepteur du système doit grandement tenir compte de la topologie de ces interconnexions afin de satisfaire les contraintes de surface, de temps et de consommation de puissance. Sans l'apport d'un outil informatique, un tel travail est impossible à réaliser. D'où l'intérêt de faire appel à l'informatique.

Du point de vue algorithmique, ce problème revient à disposer tous les composants du système de façon à satisfaire les trois contraintes. Il s'agit donc d'un problème d'optimisation combinatoire pour lequel l'obtention d'une solution exacte en un temps polynomial n'est pas possible.

Dans le cadre d'un projet mené au CDTA et portant sur une nouvelle méthodologie de conception d'un réseau sur puce, une méthode exhaustive a été réalisée. Cette méthode a pour avantage de générer une solution exacte, mais ne peut être utilisée pour un nombre considérable de composants à cause du temps CPU qu'elle engendre. D'où l'utilité du travail présenté dans ce mémoire qui consiste à développer une heuristique ayant la double difficulté d'être polynomiale et générant une solution intéressante (pré-optimale, voire optimale dans certains cas).

L'idée générale repose sur les degrés de communications: il s'agit de placer, dans la mesure du possible (le problème est combinatoire), deux composants côte à côte ou, au cas échéant, dans le même voisinage, lorsque ces composants se communiquent fortement. Le but est d'en réduire la capacité, la résistance et l'inductance induites par l'interconnexion reliant ces deux composants.

Il est aussi prévu, en perspectives, de comparer un tel placement à celui obtenu par une technique de coloration de graphes développée au CDTA. La comparaison portera sur les temps et la consommation de puissance obtenus par les deux méthodes, en utilisant des algorithmes d'estimation de temps et de consommation de puissance développés au CDTA.

1.3 Introduction sur les systèmes VLSI :

1.3.1 L'évolution technologique :

Depuis une cinquantaine d'années, l'évolution de la complexité des circuits intégrés double tous les 18 mois (loi de Moore [MOO65]). Cette évolution exponentielle a permis de réaliser, de manière monolithique, des organes électroniques de plus en plus complexes qui étaient auparavant réalisés sous la forme d'armoires (par exemple : des processeurs, des mémoires, des commutateurs téléphoniques…).

Le principal moteur de cette évolution réside dans la diminution régulière de la taille des motifs de dessin des circuits intégrés. Partis de quelques dizaines de microns dans les années 1960, ceux-ci sont maintenant de 20 nm en 2014 (>10,000,000,000 transistors) [INT14], et toute montre que cette évolution n'est pas terminée.

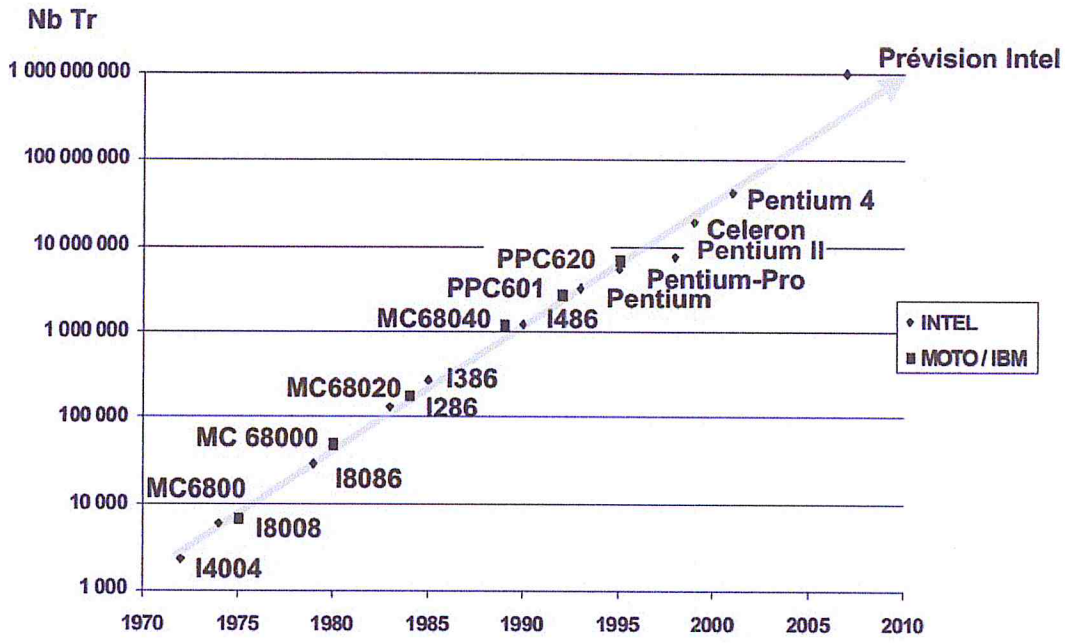


Figure 1.1 Evolution exponentielle de la complexité des microprocesseurs [MOO67]

La capacité de l'industrie microélectronique à poursuivre cette évolution est proprement incroyable. Elle surprend tout le monde, y compris les experts. Ceux-ci, réunis au sein d'une organisation appelée SIA (Semiconductor Industry Association), publient régulièrement des prédictions (appelées ITRS pour International Technology Roadmap for Semiconductors) qui s'avèrent systématiquement sous-évaluées pour un futur qui dépasse trois ans, c'est-à-dire l'horizon de leurs recherches. La meilleure prédiction est encore le simple prolongement du passé,

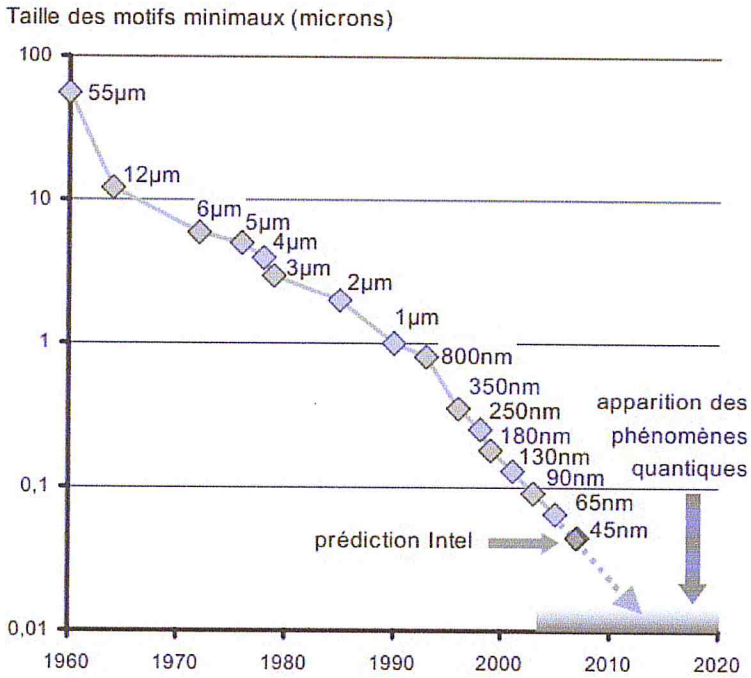


Figure 1.2 Evolution de la taille des motifs minimaux de la technologie [FRA12]

aussi incroyable qu'il puisse être.

1.4 Domaines d'application :

Cette technique de conception trouve son intérêt pour la conception de circuits (c'est à-dire non réalisables par la programmation d'un microcontrôleur pré-existant).

Ceux-ci peuvent être :

– soit des circuits rapides devant être produits en grande série (c'est-à-dire devant être optimisés en termes de surface et de rapidité). L'augmentation de la

performance des technologies récentes diminue le domaine d'emploi de cette technique par l'accroissement des performances des microcontrôleurs standard ;

– soit des circuits à réaliser à l'aide de circuits paramétrables (par exemple des FPGA).

On trouve de tels circuits : dans les télécoms (encrypteurs/décrypteurs de messages, filtres numériques, …), dans le matériel informatique (contrôleurs de périphériques, …), dans l'horlogerie (montres complexes, …),

– dans les appareils grand-public : décodeurs pour lecteurs de CD, …

Les circuits spécifiques gagneront à être réalisés en polyphasé avec des techniques d'assemblage et de compilation de silicium. La réalisation sur des circuits paramétrables se fera toujours en monophasé par compilation.

1.5 Conclusion :

On peut représenter plusieurs problèmes à l'aide d'un graphe et ainsi appliqué un ensemble d'algorithme déjà défini qui peuvent les résoudre en un temps polynomial ce qui nous laissent élargir le domaine d'application de notre application.

Dans ce but nous avons décidé de voir le problème de disposition d'IPs d'une manière abstraite pour avoir une solution générale et non une solution qui résous un seul problème.

Ce présent chapitre nous a permis de définir brièvement la problématique et montrer l'importance des interconnexions dans l'implémentation des systèmes actuels dans les nouvelles technologies des semi-conducteurs.

A cause de sa nature combinatoire, un placement efficace (tenant compte des aspects de temps et de consommation de puissance) ne peut être fait manuellement. Toujours parce qu'il s'agit d'un problème d'optimisation combinatoire, un algorithme à base d'une heuristique ou d'une méta-heuristique doit être développé afin de générer une solution intéressante en un temps CPU raisonnable.

Comme perspectives à ce travail, il est prévu d'utiliser des algorithmes d'estimation de temps et de consommation de puissance développés au CDTA pour

comparer les caractéristiques obtenues par notre heuristique à celles obtenues par une technique de coloration de graphes développée au CDTA.

Chapitre 2

Etude technique de la problématique

2.1 Introduction au problème du partitionnement de graphe :

De par l'étendue de ses applications, le partitionnement de graphe se décline en une multitude de problèmes. Cependant, nous pouvons les regrouper en deux grandes catégories. La nature d'un problème du partitionnement de graphe est très différente selon que l'on cherche à obtenir des parties de tailles très proches ou sans contraintes de taille. Cette constatation est la pierre angulaire de la différentiation que nous faisons. Ainsi, nous étudierons deux problèmes : dans le cas où les parties sont de tailles semblables, on parlera de partitionnement contraint ; dans le cas contraire, on parlera de partitionnement non contraint.

Avant de donner une définition précise de ces deux problèmes, ce chapitre présente quelques rappels de mathématiques et de théorie des graphes utilisés dans cette thèse. Ils seront suivis d'une description formelle du partitionnement de graphe et de ses objectifs. Puis les deux problèmes de partitionnement seront présentés : le partitionnement contraint et le partitionnement non contraint. Une avant-dernière section étudiera la NP difficulté de ces problèmes de partitionnement. La dernière section classera les différents problèmes de partitionnement que nous étudierons en fonction de leur caractère contraint ou non contraint.

2.2 Notions mathématiques : [CHA07]

2.2.1 Cardinal d'un ensemble :

Soit X un ensemble fini d'éléments. Le cardinal de l'ensemble X est égal au nombre d'éléments de X, et on le note *card*(X).

2.2.2 Partition :

Soit un ensemble S quelconque. Un ensemble P de sous-ensembles de S est appelé une partition de S si :

1. Aucun élément de P n'est vide ;
2. L'union des éléments de P est égal à S ;
3. Les éléments de P sont deux à deux disjoints.

Les éléments de P sont appelés les parties de la partition P. Le cardinal de la partition P est alors le nombre de parties de P.

2.2.3 Problème d'optimisation combinatoire :

Un problème d'optimisation combinatoire se définit à partir d'un triplet (E, *p*, *f*) tel que :

– E est un ensemble discret appelé espace des solutions (aussi appelé espace de recherche) ;

– p est un prédicat sur E, i.e. une fonction de E dans {vrai; faux} ;

– f : E → R associe à tout élément x ∈ E un coût f(x). f est appelée fonction objectif ou fonction de coût.

p permet de créer un ensemble E_a = {x ∈ E tel que P (x) est vrai}. L'ensemble E_a est appelé l'ensemble des solutions admissibles du problème.

Il s'agit de trouver l'élément $\tilde{x} \in E_a$ qui minimise f :

$$f(\tilde{x}) = \min_{x \in Ea} f(x) .$$

Remarque: Le problème d'optimisation combinatoire consistant à chercher un élément maximum au lieu d'un élément minimum est de même nature puisque :

$$\max_{x \in Ea} f(x) = - \min_{x \in Ea}(- f(x))$$

Les problèmes d'optimisation combinatoire sont en général très coûteux à résoudre de façon optimale. C'est en particulier le cas du partitionnement de graphe, dont nous verrons, grâce à la théorie de la complexité, qu'il est NP-difficile.

On peut décrire E comme l'ensemble des solutions du problème d'optimisation combinatoire que l'on cherche à résoudre quand on ne tient pas compte des contraintes de ce problème. Par exemple, le problème d'optimisation combinatoire, qui vise à trouver une partition des sommets d'un graphe G = (S, A) en k parties de tailles égales (on choisit k diviseur de card(S)), aura pour ensemble de solutions E l'ensemble des partitions de S dont le nombre de parties va de un au nombre d'éléments de S, et dont les parties sont de tailles quelconques. Par contre, l'ensemble des solutions admissibles du problème, E_a, doit tenir compte des contraintes de celui-ci. Ainsi, dans notre exemple, l'ensemble E_a sera constitué des partitions de S en k parties de tailles égales.

2.2.4 Optimum global, optimum local :

Soit un problème d'optimisation combinatoire (E, p, f) et E_a l'ensemble des solutions admissibles du problème induit par p. Soit $\tilde{x} \in E_a$.

– si l'on peut prouver que $\forall\ x \in E_a$, $f(\tilde{x}) \leq f(x)$, alors on dira que $\tilde{x}$ est l'optimum (minimum) global du problème ;

– s'il existe un ensemble $V \subset E_a$, contenant $\tilde{x}$, et au moins deux éléments, tel que $\forall\ x \in V$, $f(\tilde{x}) \leq f(x)$, alors on dira que $\tilde{x}$ est un optimum (minimum) local du problème.

L'espace des solutions E dispose d'une « topologie ». Connaître les caractéristiques de celle-ci est très utile pour comprendre le but du fonctionnement des métaheuristiques. Cette topologie résulte de la notion de proximité entre deux solutions, aussi appelées dans ce cas configurations.

La distance entre deux configurations représente le nombre minimum de modifications élémentaires nécessaires pour passer de l'une à l'autre. De plus, puisqu'à chaque configuration x est associée une valeur $f(x)$, l'espace des solutions est caractérisé par une courbe à plusieurs dimensions appelée « paysage énergétique ». Dans ce paysage énergétique, les optima locaux ou globaux forment des « puits énergétiques » autour d'eux. Avant de décrire une solution du problème comme étant un minimum local, on vérifie en général que l'ensemble V est suffisamment « grand » par rapport à la taille de E_a. La figure 1.1 représente l'équivalent en continu du paysage énergétique d'une fonction de coût pour un espace des solutions à une dimension.

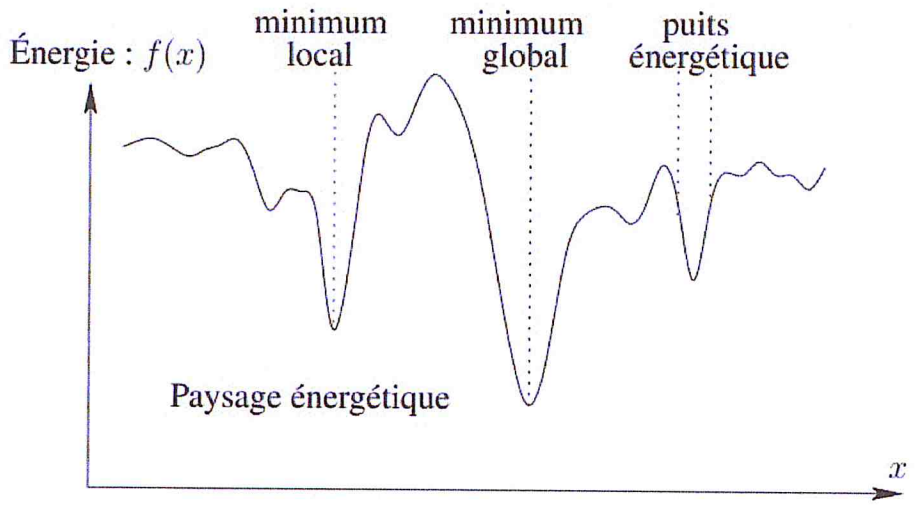


Figure 2.1 Paysage énergétique dans le cadre continu d'une fonction de coût pour un espace des solutions à une dimension.[CHA07]

2.3 Graphes :

Cette section rappelle quelques définitions de la théorie des graphes.

2.3.1 Graphe :

Soient S un ensemble de $n_S \in \mathbb{N}^*$ éléments et A un ensemble de $n_A \in \mathbb{N}$ couples d'éléments de S. On appelle graphe G le couple (S, A). Les éléments de S sont appelés sommets du graphe et ceux de A, les arcs ou arêtes du graphe, suivant qu'ils sont orientés ou non.

Lorsqu'il existe une relation entre au moins trois sommets, on dit que ces sommets sont reliés par une hyper-arête. Par extension, on ne parle plus de graphe, mais d'hypergraphe. [CHA07]

2.3.2 Hypergraphe :

Soient un ensemble S d'éléments et un ensemble A de sous-ensembles non vides d'éléments de S. On appelle hypergraphe G le couple (S, A). Les éléments de A sont appelés hyper-arêtes.

Lorsque les associations entre les sommets ne sont pas réciproques, par exemple si le sommet a peut être en relation avec le sommet b, sans que le sommet b soit en

relation avec a, on dit que les arêtes du graphe sont orientées, et on les appelle alors arcs.

2.3.3 Graphe orienté, non orienté :

Soit un graphe G = (S, A). Si $\forall(x; y) \in A, (y, x) \in A$, alors le graphe est dit non orienté et les éléments de A sont appelés arêtes du graphe. Dans ce cas, on note indifféremment une arête : $a \in A, (s, s`) \in A$ avec s et s` dans S, ou encore (s`, s). Dans le cas contraire, le graphe est dit orienté et les éléments de A sont appelés arcs du graphe.

2.3.4 Degré d'un sommet :

Dans un graphe non orienté G = (S, A), le degré d'un sommet $s \in S$ est le nombre d'arêtes auxquelles ce sommet appartient :

$$deg(s) = card(\{(s, s`) \in A, s` \in S\}) .$$

2.3.5 Boucle et arête multiple :

Une arête est appelée une boucle si ses deux extrémités sont identiques.

Si deux arêtes possèdent les mêmes extrémités, alors on dit que l'arête est multiple et que ces deux arêtes sont parallèles. Dans ce cas, la multiplicité d'une arête est le nombre total de ses arêtes parallèles, y compris elle-même.

2.3.6 Graphe simple :

Un graphe est dit simple s'il n'a ni boucle ni arête multiple.

Lorsque les associations entre les sommets sont quantifiés, on parle du poids des arêtes du graphe et on dit que le graphe est valué.

2.3.7 Graphe valué ou pondéré :

Soit un graphe G = (S, A). On dit que le graphe est valué (ou pondéré) si à chaque élément a de A est associé une valeur entière $poids(a) \in \mathbb{N}^*$.

La valeur poids(a) est appelée le poids de a.

Par extension, on considère qu'un couple de sommets $(s, s`) \in S^2$ tel que $(s, s`) \notin A$ possède un poids nul : poids (s, s`) = 0.

Le poids d'un sous-ensemble X d'éléments de A, est la somme des poids des éléments de X :

$$poids(X) = \sum_{a \in X} \text{poids}(a).$$

De même, on associe parfois à l'élément s de S un entier strictement positif appelé le poids de s et noté *poids(s)*.

Remarque : Dans la théorie des graphes, le poids d'une arête ou d'un sommet peut être réel.

Il faut faire attention à ne pas confondre le cardinal d'un ensemble de sommets (respectivement d'arêtes) et le poids d'un ensemble de sommets (respectivement d'arêtes). Ceux-ci sont égaux si les sommets (respectivement les arêtes) ne sont pas pondérés.

Dans le cas des graphes valués, on redéfinit le degré d'un sommet $s \in S$ comme étant la somme des poids des arêtes adjacentes à ce sommet. Ainsi :

$$\deg(s) = \sum_{(s,s`) \in A} \text{poids}(s, s`) .$$

À tout graphe non valué, on peut associer à chaque arête et à chaque sommet de ce graphe un poids unitaire.

Pour tout graphe G = (S, A), on peut définir une bijection $u : S \rightarrow \{1, \ldots, n\}$ qui à tout élément de S associe un entier. La fonction *u* permet de numéroter les éléments de S. Pour plus de commodité par la suite, à tout élément de l'ensemble des sommets S d'un graphe G, on associera un rang dans S, et on confondra le sommet si de S avec son rang *i* dans S. Par exemple, pour tout $(s_i, s_j) \in A$, poids (s_i, s_j) sera noté poids (i, j).

Pour un graphe quelconque, on peut définir une unique fonction *poids* : $A \rightarrow \mathbb{N}$ qui à toute arête de ce graphe associe son poids. Cette fonction poids peut être représentée sous la forme d'une matrice. On définit la matrice d'adjacence d'un graphe G comme étant la matrice associant à chaque arête de G son poids.

2.3.8 Matrices d'adjacence et des degrés :

Soit un graphe simple G = (S, A).La matrice M_{Adj} de dimensions $n_S \times n_s$, telle que $\forall (i, j) \in \{1, \ldots, n_S\}^2$:

$$(M_{Adj})_{ij} = \begin{cases} 0 & si\ i = j \\ p(i,j) & sinon \end{cases},$$

est appelée matrice d'adjacence du graphe G.

La matrice M_{Deg} de dimensions $n_S \times n_s$, telle que $\forall (i, j) \in \{1, \ldots, n_S\}^2$:

$$(M_{Deg})_{ij} = \begin{cases} \deg(i) = \sum_{k=1}^{ns} p(i,k) & si\ i = j \\ 0 & sinon \end{cases},$$

est appelée matrice des degrés du graphe G.

2.3.9 Adjacence :

Soient un graphe G = (S, A) et une arête a = (s, s`) ∈ A. On dit que les sommets s et s` sont les sommets adjacents à l'arête a. De même, on dit que a est l'arête adjacente aux sommets s et s`.

2.4 Description formelle du problème du partitionnement de graphe :

Avant de présenter ce qu'est le problème général du partitionnement de graphe, il nous faut définir ce qu'est la partition d'un graphe. Comme nous l'avons vu, un graphe est un couple formé d'un ensemble de sommets et d'un ensemble d'arêtes. Il est donc possible de faire la partition, au sens mathématique, de l'ensemble des sommets comme de l'ensemble des arêtes. Cependant, bien que certains problèmes cherchent à partitionner les arêtes d'un graphe [HOL81], on entend le plus souvent par partition d'un graphe, la partition des sommets de ce graphe.

2.4.1 Partition des sommets d'un graphe :

Soient un graphe G = (S, A) et un ensemble de k sous-ensembles de S, noté P_k = $\{S_1, \ldots, S_k\}$. On dit que P_k est une partition de G si :

– aucun sous-ensemble de S qui est élément de Pk n'est vide :

$$\forall i \in \{1, \ldots, k\}, S_i \neq \emptyset ;$$

– les sous-ensembles de S qui sont éléments de Pk sont disjoints deux à deux :

$$\forall (i, j) \in \{1, \ldots, k\}^2, i \neq j, S_i \cap S_j = \emptyset ;$$

– l'union de tous les éléments de P_k est S :

$$\cup_{i=0}^{k} S_i = S .$$

Les éléments S_i de P_k sont appelés les parties de la partition.

Le nombre *k* est appelé le cardinal de la partition, ou encore le nombre de parties de la partition.

2.4.2 Problème général du k-partitionnement de graphe :

Soit un graphe G = (S, A). Nous allons définir le triplet (E, *p*, *f*) caractérisant le problème général du *k*-partitionnement de graphe :

– l'espace des solutions E est défini comme l'ensemble des partitions de S (ces partitions ont un cardinal compris entre 1 et card(S) ;

– soit un entier $k \geq 2$ et p` un prédicat défini par P ∈ E, p`(P) = vrai ⇔ card(P) = *k*. Le prédicat *p* est défini sur E tel que *p*(P) = vrai ⇔ p`(P) = vrai ;

– soit f : E ⟶ R une fonction objectif.

L'ensemble des solutions admissibles du problème est défini par :

$$E_a = \{P \in E \text{ tel que } p(P) = \text{vrai}\}.$$

Le problème général du partitionnement de graphe consiste à trouver la partition $\tilde{P}_k \in E_a$ qui minimise f :

$$f(\tilde{P}_k) = \min_{Pk \in Ea} f(P_k) .$$

Par définition d'un graphe, l'ensemble S est fini. Le nombre de partitions d'un ensemble à *n* éléments est appelé le nombre de Bell relatif à *n*, et noté B_n. C'est le nombre de relations d'équivalences distinctes sur un ensemble à *n* éléments. On pose $B_0 = 1$. Le nombre de Bell vérifie la relation de récurrence suivante :

$$B_{n+1} = \sum_{k=0}^{n} C_n^k B_k ,$$

où $C_n^k = \frac{n!}{k!(n-k)!}$ est le coefficient binomial de n et k, c-à-d le nombre de parties à k éléments d'un ensemble de n éléments.

Le nombre de partitions en k parties d'un ensemble à n éléments est quant à lui appelé le nombre de Stirling de seconde espèce. Ce nombre est noté $S_{n,k}$ et vaut :

$$S_{n,k} = \sum_{i=0}^{k} (-1)^i \frac{(k-i)^n}{i!(k-i)!}$$

Il existe une relation entre le nombre de Bell et le nombre de Stirling de seconde espèce :

$$B_n = \sum_{i=1}^{n} S_{n,i} .$$

Ainsi, l'ensemble E est un ensemble fini. C'est donc bien un ensemble discret comme le veut la définition d'un problème d'optimisation combinatoire.

Le problème général du partitionnement de graphe se définit en fonction d'un couple prédicat, fonction objectif : (p, f). Ainsi, le partitionnement de graphe peut se diviser en de nombreux sous problèmes, suivant la nature de p et de f. Cependant, ces problèmes peuvent se classer en 2 catégories distinctes selon la nature du prédicat p.

2.5 Fonctions objectifs pour le partitionnement de graphe :

Les différentes fonctions objectifs pour le partitionnement de graphe s'articulent toutes autour de deux concepts : le coût de coupe entre les parties de la partition et le poids de ces parties.

Soit un graphe G = (S, A). Soient deux sous-ensembles $S_a \subseteq S$ et $S_b \subseteq S$, on définit le coût de coupe entre ces deux sous-ensembles par :

$$\text{Coupe}(S_a, S_b) = \sum_{u \in S_a, v \in S_b} \text{poids}(u, v).$$

Dans le reste de cette section, on choisit une partition $P_k = \{S_1, \ldots, S_k\}$ de S en k parties. Les fonctions objectifs présentées seront définies par rapport à cette partition.

La plus simple des fonctions objectifs utilisée en partitionnement de graphe est appelée le coût de coupe d'une partition, ou en anglais, ***cut***. Elle cherche à minimiser la somme des poids des arêtes entre les parties de la partition P_k. Elle est déjà utilisée par ***Brian Kernighan*** et ***Shen Lin*** dans [KER70] :

$$\text{coupe}(P_k) = \sum_{i<j} \text{coupe}(Si, Sj)$$

$$= \frac{1}{2} \sum_{i=1}^{k} coupe\ (S_i, S - S_i).$$

La fonction objectif qui vise à minimiser pour chaque partie le rapport entre son coût de coupe et son poids est appelée ratio de coupe, en anglais ***ratio cut***. Elle est introduite par ***Yen-Chuen Wei*** et ***Chung-Kuan Cheng*** dans [WEI89] :

$$\text{ratio}(P_k) = \sum_{i=1}^{k} \frac{coupe(S_i, S - S_i)}{poids(S_i)}.$$

Cependant, quand le terme ratio de coupe est utilisé, il faut être averti que d'autres fonctions objectives sont aussi appelées du même nom dans la littérature [WAN03].

La troisième et dernière fonction largement utilisée pour le partitionnement de graphe est appelée coût normalisé, ou ***normalized cut*** en anglais. Cette fonction, qui est la plus récente, a été présentée par ***Jianbo Chi*** et ***Jitendra Malik*** dans [SHI00]. Elle cherche à minimiser, pour chaque partie, le rapport entre son coût de coupe et la somme du poids des arêtes adjacentes à au moins un de ses sommets.

Autrement dit, elle cherche à minimiser, pour chaque partie, le rapport entre la somme du poids des arêtes adjacentes a exactement un de ses sommets et la somme du poids des arêtes adjacentes à au moins un de ses sommets :

$$\text{norm}(P_k) = \sum_{i=1}^{k} \frac{coupe\ (S_i, S - S_i)}{coupe\ (S_i, S)}$$

$$= \sum_{i=1}^{k} 1 - \frac{coupe\ (S_i, S - S_i)}{coupe\ (S_i, S)}.$$

La fonction de coût normalisé permet d'isoler les régions du graphe dont les sommets sont très liés entre eux. Ses auteurs l'ont créée pour remplacer le ratio de coupe dans leur problème de segmentation d'image.

Toutes ces fonctions objectives permettent de définir l'« énergie » d'une partie d'une partition. Cette énergie correspond à la valeur apportée par la partition à la fonction objective. Soit une partition $P_k = \{S_1, \ldots, S_k\}$ de S en *k* éléments. Dans le cas du coût de coupe, l'énergie d'une partie S_i sera :

$$\text{énergie}_{cc}(S_i) = \text{coupe}(S_i, S - S_i).$$

Dans le cas du ratio de coupe, l'énergie d'une partie S_i sera :

$$\text{énergie}_{rc}(S_i) = \frac{\text{coupe}(S_i, S - S_i)}{\text{coupe}(S_i, S)}.$$

Dans le cas de la coupe normalisée, l'énergie d'une partie S_i sera :

$$\text{energie}_{cn}(S_i) = \frac{\text{coupe}(S_i, S - S_i)}{\text{coupe}(S_i, S)}.$$

2.6 Le partitionnement contraint :

Le partitionnement de graphe s'attache au problème du partitionnement contraint. C'est ce problème que résolvent la plupart des outils de partitionnement de graphe tels que ***Metis, Jostle, Chaco, Scotch*** ou ***Party***.

L'un des premiers papiers à présenter ce problème fût celui de ***Brian Kernighan*** et ***Shen Lin*** [KER70]. C'est aussi ce problème qui sert le plus souvent à illustrer les nouvelles méthodes de partitionnement de graphe. C'est le cas pour la méthode spectrale présentée par ***Alex Pothen***, ***Horst Simon*** et ***Kang-Pu Liou*** [POT90], ainsi que pour la méthode multi-niveaux introduite par ***Stephen Barnard*** et ***Horst Simon*** [BAR93, BAR94].

Le partitionnement contraint sert, entre autres, à résoudre des problèmes d'ingénierie, de calcul haute performance, de résolution de systèmes linéaires, de maillage, et dans certains cas, de conception de circuits intégrés. Cependant, certaines critiques ont été formulées concernant l'utilisation de cette approche pour modéliser ces problèmes, c'est notamment le cas dans [HEN98].

Dans la littérature, le partitionnement de graphe contraint est appelé ***multi-way graph partitioning*** [YAR00] ou ***k-way graph partitioning*** [KAR98d].

Le problème du partitionnement de graphe contraint consiste à trouver une partition en k parties qui minimise une fonction objectif f et dont la balance de partitionnement soit unitaire, i.e. les parties doivent avoir le même poids, à une unité près. Or, si une partition P_k a des parties de même poids, alors son ratio de coupe devient égal à son coût de coupe, à une constante multiplicative près. En effet, dans ce cas :

$$\text{ratio } (P_k) = \sum_{i=1}^{k} \frac{coupe\ (S_i, S-S_i)}{coupe\ (S_i, S)}$$

$$= \frac{k}{poids\ (S)} \sum_{i=1}^{k} coupe\ (S_i, S - S_i)$$

$$= \frac{k}{poids\ (S)} \text{coupe } (P_k)$$

Ainsi, pour le problème du partitionnement de graphe contraint, minimiser le ratio de coupe est équivalent à minimiser le coût de coupe (remarque : si k n'est pas diviseur de *poids*(S), alors les deux minimisations de ces fonctions restent cependant très proches).

De plus, dans le cas où est cherchée une partition P_k dont les parties ont le même poids, l'utilisation du coût normalisé perd son sens. En effet, cette fonction de coût a été créée pour isoler des autres régions celles dont les sommets sont très liés, ce qui va le plus souvent dans un sens contraire à la recherche de parties de tailles égales. Cette fonction de coût n'est donc jamais utilisée dans les problèmes de partitionnement contraint.

Ainsi, le seul réel objectif du partitionnement de graphe contraint est de minimiser le coût de coupe des arêtes entre les parties du graphe. Ceci nous permet de donner sa définition :

- Soient un graphe G = (S, A), un nombre de parties k et une balance de partitionnement maximale, *balmax*. Soit l'espace des solutions E, défini par le problème général du partitionnement de graphe. L'ensemble E_a des solutions admissibles du problème est défini par :

$E_a = \{P_i \in E$ tel que card$(P_i) = k$ et bal$(P_i) \leq$ balmax$\}$.

Le problème du partitionnement contraint consiste à trouver $\tilde{x} \in E_a$ qui minimise la fonction de coût de coupe :

$$\text{coupe}(\tilde{x}) = \min_{x \in E_a} coupe(x).$$

Remarques :

– la balance de partitionnement maximale des problèmes de partitionnement contraint est souvent très petite : *balmax* ∈ [1, 0; 1, 05] ;

– dans les problèmes de partitionnement contraint, une petite relaxation de la balance de partitionnement est souvent acceptée, car il est connu qu'avec une valeur de la balance légèrement supérieure à l'unité, peuvent être trouvées des partitions de coût de coupe bien inférieures, sans que cela soit techniquement nuisible à la résolution du problème [SIM97].

2.7 Le partitionnement non contraint :

Le problème du partitionnement non contraint est un problème un peu plus « exotique » que celui du partitionnement contraint. Nous utilisons le terme exotique car ce problème se dérive en de nombreux problèmes très proches les uns des autres et dont la fonction objectif est régulièrement améliorée, ou tout du moins changée. Si le premier but du partitionnement contraint consiste à trouver une partition respectant une balance très restrictive, celui du partitionnement non contraint est de minimiser une fonction objectif. Alors que le second but du partitionnement contraint et la minimisation de la fonction de coût, le partitionnement contraint n'en a en général pas.

Le partitionnement non contraint est assez proche dans sa formulation du partitionnement de données ou ***clustering*** en anglais. Le problème du partitionnement de données est de regrouper des éléments similaires dans des ensembles les plus distincts possibles. Pour comparer deux éléments, la notion de distance entre eux est utilisée. Ce qui ramène le problème du partitionnement de données à créer des sous-ensembles d'éléments les plus distants possibles les uns des autres. Or, on peut facilement fabriquer un graphe dont les sommets sont ces éléments et dont les arêtes ont pour poids l'inverse de la distance entre deux éléments. Dans le cas où le nombre de parties cherchées est fixe, un problème de partitionnement de données se ramène à un problème de partitionnement non contraint. Il en résulte que de nombreux travaux ont cherché à résoudre des problèmes de partitionnement de données à l'aide d'outils de partitionnement de graphe.

Le partitionnement non contraint est très utilisé pour la segmentation d'images [BEN05], mais aussi pour la classification de textes [ZHA04]. Dans le cas de la segmentation d'images, de nombreux articles comparent différentes fonctions objectifs et proposent de nouvelles fonctions répondant à de nouveaux critères [FEL04]. D'autres problèmes plus originaux, comme le découpage de l'espace aérien, sont aussi des problèmes de partitionnement non contraint. Enfin, un certain nombre d'articles résolvent des problèmes de partitionnement de circuit VLSI par une approche de partitionnement de graphe non contraint [ALP95a]. Cependant, dans le cas des circuits VLSI, ceux-ci sont dans la majorité des cas considérés comme des hypergraphes plutôt que des graphes [KER70].

Les fonctions objectifs que cherche à minimiser un problème de partitionnement non contraint sont variées. La section 1.4 présente le ratio de coupe et le coût normalisé. Ces deux fonctions objectifs, avec quelques variantes très proches, sont celles qui reviennent le plus souvent dans les problèmes de partitionnement non contraint. Dans certains cas, et dans une certaine mesure, on peut considérer pour le partitionnement non contraint que la contrainte sur la balance de partitionnement est intégrée à la fonction objectif. Ainsi, contrairement au cas contraint, la plupart des problèmes omettent la contrainte sur la balance de partitionnement et n'énoncent que la fonction objectif à minimiser. La contrainte sur la balance de la partition ne sera donc pas incluse dans la définition du partitionnement non contraint, mais pourra être ajoutée par la suite :

- Soient un graphe G = (S, A) et un nombre de parties k. Soit l'espace des solutions E, défini par le problème général du partitionnement de graphe.

L'ensemble E_a des solutions admissibles du problème est défini par :

$$E_a = \{P_i \in E \text{ tel que } card(P_i) = k\}.$$

Le problème du partitionnement contraint consiste à trouver $\tilde{x} \in E_a$ qui minimise une fonction de coût f comme le coût normalisé ou le ratio de coupe :

$$f(\tilde{x}) = \min_{x \in E_a} f(x)$$

Comme l'ensemble des solutions admissibles du problème dans le cas du partitionnement contraint est bien plus petit que dans le cas non contraint, trouver une partition qui satisfasse ce dernier est plus facile que pour le premier. Dans ce dernier cas, un algorithme qui s'écarte de la balance requise aura beaucoup de mal à retrouver une partition la vérifiant. Il est donc difficile de commencer une recherche hors de l'espace des solutions admissibles dans le but d'y revenir par la suite. Ainsi, il peut sembler plus simple de trouver une bonne solution avec une balance faible qu'une balance forte. Cependant il n'en est rien, car s'il est plus facile de trouver une solution admissible dans le cas non contraint, c'est que l'espace admissible est plus grand, mais il n'est pas plus facile à explorer. De plus, un minimum local dans le cas contraint n'est pas forcément proche d'un minimum local du cas non contraint.

Dans une certaine mesure et dans le cas où une balance maximale est imposée, le partitionnement non contraint peut être vu comme un problème d'optimisation multi objectif consistant à minimiser une des fonctions objectifs et à minimiser la balance de partitionnement [SEL06]. Pour pallier les lacunes des fonctions objectifs classiques et intégrer d'autres objectifs pouvant être partiellement antagonistes, d'autres articles traitent du problème de partitionnement de graphe comme un problème multi objectif [SCH99].

Remarque : On peut transformer le problème du partitionnement de graphe non connexe en partitionnements de graphes connexes par le mécanisme suivant : dans le cas d'un graphe non connexe, une partition évidente de ce graphe est l'ensemble formé des différentes parties connexes du graphe. De plus, une telle partition a un coût de coupe nul, ce qui entraîne que les différentes fonctions objectifs classiques présentées plus haut prennent toutes une valeur nulle pour un tel graphe.

2.8 Généralités liées au partitionnement :

2.8.1 Différences entre partitionnement contraint et non contraint :

Nous avons présenté dans les sections précédentes deux problèmes : le partitionnement contraint et le partitionnement non contraint. Cependant, bien que ces deux problèmes aient une formulation assez proche, les algorithmes utilisés pour les résoudre sont en général bien différents, si ce n'est dans leurs principes, du moins

dans leurs implémentations et leurs buts. En effet, au risque de trop schématiser, ces deux problèmes diffèrent principalement dans leurs priorités :

– le partitionnement contraint a pour but premier de trouver des parties de poids identiques et comme second but de diminuer le coût de coupe ;

– le partitionnement non contraint a pour but de minimiser le coût de coupe dans la limite de parties dont la taille est inversement proportionnelle à ce coût de coupe. Le principe étant d'éviter un déséquilibre trop grand des parties, sauf dans le cas où un coût de coupe est vraiment extrêmement faible. Ainsi, le but premier du cas non contraint est de minimiser le coût de coupe, et son second but est d'équilibrer les parties.

L'inversion de l'ordre des priorités entre ces deux problèmes entraîne qu'un algorithme permettant de traiter un des deux problèmes va donner des résultats très médiocres dans le cas du second problème. C'est ce que l'on a pu constater dans [BIC06b, BIC07].

Il faut noter l'absence d'article dans la littérature abordant ces deux problèmes de front et les comparants exhaustivement. Cependant, les problèmes de partitionnement non contraint sont multiples et des efforts ont été réalisés par ***Inderjit S. Dhillon*** pour les synthétiser [DHI04b, DHI07].

2.8.2 Technique de la bissection récursive :

Algorithme 1 Algorithme récursif de bissection de graphe.

Procédure BissectionRec(G = (S, A), k)

$P_k = \{S_1, \ldots, S_k\}$ % Partition de G en k parties

Procédure ITÉRER(S', k', num)

si $k' > 1$ **alors**

$k'_1 \leftarrow \left\lfloor \frac{k'}{2} \right\rfloor$ % entier le plus grand inférieur à $\frac{k'}{2}$

$k'_2 \leftarrow k' - k'_1$

$S'_1, S'_2 \leftarrow bissection(S', \frac{k'_1}{k'}, \frac{k'_2}{k'})$

ITÉRER(S'_1, k'_1, num)

ITÉRER(S'_1, k'_1, $num + k'_1$)

sinon % Plus de bissection à effectuer, la partie S' est mise dans P_k

$S_{num} \leftarrow S'$

fin si

fin Procédure

Itérer(S, k, 1)

retourner P_k

fin Procédure

Beaucoup d'algorithmes de partitionnement ne s'intéressent qu'au cas de la bissection de graphes [RON05, MAR05, MAR06, CHA07]. Lorsqu'il est possible d'ajuster précisément la balance de partitionnement, la technique de la bissection récursive permet d'adapter ces algorithmes au k-partitionnement. Cependant, lorsqu'il n'est pas possible de régler la balance de partitionnement, la technique de la bissection récursive ne permet de faire que des 2^i-partitionnements ($i \in \mathbb{N}$).

L'algorithme 1 présente la méthode d'adaptation d'un algorithme, que l'on nommera *bissection*, au k-partitionnement. L'algorithme 1 utilise une fonction récursive, ITÉRER, prenant en paramètres la partie S' du graphe à partitionner, le cardinal k' de la partition de S', ainsi que le rang *num* de la première partie de la partition de S' dans la partition finale P_k. La valeur prise par k'_1 est l'entier le plus grand inférieur à k'_2, celle prise par k'_2 correspond au reste du nombre de parties à trouver. La fonction *bissection* doit trouver une bissection de S' en deux parties S'_1et S'_2 telles que :

$$poids(S'_1) \leq \frac{k'_1}{k'}\ poids(S') \text{ et } poids(S'_2) \leq \frac{k'_2}{k'} poids(S').$$

Pour que la partition finale respecte la balance de partitionnement, plusieurs solutions existent. La plus simple consiste à chercher à chaque itération une bissection du graphe en 2 parties de tailles égales. La partition trouvée sera alors parfaitement balancée, i.e. $bal(P_k) = 1{,}00$. Lorsqu'un algorithme d'affinage sera appliqué à la partition P_k, sa balance pourra alors évoluer. Une seconde solution consiste à être très permissive sur la balance de partitionnement dès le départ, puis à la durcir en fonction de la taille des parties déjà obtenues. Cette solution est plus compliquée car il faut après chaque bissection recalculer la balance maximale que

devront respecter les bissections de chacune des deux parties trouvées, afin qu'au final la partition respecte bien la balance demandée.

Malgré l'intérêt de pouvoir adapter les méthodes de bissection au *k*-partitionnement, il a été montré dans [SIM97] que l'algorithme récursif de bissection ne pouvait explorer certaines solutions du problème. Celles-ci ne peuvent être trouvées qu'au moyen de méthodes de *k*-partitionnement direct, d'où l'intérêt de ces dernières.

2.9 Complexité algorithmique :

Un algorithme est un ensemble d'étapes qui permettent de donner une réponse à un problème. Le mot algorithme vient du nom du mathématicien arabe Al-Khawarismi, connu comme le père de l'algèbre. Sa définition a été donnée bien le début de l'informatique par David Hilbert en 1928, lorsqu'il cherchait une suite d'instruction, finies et non-ambiguës, pour résoudre son «Entscheidungsproblem » (problème de décision). La non-ambiguïté des instructions permit ainsi de distinguer les algorithmes mathématiques, des algorithmes non mathématiques tels que les recettes de cuisine. La complexité algorithmique, plus communément appelée complexité, a pour but de donner une indication sur la durée d'exécution ou l'espace mémoire occupé par un algorithme.

La notion de complexité que nous présentons ci-dessous, est très liée au modèle de machine de calcul d'Alan Turing [TUR36]. Dans ce modèle une action est une suite d'opérations élémentaires. A chacune des étapes de cette suite, selon l'état de la mémoire de la machine, une opération (élémentaire) est choisie dans l'ensemble des opérations possibles. On dit que cette machine est déterministe lorsque, à chaque étape, le choix de l'opération dépend, et est déterminé, par l'état courant de la mémoire.

2.9.1 Algorithme et mesure de complexité :

La théorie de la complexité fournit des outils pour l'étude et l'évaluation des performances d'un algorithme, du point de vue des ressources (en temps et en espace) essentielles à son exécution. A l'aube de l'informatique (après la seconde guerre mondiale) l'évaluation intrinsèque de la performance était relativement "primitive" : les scientifiques comparaient des mesures qui dépendaient du matériel

(le hardware), du compilateur et du langage (le software) utilisés. Ces mesures étaient prises lors des exécutions des algorithmes et on mesurait le nombre d'opérations élémentaires. La définition d'une opération élémentaire dépend du détail que l'on souhaite avoir sur la complexité d'un algorithme. Nous considérons ici, une opération élémentaire comme une opération *simple,* d'addition, de soustraction, d'affectation de valeurs,... Le fait, par exemple, simplement de demander à la machine de faire l'addition de deux nombres *a* et *b* peut être décomposé en plusieurs opérations : (1) lire *a*; (2) lire *b*; (3) additionner *a et b* (4) afficher le résultat. L'évaluation de la complexité essaye de quantifier ces opérations car plus il y a d'opérations élémentaires pour une action, plus cette action va prendre du temps à être exécutée. Or, une même action peut se décomposer en plusieurs suites différentes d'opérations élémentaires. Pour multiplier *a* et *b*, une ancienne machine additionnerait *b* fois la valeur de *a* alors que sur les nouvelles machines, l'opération de multiplication est intégrée, donc la multiplication est réalisée en une seule opération. De la même manière, pour résoudre un même problème, il peut exister plusieurs algorithmes qui peuvent avoir des complexités différentes. Cette section est l'occasion de clarifier tout cela.

C'est en 1968 que ***Donald Knuth*** propose dans ***The Art of Computer Programming*** [KNU68], une méthode d'évaluation indépendante de l'environnement de développement et mesurant non plus les opérations élémentaires mais la quantité de *ressources* utilisées par un algorithme pour son fonctionnement. On entend par ressource, le temps nécessaire à l'algorithme pour se terminer ; ou encore, l'espace mémoire utilisé lors de son exécution. L'addition de *a* et *b* peut être considérée comme une seule étape.

Il propose aussi de mesurer la ressource (en temps ou en mémoire) consommée avec le scénario le plus défavorable : c'est ce qu'on appelle la mesure en pire cas. Par exemple, lorsqu'on cherche au hasard (de manière équiprobable) un nombre *m* parmi une liste de *n* valeurs différentes, la situation la plus défavorable est celle où *m* se trouve à la fin de la liste. L'algorithme consistant à parcourir et tester chaque élément de la liste pour trouver *m*, testera alors les *n* éléments. La complexité en pire cas (généralement appelé complexité) est donc *n*, bien qu'avec une autre liste, l'algorithme pourrait très bien rencontrer *m* en moins de *n* tests. Ainsi pour comparer

les algorithmiques, on peut comparer le nombre maximum d'instructions qu'ils nécessitent pour se réaliser. Il est facile de comprendre que ce genre de mesure ne permet pas réellement de comparer les performances des algorithmes dans la pratique. Par la suite il est donc proposé de mesurer la complexité en moyenne. : évaluer le nombre moyen d'instructions d'un algorithme. Reprenons l'exemple de la recherche du nombre *m* dans une liste de *n* nombres (*m* est parmi les éléments de la liste). De manière assez triviale, on peut calculer qu'en moyenne l'algorithme s'arrêtera au bout de *n* tests (modulo quelques hypothèses que nous ne précisons pas ici).

2.9.2 Évaluation et comparaison de complexité :

Considérons un problème quelconque *P*, appelons instance de *P* l'ensemble des données d'entrée et supposons que l'on ait un algorithme qui le résout. Nous souhaitons étudier sa complexité par rapport à la taille de l'instance de *P*. Dans le cas général, cette taille est notée *n*, et représente l'espace mémoire occupée par l'instance. La complexité est alors définie comme le nombre d'instructions exécutées pour résoudre une instance de *P* en fonction de *n*.

Prenons par exemple la somme $S_n = \sum_{i=1}^{n} i$ qu'il faut calculer ($S_1 = 1$; $S_2 = 1+2 = 3$; $Sn = 1 + \ldots + n$). L'approche la plus basique consiste à additionner successivement les *i*. Il faut alors faire *n* - 1 additions. Cela signifie que si on double la valeur de *n*, l'algorithme va prendre (à une unité près) deux fois plus de temps à s'exécuter. Or il existe une formule appelée l'identité de la somme des premiers entiers qui permet de calculer la somme : $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Avec cette identité, on peut écrire un nouvel algorithme nécessitant beaucoup moins de calcul (une addition, une multiplication et une division) quel que soit la valeur de *n*. Nous constatons ainsi que pour faire le même travail, la comparaison des deux algorithmes permet de choisir la plus rapide.

Notons cependant que cette manière d'évaluer la complexité, communément utilisée, est discutable car elle ne prend pas en compte le type de données traitées lors de chaque instruction.

Étant donnée la difficulté pratique de ces méthodes de mesures et de comparaisons, on préfère souvent exprimer la complexité en terme de *grandeur de dominance* : O(.). Ainsi un algorithme qui fait *n* - 1 opérations (où *n* est la taille de

l'instance) est en $O(n)$ (le terme *dominant* est n, *c'est-à-dire* que la complexité augmente linéairement à n) alors que le calcul de la série S_n avec l'identité de la somme des premiers entiers ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$) est en $O(1)$ (le terme dominant est une *constante, c'est-à-dire* que quelle que soit la valeur de n la complexité est constante).

On distingue ainsi la complexité des algorithmes par la grandeur qui la domine. Nous donnons ci-dessous une liste non exhaustive :

- O(1) : complexité constante, indépendante de n;
- $O(log(n))$: complexité logarithmique;
- $O(n^p)$: complexité polynomiale ($p \in \mathbb{R}, p > 1$) ;
- $O(a^n)$: complexité exponentielle ($a \in \mathbb{R}, a > 1$).

2.10 Classes P, NP, NP-complet et NP-difficile :

La classe *P* regroupe les problèmes dont la *résolution* peut se faire en temps polynomial. La classe *NP* regroupe les problèmes dont la vérification d'une solution accompagnée du *certificat* se déroule en temps polynomial.

Ainsi en terme de définition, il n'y a pas vraiment de relation entre *P* et *NP*. Cependant il est trivial de constater que les problèmes qui peuvent être résolu en temps polynomial (donc dans *P*), ont une méthode de validation (certificat) au plus en temps polynomial (donc dans *NP*). Cela montre la relation $P \subseteq NP$. La preuve de l'égalité (ou non) de cette relation est un sujet fondamental de l'informatique. Beaucoup de travaux supposent l'inégalité des deux ensembles. La démonstration de leur égalité serait un bouleversement, car cela induirait le fait que nous pouvons résoudre de manière exacte et en complexité polynomiale des problèmes pour lesquels nous ne connaissons actuellement, pour les résoudre, que des algorithmes de complexité exponentielle.

La classe *NP-complet* est un sous ensemble de la classe *NP*, regroupant les problèmes qui *dominent* tous les autres problèmes de la classe *NP*. Dans la pratique, cela veut dire que si on trouve un algorithme qui résout un problème de la classe *NP-complet* alors on pourra l'adapter pour résoudre tous les autres problèmes de la classe *NP-complet*. D'où l'intérêt de traiter les problèmes dits *NP-complet*.

Pour montrer qu'un problème *P est NP-complet* il faut montrer deux choses :

(1) une solution peut être *certifiée* en temps polynomial (le problème est alors dans

NP)

(2) tous les problèmes de la classe *NP* peuvent se réduire de manière polynomiale à *P* (on dit alors que le problème *P* est au moins aussi difficile que tous les autres problèmes de la classe *NP*). Les problèmes qui remplissent la dernière propriété sont de la classe *NP-difficile.* Les figures 2.2 et 2.3 résument ses imbrications de classes dans les cas où $P \neq NP$ *et* $P = NP$. [RAK13]

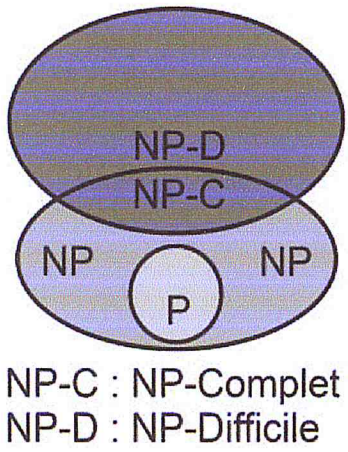


Figure 2.2. Les classes P, NP, NP-complet et NP $P \neq NP$ *[RAK13]*

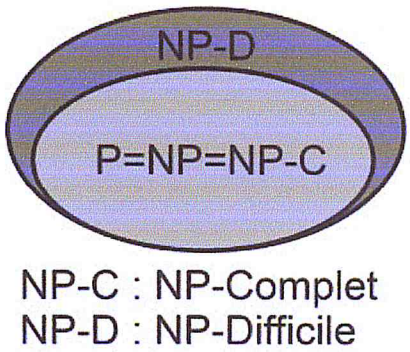


Figure 2.3. Les classes P, NP, NP-complet et NP $P = NP$ *[RAK13]*

Dans la pratique, il est toujours intéressant de démontrer qu'un problème est *NP- complet,* ou du moins *NV-difficile.* Montrer qu'un problème est dans NP *c'est-à-dire* que le certificat s'obtient en temps polynomial est souvent évident, mais réduire tous les problèmes *NP* au problème traité est beaucoup plus fastidieux. Habituellement on part d'un problème connu pour être *NP-complet,* que l'on déduit au problème traité.

Comme nous l'avons souligné, montrer que P = *NP* reviendrait à pouvoir résoudre de manière polynomiale tous les problèmes difficiles que nous connaissons actuellement dans le domaine de la recherche opérationnelle.

2.11 NP-difficulté des problèmes de partitionnement :

2.11.1 Le cas du partitionnement contraint :

Pour montrer qu'un problème d'optimisation est NP-difficile, il suffit de montrer que le problème de décision associé est NP-complet. Or, le problème de décision

associé au problème du partitionnement de graphe contraint peut se formaliser de la façon suivante :

- ***Problème de décision associé au problème du partitionnement de graphe contraint*** :

Instance du problème. Soit un graphe G = (S, A) pondéré sur ses sommets par ∀s ∈ S, *poids*(s) ∈ ℕ* et sur les arêtes par ∀a ∈ A, poids(a) ∈ N*. Soient $poids_{max} \in \mathbb{N}^*$ et $coupe_{max} \in \mathbb{N}^*$.

Problème. Existe-t-il une partition P_k de S en *k* parties disjointes $S_1,\ldots, S_k$ telle que pour toute partie S_i,

$$\sum_{s \in S_i} poids(s) \leq poids_{max}$$

et telle que :

$$coupe(P_k) \leq coupe_{max} ?$$

Solution. Ce problème reste NP-complet pour $poids_{max} \geq 3$, même lorsque le poids des sommets et des arêtes est unitaire. Il peut être résolu en temps polynomial quand $poids_{max} = 2$.

Ce problème de décision, appelé graph ***partitioning***, est traité dans [HYA73] et d'une manière qui fait référence dans [GAR79]. Ainsi, le problème du partitionnement de graphe contraint est NP-difficile.

2.11.2 Le cas du partitionnement non contraint :

Le but de cette sous-section est de montrer que ce problème est NP-difficile pour les fonctions objectifs de coupe normalisée et de ratio de coupe. Pour montrer qu'un problème d'optimisation est NP-difficile, nous allons montrer que le problème de décision associé, noté Π, est NP-complet. Pour montrer que Π est NP-complet, d'après [GAR79], il suffit de suivre les étapes suivantes :

1. montrer que Π est dans NP ;
2. trouver un problème Π′ NP-complet proche de ;
3. réduire Π′ à Π : trouver un sous-problème de Π′ équivalent à Π.

2.11.2.1 NP-difficulté de la coupe normalisée :

- ***Problème de décision associé au problème de partition :*** [SHI00]

Instance du problème. Soient un ensemble S fini et une fonction de poids pour tous les éléments s ∈ S : poids(s) ∈ ℕ*.

Problème. Existe-t-il un sous-ensemble $S' \subseteq S$ tel que $\sum_{s \in S'} poids(s) = \sum_{s \in S-S'} poids(s)$?

La démonstration de la NP-difficulté du problème de partitionnement utilisant la coupe normalisée repose sur la construction d'un graphe bien particulier. La propriété de ce graphe est qu'une bissection de ce graphe a une coupe normalisée suffisamment petite si et seulement si on peut trouver un sous-ensemble de *S* dont la somme des poids est égale à la moitié du poids de *S*. Le problème d'existence de la coupe normalisée se ramène alors à celui de l'existence de la partition, ce qui achève la démonstration.

2.11.2.2 NP-difficulté du ratio de coupe :

Le problème Π d'existence associé au ratio de coupe peut se formaliser de la façon suivante :

- ***Problème de décision associé au problème du ratio de coupe :***

Instance du problème. Soit un graphe G = (S, A) pondéré sur les sommets par ∀s ∈ S, poids(s) ∈ ℕ* et sur les arêtes par ∀a ∈ A, $poids(a)$ ∈ ℕ*. Soit $coupe_{max}$ ∈ ℕ*.

Problème. Existe-t-il une partition P_k de S en k sous-ensembles S_1, …, S_k disjoints, non vides, telle que $ratio(P_k)$ $coupe_{max}$?

Pour montrer que Π est dans NP, il suffit de montrer que la vérification d'une solution au problème Π s'effectue en un temps polynomial. Soit une solution P_k au problème Π. Vérifier que les ensembles $S_i \in P_k$ sont disjoints et inférieurs à un poids maximal se fait en $O(n_S)$. Le calcul du ratio de coupe se fait en parcourant les arêtes, i.e. en $O(n_A)$. Vérifier que chaque ensemble S_i est non vide est une opération en $O(k)$. Ainsi, vérifier qu'une partition est solution de Π se fait en temps polynomial.

Il existe plusieurs méthodes permettant de prouver la NP-complétude d'un problème d'optimisation [GAR79]. La méthode la plus commune consiste à

restreindre le problème de départ pour montrer qu'un cas particulier de celui-ci est un problème NP-complet.

Le problème de la coupe minimale dans des ensembles finis, intitulé ***minimum cut into bounded sets*** dans [GAR79], est proche du problème Π. Il s'énonce ainsi :

- **Problème de décision associé au problème de la coupe minimale :**

Instance du problème. Soit un graphe G = (S, A) pondéré sur les sommets par $\forall s \in S$, $poids(s) \in \mathbb{Z}^+$ et sur les arêtes par $\forall\ a \in A$, $poids(a) \in \mathbb{Z}^+$. Soient $s_1$et s_2 deux sommets différents de S. Soient $poids_{max} \leq poids(S)$ et $coupe_{max}$ deux entiers positifs.

Problème. Existe-t-il une bissection P_2 de S en deux sous-ensembles S_1, S_2 disjoints, vérifiant $s_1 \in S_1$, $s_2 \in S_2$, $poids(S_1) \leq poids_{max}$ et $poids(S_2) \leq poids_{max}$, telle que

$coupe(S_1, S_2) \leq coupe_{max}$?

Montrons maintenant que Π est NP-complet :

Les ensembles S_1 et S_2 sont non vides et disjoints. De plus, leur poids est borné. Supposons que

$$poids(S_1) \geq poids(S_2) :$$

Pour $k = 2$, le ratio de coupe devient :

$$ratio(S_1, S_2) = \frac{coupe(S_1,S_2)}{poids(S_1)} + \frac{coupe(S_2,S_1)}{poids(S_2)}$$

Puisque S_1 et S_2 son non vides, $poids(S_2) \geq 1$, puis

$$0 < \frac{1}{poids(S_1)} \leq \frac{1}{poids(S_2)} \leq 1,$$

et donc

$$ratio(S_1, S_2) \leq 2\frac{coupe(S_1,S_2)}{poids(S_2)}$$

$$\leq 2coupe(S_1, S_2)$$

$$\leq 2coupe_{max}.$$

Ainsi, Π est NP-complet.

Le problème d'optimisation du partitionnement de graphe non contraint pour le ratio de coupe est donc NP-difficile.

2.12 Conclusion :

Ce chapitre a commencé par rappeler quelques notions d'optimisation et de théorie des graphes. Puis, le problème général du partitionnement de graphe et plusieurs fonctions objectifs utilisables par ce problème ont été décrits. Comme ce problème se divise en deux sous-problèmes, le partitionnement « contraint » et le partitionnement « non contraint », ceux-ci sont présentés de manière formelle. La technique de la bissection récursive. Enfin, la NP-difficulté des différentes formes de problèmes de partitionnement a été étudiée.

En plus après avoir terminé cette partie l'algorithme de kernighan-lin nous semble le plus approprier à notre problème puisque nous voulons minimiser le cout de coupe, et puis c'est l'algorithme qui a le meilleur résultat pour le problème du voyageur de commerce et il est très utiliser par la communauté scientifique qui travaille dans ce domaine.

Chapitre 3

Solution proposée

3.1 Introduction :

Ce chapitre sera consacré à la présentation de l'heuristique que nous avons développée. Auparavant, nous donnerons une explication globale de l'algorithme de Kernighan-lin avec un exemple de partitionnement d'un graphe en bissection juste en utilisant l'algorithme d'origine. Ensuite, nous expliquerons les changements que nous avons apportés à cet algorithme afin de l'adapter à nos besoins. Les détails des structures de données et des formats des enregistrements des fichiers utilisés par notre application suivront. Enfin, nos procédures algorithmiques, commentées, seront décrites, puis nous terminerons ce chapitre par une conclusion qui sera une synthèse du contenu de ce chapitre.

3.2 Algorithme de partitionnement de Kernighan-Lin :

3.2.1 Présentation :

L'algorithme de Kernighan-Lin permet d'affiner la bissection d'un graphe. Il s'agit d'un algorithme d'optimisation locale. Le principe de l'algorithme est de trouver deux sous-ensembles de sommets de même taille, chacun dans une partie de la bissection pour trouver « une partition localement optimale ». A partir d'une bissection existante, l'algorithme échange successivement deux sous-ensembles de la bissection jusqu'à elle ne peut pas diminuer le coût de coupe.

Soit un graphe G = (S, A) avec une bissection initiale $P_2(G)$ $\{S_1, S_2\}$,

$S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$

Pour trouver facilement deux sous-ensembles à échanger, les auteurs introduisent deux notions: le coût intérieur et le coût extérieur.

Le coût intérieur I(s) d'un sommet « s » de la partie S_i de la bissection est la somme de poids des arêtes adjacent à s dont le second sommet est dans S_i :

$$I(s) = \sum_{s' \in S_i} poid(s, s')$$

A l'inverse, le coût extérieur E(s) d'un sommet s de la partie S_i est défini par la somme de poids des arêtes adjacent à « s » dont le second sommet n'est pas dans S_i :

$$E(s) = \sum_{s \in S - S_i} poid(s, s')$$

D(s) se définit comme la différence entre le coût extérieur et le coût intérieur du sommet « s » :

$$D(s)=E(s)-I(s), \qquad s \in S$$

Les valeurs D(s) sont réparties en deux ensembles D_1 et D_2 pour les deux parties. Si l'on échange deux sommets de deux parties $s_1 \in S_1$ et $s_2 \in S_2$ les nouvelles valeurs du coût extérieur E' sont :

$$E'(s_1) = I(s_1) + \text{poids}(s_1, s_2)$$

$$E'(s_2) = I(s_2) + \text{poids}(s_1, s_2)$$

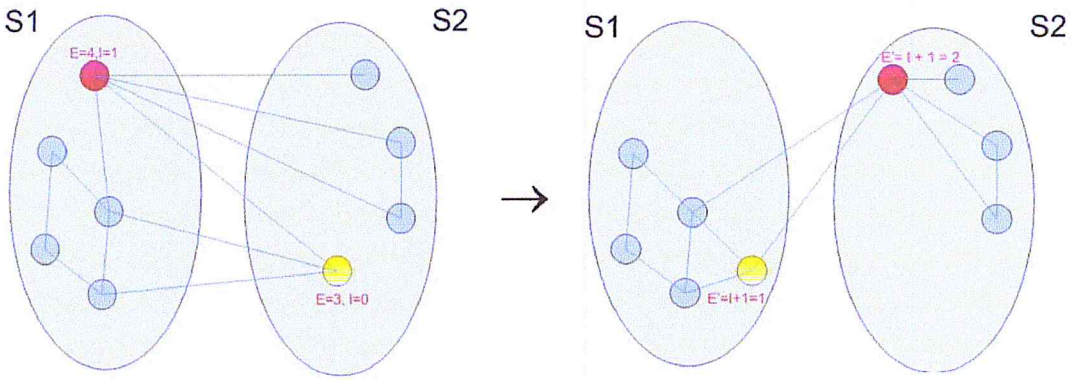


Figure 3.1 Exemple de l'échange de deux sommets [Kar98a]

Le gain de l'échange deux sommets de deux parties $s_1 \in S_1$ et $s_2 \in S_2$ est donc égale à:

$$\begin{aligned} \text{gain}(s_1, s_2) &= \text{coupe}(S_1, S_2) - \text{coupe}(S_1', S_2') \\ &= E(s_1) + E(s_2) - E'(s_1) - E'(s_2) \\ &= E(s_1) + E(s_2) - I(s_1) - \text{poids}(s_1, s_2) - I(s_2) - \text{poids}(s_1, s_2) \end{aligned}$$

Pour chaque sommet $s \in S$, la valeur D(s) est calculée. Soit l'algorithme à l'itération p, elle commence par chercher deux sommets $s_1^p \in S_1$ et $s_2^p \in S_2$ pour

que la valeur gain(s_1^p , s_2^p) soit maximal. Et puis, ces deux sommets sont sélectionnés et verrouillés, ce qui veut dire qu'ils ne pourront plus être sélectionnés dans les itérations après. Les valeurs D(s) sont recalculées pour les sommets non verrouillés.

$$D_1'(s) = D_1(s) + 2poids(s_1^p, s) - 2poids(s_2^p, s), s \in S_1 - \{s_1^p\}$$

$$D_2'(s) = D_2(s) + 2poids(s_2^p, s) - 2poids(s_1^p, s), s \in S_2 - \{s_2^p\}$$

L'algorithme est itéré avec la bissection ($S_1 \setminus s_1^p$, $S_2 \setminus s_2^p$) et des ensembles D_1' et D_2'. Elle se termine s'il n'y a plus de sommets à échanger.

Le gain résultant de l'échange de l'ensemble des sommets $\{ s_1^1, s_1^2, \dots , s_1^i\}$ et $\{ s_2^1, s_2^2, \dots, s_2^i\}$ est calculé

$$G(i) = \sum_{k=1}^{i} gain(s_1^k, s_2^k)$$

Si le gain obtenu est positif, alors les deux ensembles de sommets qui maximisent G sont échangés. Si non, l'algorithme s'arrête.

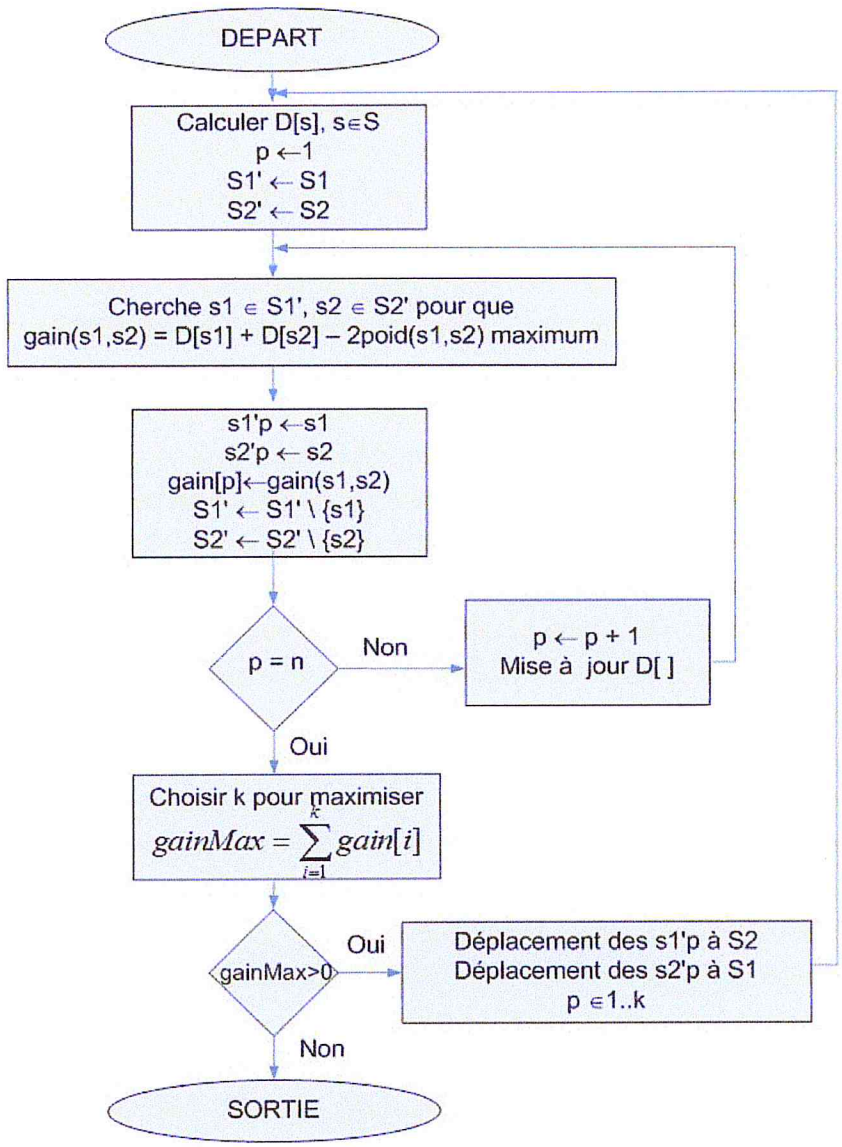


Figure 3.2 Algorithme de Kernighan-Lin [KAR70]

3.2.2 Exemple :

Prenons l'exemple du graphe pondéré G = (S, A) avec :

- S l'ensemble des sommets. **(|S| = n)**
- A l'ensemble des arêtes. **(|A| = a)**
- *Poids* (A,B) c'est le poids pour toutes arête (A , B) ∈ E.

À la fin de l'exécution de l'algorithme nous aurons deux partitions S_1 et S_2 tel que :

$$S = S_1 \cup S_2 \text{ et } S_1 \cap S_2 = \{\}.$$

Chaque partition contient **(n/2)** sommets et la somme des poids des arêtes entre les deux partitions est minimisée.

Comme nous l'avons déjà expliqué ce problème est NP-Complet. Pour quatre sommets (A, B, C, D) nous pouvons avoir trois possibilités :

1. $S_1 = \{A, B\}$ & $S_2 = \{C, D\}$
2. $S_1 = \{A, C\}$ & $S_2 = \{B, D\}$
3. $S_1 = \{A, D\}$ & $S_2 = \{B, C\}$

Prenons l'exemple du graphe pondéré G avec $S(G) = \{a, b, c, d, e, f\}$

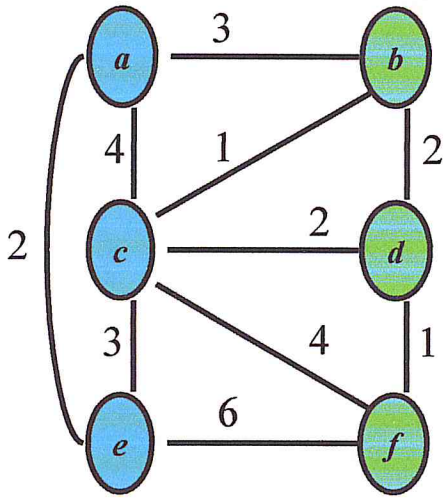


Figure 3.3 Exemple de graphe pondéré

Comme partition initiale nous avons :

$S_1 = \{a, c, e\}$

$S_2 = \{d, b, f\}$

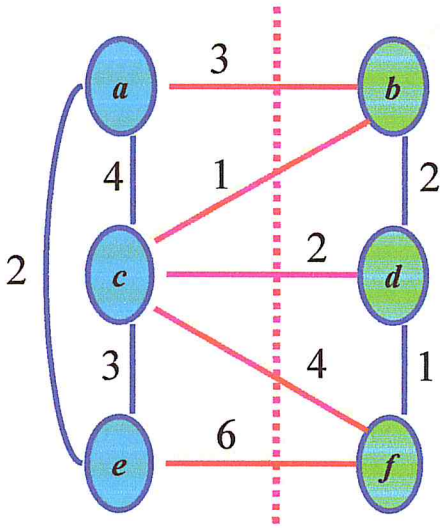


Figure 3.4 Partitionnement initial

L'objectif de l'algorithme est de minimiser la somme des poids des arêtes qui relient les deux partitions. On va appeler cette valeur le « coût ».

Donc le coût initial est égal à :

$$coût = 3 + 1 + 2 + 4 + 6 = 16$$

Nous commençons par calculer le gain $D(s)$ de déplacement de chaque sommet des deux partitions :

$$D(s) = E(s) - I(s)$$

$E(s)$ = somme des poids des arêtes qui raccorde le nœud « s » avec les nœuds de l'autre partition.

$I(s)$ = somme des poids des arêtes qui raccorde le nœud « s » avec les nœuds de sa partition.

$$D_a = E_a - I_a = -3 \quad (= 3 - 4 - 2)$$

$$D_c = E_c - I_c = 0 \quad (= 1 + 2 + 4 - 4 - 3)$$

$$D_e = E_e - I_e = +1 \quad (= 6 - 2 - 3)$$

$$D_b = E_b - I_b = +2 \quad (= 3 + 1 - 2)$$

$$D_d = E_d - I_d = -1 \quad (= 2 - 2 - 1)$$

$$D_f = E_f - I_f = +9 \ (= 4 + 6 - 1)$$

La valeur du gain obtenu en échangeant le sommet a avec le sommet b est égale à :

$$D_a + D_b$$

Toutefois, le gain de l'arête directe a été compté deux fois. Mais cette arête relie encore les deux groupes.

Par conséquent, le véritable «gain» de cet échange est :

$$G_{ab} = D_a + D_b - 2p_{ab}$$

$$D_a = E_a - I_a = -3 \ (= 3 - 4 - 2)$$

$$D_b = E_b - I_b = +2 \ (= 3 + 1 - 2)$$

$$G_{ab} = D_a + D_b - 2p_{ab} = -7 \ (= -3 + 2 - 2*3)$$

donc on a :

$D_a = -3$	$D_b = +2$
$D_c = 0$	$D_d = -1$
$D_e = +1$	$D_f = +9$

Calcul de tous les gains D :

$$G_{ab} = D_a + D_b - 2p_{ab} = -3 + 2 - 2\cdot 3 = -7$$

$$G_{ad} = D_a + D_d - 2p_{ad} = -3 - 1 - 2\cdot 0 = -4$$

$$G_{af} = D_a + D_f - 2p_{af} = -3 + 9 - 2\cdot 0 = +6$$

$$G_{cb} = D_c + D_b - 2p_{cb} = 0 + 2 - 2\cdot 1 = 0$$

$$G_{cd} = D_c + D_d - 2p_{cd} = 0 - 1 - 2\cdot 2 = -5$$

$$G_{cf} = D_c + D_f - 2p_{cf} = 0 + 9 - 2\cdot 4 = +1$$

$$G_{eb} = D_e + D_b - 2p_{eb} = +1 + 2 - 2\cdot 0 = +3$$

$$G_{ed} = D_e + D_d - 2p_{ed} = +1 - 1 - 2\cdot 0 = 0$$

$$G_{ef} = D_e + D_f - 2p_{ef} = +1 + 9 - 2\cdot 6 = -2$$

Paire avec le plus grand gain

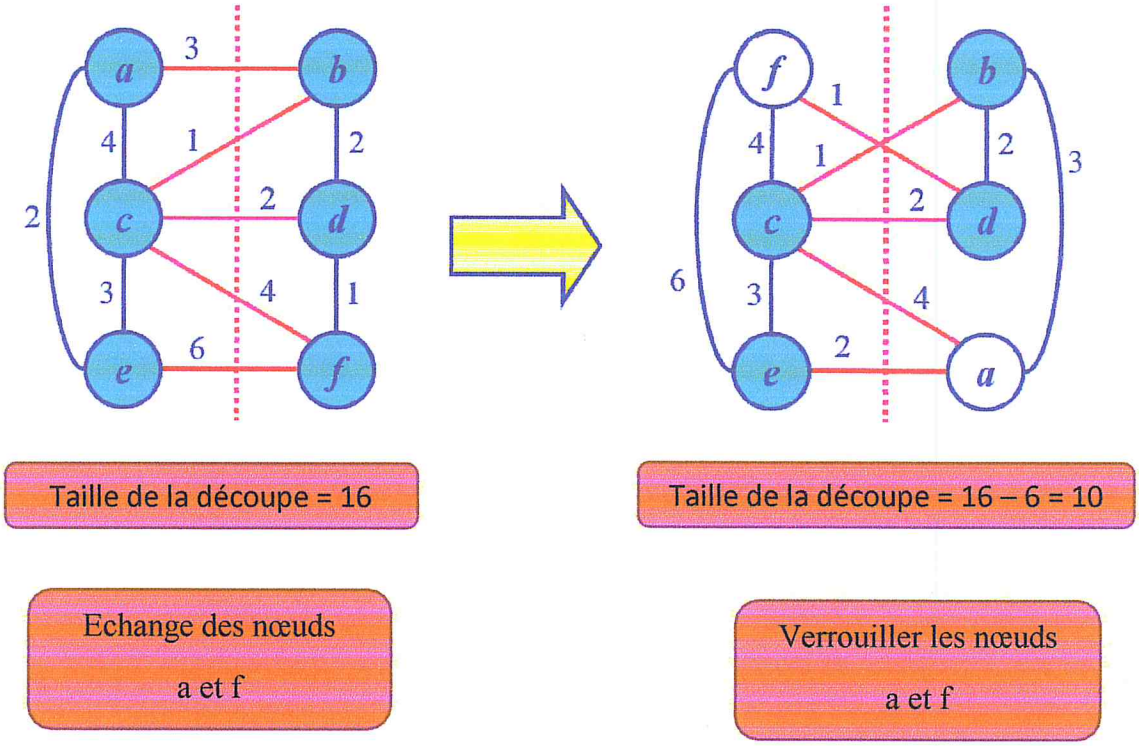


Figure 3.5 Exemple d'échange de noeuds

$$D_{af} = D_a + D_f - 2P_{af} = -3 + 9 - 2 \cdot 0 = +6$$

Ensuite on doit mettre à jour les valeurs D des nœuds déverrouillés :

$$D'_c = D_c + 2p_{ca} - 2p_{cf} = 0 + 2(4 - 4) = 0$$

$$D'_e = D_e + 2p_{ea} - 2p_{ef} = 1 + 2(2 - 6) = -7$$

$$D'_b = D_b + 2p_{bf} - 2p_{ba} = 2 + 2(0 - 3) = -4$$

$$D'_d = D_d + 2p_{df} - 2p_{da} = -1 + 2(1 - 0) = 1$$

et puis on doit recalculer les gains :

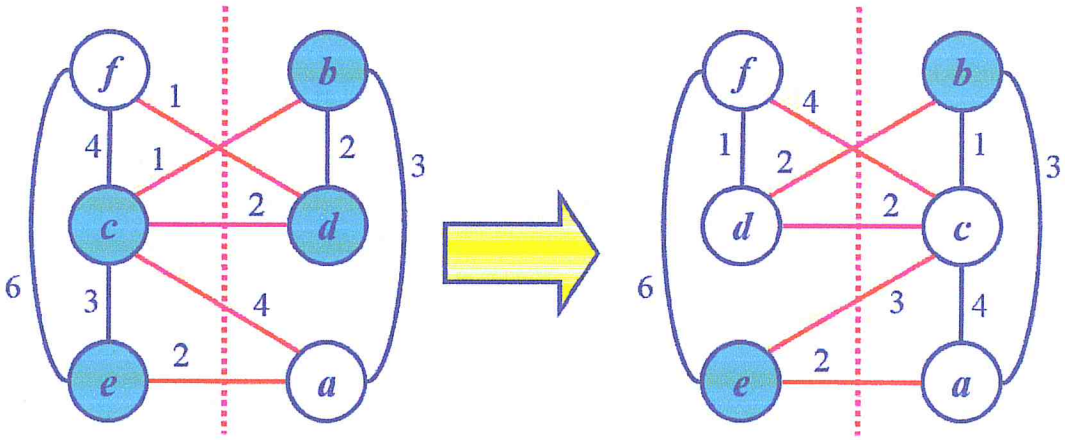
$$G'_{cb} = D'_c + D'_b - 2p_{cb} = 0 - 4 - 2 \cdot 1 = -6$$

$$G'_{cd} = D'_c + D'_d - 2p_{cd} = 0 + 1 - 2 \cdot 2 = -3$$

$$G'_{eb} = D'_e + D'_b - 2p_{eb} = -7 - 4 - 2 \cdot 0 = -11$$

$$G'_{ed} = D'_e + D'_d - pc_{ed} = -7 + 1 - 2 \cdot 0 = -6$$

La paire avec le maximum de gain peut aussi être négative



Taille de la découpe = 10

Taille de la découpe = 10 – (-3) = 13

Echange des nœuds c et d

Verrouiller les nouds c et d

Figure 2.6 Exemple de continuité du traitement

$$G'_{cd} = D'_c + D'_d - 2c_{cd} = 0 + 1 - 2 \cdot 2 = -3$$

Mise à jour des valeurs D des nœuds déverrouillés :

$$D''_e = D'_e + 2p_{ed} - 2p_{ec} = -7 + 2(0 - 3) = -1$$

$$D''_b = D'_b + 2p_{bd} - 2p_{bc} = -4 + 2(2 - 1) = -2$$

Calcul des gains:

$$G''_{eb} = D''_e + D''_b - 2p_{eb} = -1 - 2 - 2 \cdot 0 \quad = -3$$

La somme des gains :

- $G = +6$
- $G + G' = +6 - 3 = +3$
- $G + G' + G'' = +6 - 3 - 3 = 0$

A cette étape on doit choisir k telque :

Choisir k pour maximiser

$$gainMax = \sum_{i=1}^{k} gain[i]$$

Donc afin que le gain soit maximal on choisit k = 1. On obtient G = +6, et on permute seulement les deux sommets « a » et « f », La première itération est ainsi terminée, le gain est supérieur à zéro, et on doit donc relancer le programme avec les nouvelles partitions après avoir déverrouillé tous les nœuds.

On s'arrête lorsque le gain est inférieur ou égal à zéro.

3.2.3 Changement apporté

L'un des désavantages de cet algorithme est qu'il ne génère que deux partitions. Afin de palier à ce problème nous avons rendu l'algorithme récursif.

Pour cela nous avons introduit une liste chaînée de fichiers avec laquelle le programme commence, le graphe initiale étant au début de la liste de fichier.

A la fin de la première itération le programme supprime le graphe envoyé en argument et le remplace avec les deux partitions générées. Ensuite le programme cherche dans la liste de fichiers s'il existe un graphe avec plus de deux sommets. Si c'est le cas alors le programme se relance avec ce graphe en argument sinon le programme se termine. A chaque fois que le programme trouve dans la liste chainée de fichiers un graphe avec deux ou un sommet elle écrit les sommets dans un

fichier « res.gra » puis elle avance dans la liste les détails de l'exécution suivront dans le chapitre suivant.

Ainsi, cette méthode permettra d'obtenir l'ordre dans lequel les sommets doivent être disposés. Ceci revient à placer, côte à côte, les composants qui se communiquent très fortement, et ce, dans la mesure du possible puisque le problème est combinatoire.

Cette disposition de composants permet d'économiser de l'énergie et d'améliorer le temps de transfert des données. Un traitement supplémentaire serait de placer ces composants dans une zone de surface bien définie.

3.3 Format des fichiers d'enregistrement :

3.3.1 Les fichiers d'entrée :

Il y'a un seul fichier d'entrée décrivant les degrés de communications entre les composants du système VLSI.

3.3.1.1 Nom_fichier.gr :

Le fichier doit avoir l'extension « .gr » et il doit contenir des champs séparés par des espaces (il peut arbitrairement contenir des tabulations et des espaces, ceux-ci ne sont pas pris en compte). Un fichier valide contient dans la première ligne le nombre de sommets et dans les autres lignes la description des arcs sous la forme suivante :

1 5 25

Cette ligne décrit un arc sortant du sommet avec l'identifiant 1 et allant vers le sommet avec l'identifiant 5 portant le poids 25.

Si un sommet est isolé (aucun arc) alors il doit être décrit de la façon suivante :

2 -1 -1

Ici nous décrivons le sommet avec l'identifiant 2 et qui n'a pas de relation avec d'autres sommets. Nous avons choisi -1 car dans notre cas il est impossible pour un sommet d'avoir un identifiant négatif, ce qui est de même pour le poids d'un arc.

Exemple d'un graphe et sa description :

- Graphe :

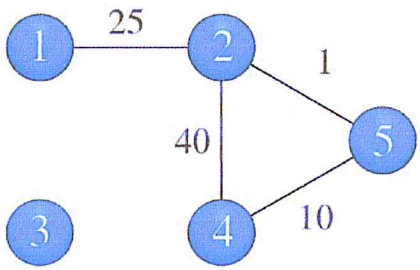


Figure 3.7 Exemple d'un graphe pondéré

- Description :

```
5       // nombre de sommets

1       2       25
2       1       25
2       4       40
4       2       40
2       5       1
5       2       1
4       5       10
5       4       10
3       -1      -1
```

3.3.2 Les fichiers de sortie :

On a un seul fichier de sortie et qui décrit l'ordre optimal ou pré-optimal du placement des IPs.

3.3.2.1 Nom_fichier.res :

Le fichier porte le nom du fichier en entrée mais avec l'extension « .res ». Il comporte tous les sommets du graphe en entrée, ordonnées de gauche à droite et séparés par des espaces.

Exemple de résultat pour le graphe décrit dans la figure 1.

4 2 3 5 1

3.4 Structures de données :

Les structures de données qui ont permis d'implémenter nos algorithmes sont les suivantes :

3.4.1 GRAPHE :

Un moyen commode de représenter un graphe est de considérer des listes d'adjacence. Cette structure de données est particulièrement bien adaptée aux graphes peu denses c'est-à-dire comportant peu d'arcs par sommet. En effet, si un graphe comporte de nombreux arcs par sommet, il est plus efficace de coder le graphe par une matrice (on évite alors la manipulation plus lourde des listes chainées).

Une représentation par liste d'adjacence est une liste chainée de structures de liste d'adjacence. Ces dernières sont formées de deux champs : un sommet et une liste chainée des sommets adjacents à celui-ci. Trois structures sont utilisées pour coder un graphe : ELTADJ, SOMMET et GRAPHE.

La figure 3.8 reprend schématiquement un graphe et son codage.

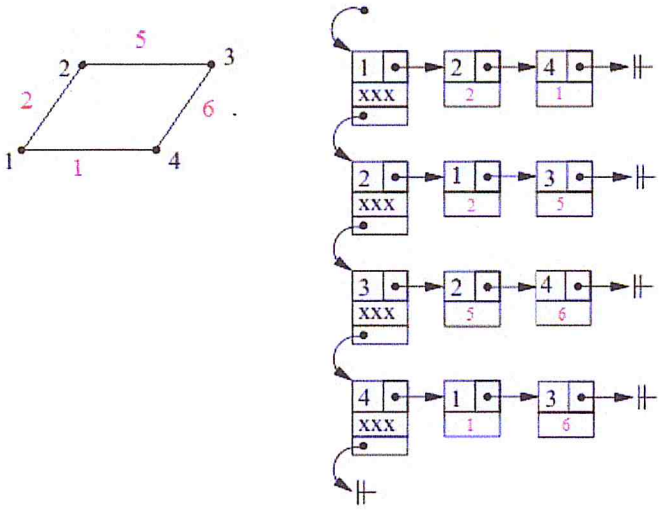


Figure 3.8 Exemple de représentation d'un graphe pondéré

3.4.2 SOMMET :

Permet de définir une liste chainée destinée à coder les sommets du graphe. Chaque sommet possède un label (un indice) ainsi qu'un Lock qui lui est propre (permettant de retenir si le sommet a déjà été visité) et un entier D qui est égal à :

$$D_a = E_a - I_a$$

E_a est le coût externe du sommet 'a' (la somme des poids des arcs entre le sommet 'a' et les sommets dans le graphe 'B')

I_a est le cout interne du sommet 'a' (la somme des poids des arcs entre le sommet 'a' et les sommets dans le graphe 'A')

Les deux derniers champs de la structure sont plus spécifiques : suivant pointe simplement vers l'élément suivant de la liste chainée. Enfin, adj est un pointeur vers une variable de type eltadj. Cela permet donc de lier un sommet à la liste des sommets qui lui sont adjacents. La figure 3.9 reprend schématiquement une liste chainée de sommets.

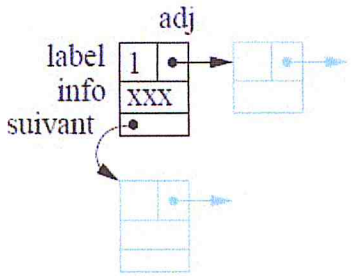


Figure 3.9 Liste chaînée de sommets

3.4.3 ELTADJ :

Permet de lister les sommets adjacents à un sommet donné. Autrement dit, elle sert à construire une liste d'adjacence des arcs issus d'un sommet donné. Le champ « dest » contient l'extrémité de l'arc, et le champ « face » est égal à 1 si la « dest » appartient au même sous graphe sinon est égal à 0 (on prend 1 et 0 comme « char » afin d'optimiser la consommation de mémoire car « char » nécessite 8 bits tandis qu'un entier « int » nécessite 32 bits) ; « info » contient le poids de l'arc codé et enfin, suivant est un pointeur vers l'élément suivant de la liste chainée. La figure 3.10 reprend schématiquement une liste d'adjacence.

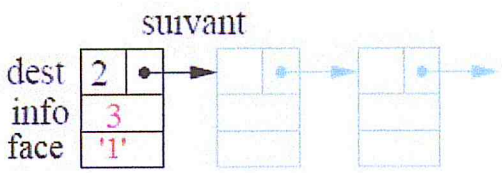


Figure 3.10 Exemple de liste d'adjacence de sommets

3.4.4 DELTA :

Liste chainée qui permet de sauvegarder les sommets qui maximisent le gain ainsi que leurs précédents. Elle comporte un entier « val » qui stocke la valeur du

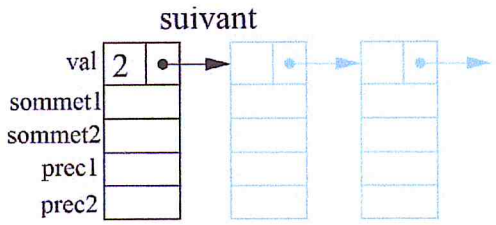


Figure 3.11 Représentation de la structure DELTA

gain pour décider de faire un échange de sommets entre les deux partitions courantes. Sont aussi utilisés quatre pointeurs « sommet1 », « sommet2 », « prec1 », « prec2 » qui pointent sur des variables de type sommet, et un pointeur « suivant » qui pointe sur l'élément suivant.

3.4.5 FILES :

Liste chainée de fichiers qui comporte un champ « fichier » qui est un pointeur sur un élément de type FILE, et un pointeur « suivant » qui pointe sur l'élément

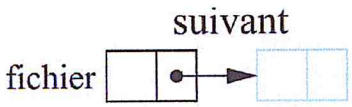


Figure 3.12 Représentation de la liste FILES

suivant.

3.5 Algorithmes :

3.5.1 Algorithme pour la fonction initialiser Graphe :

On transmet à cette fonction l'adresse d'une variable de type GRAPHE. La variable correspondante sera initialisée au graphe vide.

3.5.2 Algorithme pour la fonction ajouterSommet :

On transmet à cette fonction l'adresse d'une variable de type GRAPHE et un entier. Le champ maxS est incrémenté d'une unité, un sommet de label maxS (après

incrémentation) est ajouté au graphe en fin de liste. Si l'allocation de mémoire n'est pas possible il retourne -1.

On utilise cette fonction pour générer des graphes aléatoires.

procedure ajouterSommet (G = (S, A))

Début

Si (nombre de sommets = 0)

Alors Ajouter le sommet au début de la liste

Sinon

maxS ← maxS+1

Ajouter le sommet avec le label maxS à la fin de la liste

fsi

Fin

3.5.3 Algorithme pour la fonction ajouterSommet 2 :

On transmet à cette fonction l'adresse d'une variable de type GRAPHE et un entier. Le champ maxS est incrémenté d'une unité, un sommet de label égal à l'entier en argument est ajouté au graphe. Si l'allocation de mémoire n'est pas possible il retourne -1.

procedure ajouterSommet2(G = (S, A) :Graphe, label :entier)

Début

Si (nombre de sommets = 0)

AlorsAjouter le sommet au début de la liste

Sinon **Si** (le sommet n'existe pas dans le graphe)

Alors Ajouter le sommet à la fin de la liste

Sinon Ne rien faire

Fsi

Fsi

nombreSommet ← nombreSommet + 1

Fin

3.5.4 Algorithme pour la fonction ajouterArc :

Crée un nouvel arc du sommet de label a vers le sommet de label b avec l'information supplémentaire donnée par info. Les arcs sont rangés dans la liste d'adjacence par indice croissant. Si un arc entre a et b existe déjà, le champ info est mis à jour sans créer de nouvel arc dans la liste d'adjacence. Si l'un des deux sommets ou les deux n'existent pas dans le graphe alors la procédure créé ce qui manque et elle l'ajoute au graphe.

procedure ajouterArc (G = (S, A): Graphe, *a*: entier, *b* : entier, info: entier)

Debut

Si (*a* n'existe pas) **alors** ajouteSommet (graphe, *a*)

Fsi

Si (*b* n'existe pas) **alors** ajouteSommet (graphe, *b*)

Fsi

Si arc_{ab} existe **alors** mise à jour de l'info

Sinon ajouter arc dans liste d'adjacence de *a* et *b*

Nombre d'arc ← Nombre d'arc + 1

Fsi

Fin

3.5.5 Algorithme pour la fonction lireFichier :

On fournit à cette fonction un pointeur sur un fichier de données et un pointeur vers une variable de type GRAPHE. Elle crée une structure de graphe à partir du fichier texte.

procedure lireFichier (G = (S, A) : Graphe, file : FICHIER)

Début

Si nonVide (fichier)

Alors

nombreSommet ← première ligne

pour chaque ligne

faire

a ← colone1

Si nombreColone = 1

Alors ajouterSommet2 (graphe, *a*)

Sinon

b ← colone2

poid ← colone3

ajouterArc(graphe, *a*, *b*, poid)

Fsi

Fin Faire

Fin

3.5.6 Algorithme pour la fonction écrireFichier :

Cette fonction fait l'inverse de la fonction lireFichier puisque celle-ci génère un fichier texte avec l'extension « .gr » et un nom donné en argument qui décrit une structure d'un graphe donné aussi en argument.

procedure ecrireFichier (G = (S, A): Graphe, file: FICHIER)

Debut

premiereLigne ← nombreSommet

pour tout sommet dans le graphe

faire pour chaque élément adjacent

faire

colone1 ← label do sommet

colone2 ← dest de l'élément adjacent

colone3 ← info de l'élément adjacent

```
            Fin pour
        Fin pour
    Fin
```

3.5.7 Algorithme pour la fonction supprimerSommet :

Supprime un sommet et les arcs correspondants (i.e., les arcs ayant leur origine ou leur extrémité correspondant à label). Cette fonction libère la mémoire correspondante.

```
procedure supprimerSommet (G = (S, A): Graphe, sommet: entier)
Debut
    Si (trouver (sommet)) alors
    PrécédantSommet ← (sommet → suivant)
    Liberer (sommet)
        pour tout sommet dans le graphe
            faire pour chaque élément adjacent
                faire Si (dest de l'élément adjacent = label do sommet)
                        supprimer (élémentAdjacent)
                Fin pour
            Fin pour
        nombreSommet ← nombreSommet - 1
Fin
```

3.5.8 Algorithme pour la fonction supprimerArc :

Supprime l'arc joignant a à b. Cette fonction libère la mémoire correspondante.

```
procedure supprimerArc (G = (S, A): Graphe, a: entier, b: entier)
Debut
    Si nonVide (graphe) alors
```

```
            Si (Trouver(a)) alors
                Si (trouverÉlémentAdjacent(b)) alors
                    LibererÉlémentAdjacent(b)
                    nombreArc ← nombreArc – 1
                Fsi
            Fsi
        Fsi
Fin
```

3.5.9 Algorithme pour la fonction cout :

On fournit à cette fonction un pointeur sur un des deux sous graphes et elle calcule la somme des poids des arcs entre les deux sous graphes

```
Fonction  cout (G = (S, A): Graphe)
Debut
Cout ← 0
        pour tout sommet dans le graphe
            faire pour chaque élément adjacent
                faire
                    Si (face = '0') alors
                    Cout ← Cout + info
                    Fsi
                Fin pour
            Fin pour
        Retourner Cout
Fin
```

3.5.10 Algorithme pour la fonction écrireFichier2 :

On fournit à cette fonction un pointeur sur un fichier (le fichier résultat) et un pointeur sur un graphe. Elle ajoute à la fin du fichier tous les sommets du graphe il fait appel à cette fonction à chaque fois qu'on trouve un graphe avec un nombre de sommets inferieur ou égal à 2.

Debut

pour tout sommet dans le graphe

faire ajouterAuFichier (label do sommet)

Fin pour

Fin

3.5.11 Algorithme pour la fonction supprimeGraphe :

La fonction libère l'ensemble de la mémoire utilisée par un graphe et rend simplement un graphe "vide".

fonction supprimeGraphe (G = (S, A): Graphe)

Debut

pour tout sommet dans le graphe

faire

pour chaque élément adjacent

faire

Libérer (élément_adjacent)

Fin pour

Libérer (sommet)

Fin pour

initialiserGraphe (G = (S, A): Graphe)

Fin

3.5.12 Algorithme pour la fonction afficherGraphe :

La fonction afficherGraphe affiche tout d'abord plusieurs informations de base sur le graphe d'adresse *g (nombre de sommets, nombre d'arêtes, . . .). Ensuite la liste chainée des sommets est parcourue et pour chacun de ces sommets, on parcourt l'entièreté de la liste d'adjacence correspondante.

```
fonction  afficherGraphe (G = (S, A): Graphe)
Debut
        pour tout sommet dans le graphe
          faire
            afficher (« sommet de label : »)
            afficher (label de sommet)
            retourChariot
            afficher (« adjacent avec : »)
              pour chaque élément adjacent
              faire
                    afficher (dest de élément adjacent)
              Fin pour
            retourChariot
          Fin pour
Fin
```

3.5.13 Algorithme pour la fonction classement :

On envoie à cette fonction un pointeur sur un graphe et l'identifiant d'un sommet et elle nous renvoie sa position dans la liste chainée qu'on utilise pour faire le partitionnement initial du graphe, plus précisément pour remplir le champ « face » des éléments adjacent. Si le classement du label de l'élément adjacent est inférieur à la moitié du nombre de sommets alors « face » = 1 pour la première partition ou = 0 pour la deuxième. Sinon, c'est l'inverse: elle renvoie -1 si elle ne trouve pas le sommet en question dans le graphe.

fonction classement (G = (S, A): Graphe, ArgSommet: entier)

Debut

Classement ← -1

pour tout sommet dans le graphe

faire

Classement ← Classement + 1

Si (labelSommet = ArgSommet) **alors**

sortieDeBoucle

Fsi

Fin pour

Retourner Classement

Fin

3.5.14 Algorithme pour la fonction initialiserSubGraphe :

On envoie à cette fonction 3 pointeurs : un sur le graphe initial et deux sur les sous-graphes. Elle fait appel à la fonction classement pour récupérer l'ordre de chaque élément des listes d'adjacence afin de déterminer la valeur de la variable « face » (si le nœud adjacent appartient au même sous-graphe que le sommet alors son champ « face » est fixée à 1, sinon à 0)

fonction initialiserSubGraphe (G = (S, A): Graphe, $sg_1 = (S_1, A_1)$: Graphe,

$sg_2 = (S_2, A_2)$: Graphe)

Debut

initialiserGraphe (sg1)

initialiserGraphe (sg2)

nombreSommet ← 0

ordre ← 0

classement ← 0

Si (nonVide (graphe G)) **alors**

pour tout « sommet » dans le graphe G

faire

```
nombreSommet ← nombreSommet + 1
Si (nombreSommet ⇐ nombreSommet_de_G/2) alors
    ajouterSommet2 (sg1, « sommet »)
    pour chaque « élément_adjacent » de « sommet »
        faire
            classement ← classment (G, «élément_adjacent»)
        Si (classement ⇐ nombreSommet_de_G/2)
        Alors
            Face ← '1'
        Sinon
            Face ← '0'
        Fsi
    Fin pour

Sinon
ajouterSommet2 (sg2, « sommet »)
    pour chaque « élément_adjacent » de « sommet »
        faire
            classement ← classment (G, «élément_adjacent»)
        Si (classement ⇐ nombreSommet_de_G/2)
        Alors
            Face ← '0'
        Sinon
            Face ← '1'
        Fsi
    Fin pour
Fsi
Fin pour
```

supprimerGraphe (G)

Fin

3.5.15 Algorithme pour la fonction moyennePoid :

On envoie à cette fonction un pointeur sur un graphe. Elle calcule la moyenne des poids et elle stocke la valeur dans le champ du graphe « moyP ».

fonction moyennePoid (G = (S, A): Graphe)

Debut

Somme ← 0

Moyenne ← 0

pour tout « sommet » dans le graphe

faire

pour chaque «élément adjacent »

faire

somme ← somme + poids

Fin pour

Fin pour

Moyenne ← somme / nombreArc

Retourner Moyenne

Fin

3.5.16 Algorithme pour la fonction pointer :

On envoie à cette fonction un pointeur sur un graphe et un identifiant d'un sommet et un pointeur. Elle récupère l'adresse des sommets et elle le stocke dans le pointeur. Elle retourne -1 si le sommet en question n'existe pas dans le graphe, 0 si le sommet existe.

fonction pointer (G = (S, A): Graphe, label: entier, pointeur: *SOMMET)

Debut

Debut

pour tout sommet dans le graphe

faire

pour chaque élément adjacent

faire Si (« face »=0) **alors**

E ← E + « poids »

Sinon

I ← I + « poids »

Fin pour

D_sommet ← E - I

Fin pour

Fin

3.5.19 Algorithme pour la fonction calculeD :

On fournit à cette fonction un pointeur sur un des deux sous graphes et elle calcule pour chaque sommet dans ce sous graphe la valeur D qui a déjà été définie. Elle la stocke alors dans la stocke dans le champ D du sommet.

3.5.20 Algorithme pour la fonction maxDelta :

On fournit à cette fonction les adresses des deux sous graphes et un pointeur sur la tête de la liste des deltas. Elle nous donne comme résultat la liste de tous les sommets qui maximisent delta deux à deux et cette valeur de delta.

Procédure maxDelta (sg1: Graphe, sg2: Graphe, delta :*DELTA)

Debut

Pour $i \leftarrow 1$ **jusqu'à** (i = nombreSommet_sg1)

Trouver deux sommets déverrouillés $a \in$ sg1, $b \in$ sg2 tel que

$G \leftarrow D_a + D_b - 2 *$ poids (a, b) soit maximal

Verrouiller (a, b)

Pour tout sommet « x » $\in$ sg1 non verrouillé

pour tout « sommet » dans le graphe

faire

Si (label = « sommet ») **alors**

Pointeur ← adress (« sommet »)

sortirDeBoucle

Fsi

Fin pour

Fin

3.5.17 Algorithme pour la fonction poidDe :

On envoie à cette fonction un pointeur sur un graphe et deux identifiants de sommet. Elle nous renvoie -1 en cas d'erreur sinon le poids de l'arête les reliant.

procedure poidDe (G = (S, A): Graphe, *a*: entier, *b*: entier)

Debut

Si nonVide (graphe) **alors**

Si (Trouver(*a*)) **alors**

Si (trouverÉlémentAdjacent(*b*)) **alors**

Retourner poid

Fsi

Fsi

Fsi

Fin

3.5.18 Algorithme pour la fonction random_number :

On envoie à cette fonction deux entiers qui représentent les limites de l'intervalle (min, max). Elle renvoie un entier appartenant à cet intervalle.

fonction calculeD (G = (S, A): Graphe)

E ← 0

I ← 0

3.5.22 Algorithme pour la fonction échange :

On envoie à cette fonction l'adresse du deux sous-graphes, la liste des Delta et les valeurs des deux variables et G, K. Elle se charge de permuter les sommets qui maximisent le gain.

fonction échange (sg1: GRAPHE, sg2: GRAPHE, Delta: DELTA, G: entier, K: entier)

Debut

Pour *i* ← 1 **jusqu'à** K

Intervertir (Delta[*i*] [sommet1], Delta[*i*] [sommet2])

Fin Pour

Déverrouiller Tous les Sommets

Fin

3.6 Conclusion :

Comme il existe un algorithme de partitionnement efficace et généralement utilisé par la communauté scientifique travaillant dans le domaine, nous l'avons adapté à notre problématique. Il s'agit de l'algorithme de partitionnement de Kernighan-Lin qui, étant donné un graphe muni de poids sur chaque arc, le partitionne efficacement (ce problème n'est pas polynomial) en deux sous-graphes de sorte à minimiser la somme des poids associés à des arcs dont les extrémités ne se trouvent pas dans le même sous-graphe.

Ainsi, nous faisons appel à cet algorithme plusieurs fois jusqu'à ce que dans chaque sous-graphe il ne subsiste que deux nœuds. Le placement des composants du système se fera donc de la manière suivante :

- placer côte à côte deux composants dont les nœuds associés se trouvent dans le même sous-graphe.
- deux paires de composants seront placées côte à côte si les quatre nœuds associés ont été générés, à une étape donnée, dans le même sous-graphe.

Comme il a été expliqué précédemment, il est évident qu'un tel placement va engendrer des valeurs intéressantes de temps et de consommation de puissance dus aux différentes interconnexions.

Ce chapitre a permis de montrer les détails techniques de notre méthode (algorithmes, structures de données et formats des enregistrements des fichiers utilisés). Nous avons notamment utilisé la récursivité pour des fins d'efficacité. Néanmoins, ceci nous a demandé de prendre un certain nombre de précautions afin d'éviter que l'exécution ne boucle indéfiniment, ou qu'elle se déroule autrement que le demande la logique de l'application.

Chapitre 4

Tests et résultats

4.1 Introduction :

Les résultats qui vont être présentés dans ce chapitre ont été obtenus sur une machine dotée d'une mémoire principale de capacité de 4 Go et d'un processeur fonctionnant à 2.20 GHz s'exécutant sous le système d'exploitation Linux.

Le code source de notre application a été développé en utilisant le langage C. Dans un souci de conception modulaire, notre application a été décomposée en plusieurs modules dont la compilation a été rendue efficace et aisée grâce à l'utilisation du ***Makefile***, fichier indiquant les dépendances entre les différents fichiers ainsi que la (les) commande(s) de compilation (ou d'édition de liens) à exécuter en cas de modification d'un fichier source ou objet.

4.2 Présentation des résultats :

Nous allons commencer par donner le résultat final de l'exemple présenté dans le précédent chapitre avec un peu d'explication. Suivront alors les résultats de graphes générés aléatoirement pour montrer l'efficacité de notre technique en termes de temps CPU ainsi que l'impact de la solution générée sur la dissipation de la puissance. Enfin, nous terminerons par une conclusion.

4.2.1 Suite de l'exemple :

Donnons tout d'abord la description du graphe :

Data.gr

6 //nombre de sommets
1 2 25
2 1 25
2 4 40
4 2 40
2 5 1
5 2 1
4 3 5
3 4 5
5 3 8

3 5 8

3 6 3

6 3 3

Il y a six sommets et le programme fait des divisions de deux. Dans la première itération, il devrait donc générer deux partitions de trois sommets chacune. Considérons le résultat:

Première partition :

3

2 3 3

2 6 4

3 2 3

3 6 6

6 2 4

6 3 6

Deuxième partition :

3

5 4 2

4 1 3

4 5 2

1 4 3

Pour la deuxième itération le programme teste la taille des deux partitions générées. Si l'une des deux est supérieure à deux alors le programme se relance avec cette partition. Sinon, il enregistre les sommets dans le fichier «res.gra».

Dans notre cas il va se relancer pour les deux et générer les fichiers suivants :

Partitionnement de la première partition :

Partition 1 :

1

2 0 0

Partition 2 :

2

6 3 6

3 6 6

Partitionnement de la deuxième partition :

Partition 1 :

1

5 0 0

Partition 2 :

2

4 1 3

1 4 3

Comme nous pouvons le remarquer aucun cardinal d'une partition générée ne dépasse pas deux. L'exécution s'arrête donc après avoir enregistré les sommets dans

Fichier res.gra :

2 6 3 5 4 1

le fichier « res.gra » suivant l'ordre de l'exécution.

```
cout = 16
nouveau cout = 10
cout = 7
nouveau cout = 7
cout = 2
nouveau cout = 2
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

Figure 4.1 Capture d'écran lors de l'exécution

4.2.2 Estimation de la puissance :

Grace à un programme conçu par le Dr.Ali Mahdoum nous avons pu estimer la consommation de la puissance pour des transferts de données correspondant à des graphes pondérés générés aléatoirement. Les résultats obtenus sont indiqués dans le tableau suivant:

Nombre de nœuds	Puissance estimée (µW)	Temps CPU (S)
20	0.42	0.125
25	0.44	0.148
30	0.51	0.172
50	1.2	0.294
100	1.7	0.763
200	3.4	3.455
250	4.3	6.588
300	5.7	12.091
400	7.8	26.000
500	9.6	97 (1 mn 37 s)
1000	20.4	1499 (24 mn 59 s)

Table 4.1 Résultats obtenus

Conclusion générale

Conclusion générale:

Du fait que le délai et la consommation de puissance dans les systèmes VLSI actuels sont largement dominés par les interconnexions, un aspect important doit être donné à ces dernières. Les architectures à base de bus classiques ou de barres croisées (cross-bars) n'étant plus efficaces pour les systèmes actuels qui sont très complexes et se caractérisant par un fort degré de communications, il est nécessaire de développer un réseau d'interconnexions spécifique. A cet effet, un projet a été défini au CDTA où une nouvelle méthodologie de conception de réseaux sur puce a été proposée. Dans le cadre de cette méthodologie, nous avons développé une heuristique de placement afin de pouvoir faire face à un nombre quelconque de composants (IPs) en générant une solution intéressante en un temps CPU appréciable.

La concrétisation de ce projet nous a permis de nous familiariser avec quelques aspects de la conception de circuits et de systèmes VLSI, des notions de théorie des graphes et problèmes combinatoires tout en mettant en pratique certaines de nos connaissances acquises dans notre cursus universitaire.

En perspective à ce travail, nous prévoyons d'abord de le comparer à d'autres méthodes afin de d'évaluer son efficacité. Nous comptons aussi développer une interface graphique qui affiche la disposition des composants du système.

Bibliographie

Bibliographie :

[ALP95a] : Charles J. ALPERT et Andrew B. K AHNG : Recent directions in netlist partitioning : A survey. Integration, the VLSI Journal, 19(12):1–81, 1995.

[BAR93] : Stephen T. Barnard et Horst D. Simon.A fast multilevel implementation of recursive spectral bisection for patitioning unstructured problems. s.l. : Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, 1993.

[BIC07] : Charles-Edmond BICHOT : A new Method, the Fusion Fission, for the relaxed k-way graph partitioning problem, and comparisons with some Multilevel algorithms. Journal of Mathematical Modeling and Algorithms (JMMA), 6(3):319–344, 2007.

[BIC06b] : Charles-Edmond BICHOT : Metaheuristics versus spectral and multilevel methods applied on an Air Traffic Control problem. In Proceedings of the 12th IFAC Symposium on Information Control Problems in Manufacturing (INCOM), pages 493–498, may 2006.

[BUI93] : T.Bui et C.Jones.A heuristic for reducing fill in sparse matrix factorization. s.l. : Proceedings of 6th SIAM Conference on Parallel Processing for Scientific Computing, 1993.

[CHA07] : Charles-Edmond BICHOT : élaboration d'une nouvelle métaheuristique pour le partitionnement de graphe : la méthode de fusion- fission .application au découpage de l'espace aérien, thèse Doctorat Polytechnique De Toulouse 2007.

[CHA07] : P. C HARDAIRE, M. BARAKE et G. P. M CKEOWN : A PROBE based heuristic for Graph Partitioning. IEEE Transaction on Computers, 2007. in submission process.

[DON72] : W.E. Donath et A.J. Hoffman.Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. s.l. : IBM Technical Disclosure Belletin, 1972.

[DRE03] : Johann Dréo, Alain Pétrowski, Patrick Siarry et Eric Taillard.Métaheuristiques pour l'optimisation difficile. s.l. : Eyrolles, 2003.

[DHI04b] : Inderjit S. D HILLON, Yuqiang G UAN et Brian K ULLIS : A Unified View of Kernel kmeans, Spectral Clustering and Graph Cuts. Rapport technique TR-04-25, University of Texas at Austin, 2004.

[DHI07] : Inderjit S. D HILLON, Yuqiang G UAN et Brian K ULIS : Weighted Graph Cuts without Eigenvectors : A Multilevel Approach. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2007. To appear.

[FRA12] : François Anceau, Yvan Bonnassieux « conception des circuits VLSI du composant au système ». chapitre 1, 2012.

[FEL04] : Pedro F. FELZENSZWALB et Daniel P. H UTTENLOCHER : Efficient Graph-Based Image Segmentation. International Journal of Computer Vision, 59(2):167–181, 2004.

[GEO98] : George Karypis et Vipin Kumar.A fast and high quality multilevel schemefor partitioning irregular graphs. s.l. : SIAM journal of scientific computing, 1998.

[GAR79] : Michael R. GAREY et David S. J OHNSON : Computers and intractability : A Guide to the Theory of NP-Completeness. Freeman, 1979.

[HEN98] : Bruce HENDRICKSON : Graph Partitioning and Parallel Solvers : Has the Emperor No Clothes ? (Extended Abstract). In Proceedings of the Workshop on Parallel Algorithms for Irregularly Structured Problems, pages 218–225, 1998.

[HOL81] : Ian H OLYER : The NP-Completness of Some Edge-Partition Problems. SIAM Journal of Computing, 10(4):713–717, 1981.

[INT14] : "Intel Readying 15-core Xeon E7 v2". AnandTech. Retrieved 2014-08-09.

[KER70] : B.W. Kernighan et S.Lin.An efficient heuristic procedure for partitioning graphs. s.l. : Bell System Technical Journal, 1970.

[KNU68] Donald E. Knuth. The art of computer programming, volume i : Fünda-mcntal algorithms. Addison-Wcslcy, 1968. (Cité en page 4.)

[KAR98d]: George KARYPIS et Vipin K UMAR : Multilevel k-way Partitioning Scheme for Irregular Graphs. Journal of Parallel and Distributed Computing, 48(1):96–129, 1998.

[MAH06]: A. Mahdoum, N. Badache, H. Bessalah "An Efficient Assignment of Voltages and Optional Cycles for Maximizing Rewards in Real-Time Systems with Energy Constraints" Journal Of Low Power Electronics (JOLPE), Vol.2, N°2, August 2006, American Scientific Publishers, pp. 189-200, ISSN: 1546-1998

[MAH07]: A. Mahdoum, M. L. Berrandjia "FREEZER2: Un Outil à Base d'un Algorithme Génétique pour une Aide à la Conception de Circuits Digitaux à Faible Consommation de Puissance" IEEE/FTFC'07, 21-23 Mai 2007, Paris, France, pp. 143-148

[MAH09]: A. Mahdoum "Synthèse de Systèmes Monopuce à Faible Consommation d'Energie" FTFC'09, 3-5 Juin 2009, Centre Suisse d'Electronique et de Microtechnique, Neuchâtel, Suisse

[MAH10]: A. Mahdoum, R. Benmadache, M.L. Berrandjia, A. Chenouf "An Efficient Low-Power Buffer Insertion with Time and Area Constraints" 14th ICC (International Conference on Circuits), 22-24 July 2010, Corfu, Greece

[MAH11]: A. Mahdoum, L. Hamimed, M. Louzri, M. Saadaoui "Data Coding Methods for Low-Power Aided Design of Submicron Interconnects" IEEE /FTFC'11 (Faible Tension Faible Consommation), May 30 - June 1 2011, Marrakech, Morocco

[MAH12]: A. Mahdoum "A New Design Methodology of Networks on Chip" IEEE /ASQED'12 (Asian Symposium on Quality Electronic Design), July 10-11 2012, Penang, Malaysia

[MAH13]: A. Mahdoum "Architectural Synthesis of Networks On Chip" IEEE /ICIEA'13 (International Conference on Industrial Electronics and Applications), 19-21 June 2013, Melbourne, Australia

[MAH14] A. Mahdoum "Networks on Chip Design for Real-Time Systems" 27th IEEE/SOCC'14 (System On Chip Conference), 2-5 Septembre 2014, Las Vegas, Nevada, USA

[MOO65] : Gordon E. Moore, « Cramming More Components Onto Integrated Circuits », *Electronics*, vol. 38, 19 avril 1965.

[MAR05] : Jacob G. M ARTIN : Subproblem Optimization by Gene Correlation with Singular Value Decomposition. In Proceedings of the ACM Genetic and Evolutionary Computation Conference, pages 1507–1514, 2005.

[MAR06] : Jacob G. M ARTIN : Spectral techniques for graph bisection in genetic algorithms. In Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation, pages 1249–1256, 2006.

[ROU05] : Frédéric ROUSSEAU « Conception des systèmes VLSI ».Techniques de l'ingénieur Architecture et tests des circuits numériques, 2005.

[RAK13] : Raksmey PHAN : Méthodes exactes et approchées par partition en cliques de graphes. Thèse de doctorat université Blaise Pascal, 2013.

[RON05] : Dorit RON, Sharon W ISHKO -S TERN et Achi B RANDT : An Algebraic Multigrid based Algorithm for Bisectioning General Graphs. Rapport technique MCS05-01, Weizmann Institute of Science, 2005.

[SIM97] : Horst D. SIMON et Shang-Hua T ENG : How Good is Recursive Bisection ? SIAM Journal on Scientific Computing, 18(5):1436–1445, 1997.

[SEL06] : Navaratnasothie SELVAKKUMARAN et George K ARYPIS : Multi-Objective hypergraphpartitioning algorithms for cut and maximum subdomain-

degree minimization. IEEE Transactions on Computer-Aided Design, 25(3):504–517, 2006.

[SCH99] : Kirk SCHLOEGEL, George K ARYPIS et Vipin K UMAR : A New Algorithm for Multiobjective Graph Partitioning. In Proceedings of the European Conference on Parallel Processing, pages 322–331, 1999.

[TUR36] : Turing, A. M. (1937) [Delivered to the Society November 1936]. "On Computable Numbers, with an Application to the tscheidungsproblem". *Proceedings of the London Mathematical Society*. 2 **42**.

[WAN03] : S. W ANG et J. S ISKIND : Image segmentation with ratio cut. IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(6):675–690, 2003.

[YAR00] : Elie YARACK : An Evaluation of Move-Based Multi-Way Partitioning Algorithms. In Proceedings of the 2000 IEEE International Conference on Computer Design : VLSI in Computers & Processors, page 363, 2000.

[ZHA04] : Ying ZHAO et George K ARYPIS : Empirical and Theoretical Comparisons of Selected Criterion Functions for Document Clustering. Machine Learning, 55(3):311–331, 2004.