

TME 8 : Méthode de partitionnement de graphe

Min-Cut

1. Objectif
2. A. Introduction
3. B. Algorithme MinCut? glouton
4. C. Structures de données
 1. C.1. représentation du graphe
 2. C.2. représentation du gestionnaire des mouvements
5. D. Fonctions d'accès aux structures de données
6. E. Travail à réaliser

Objectif

On souhaite implanter en langage C un algorithme de partitionnement de graphe utilisant la méthode Min-Cut, dont le principe a été présenté en cours.

A. Introduction

Soit un graphe $G(N, H)$, où N est l'ensemble des noeuds, et H l'ensemble des arêtes. Une bipartition de G est formée de 2 sous-ensembles L (left) et R (right) de l'ensemble des noeuds N , dont l'intersection est vide, et l'union est égale à N .

En supposant que le nombre de noeud est pair, on dit qu'une bipartition est équilibrée si $L = R = N / 2$

Pour ce qui nous concerne, on s'intéresse à une net-list de cellules précaractérisées, et on cherche à partitionner l'ensemble des cellules en deux sous-ensembles L et R tels que le nombre de signaux traversant la frontière entre les deux sous-ensembles soit minimal.

Nous allons raisonner sur un graphe non-orienté, défini de la façon suivante:

Les noeuds C_i représentent les cellules de la net-list ($0 < i < N-1$). Il existe une arête entre deux cellules C_i et C_j chaque fois qu'il existe au moins un signal connectant un port de la cellule C_i à un port de la cellule C_j .

Attention: avec cette définition, à un seul signal de la net-list peuvent correspondre plusieurs arêtes dans le graphe: si le signal de sortie d'une porte logique C_x attaque n portes C_1 à C_n , on a n arêtes dans le graphe.

On appelle «matrice d'adjacence» $\{A_{ij}\}$ la matrice carrée $N * N$ qui représente la structure du graphe:

- $A_{ij} = 1$ si il existe une arête dans le graphe entre les noeuds C_i et C_j
- $A_{ij} = 0$ sinon

On définit comme suit la fonction de coût du problème d'optimisation MinCut?:

$$FC = \sum_{i,j} A_{ij} * K_{ij}$$

avec $K_{ij} = 1$ si les cellules C_i et C_j sont de part et d'autre de la frontière
 $K_{ij} = 0$ sinon

B. Algorithme MinCut? glouton

On part d'une bi-partition équilibrée initiale arbitraire en deux ensembles L et R, et on cherche à optimiser la fonction de coût FC, sans forcément rechercher le minimum absolu, mais avec un temps de calcul très court.

L'idée de base de l'algorithme est de pré-calculer, pour chaque cellule C_i un gain $G(i)$ représentant la variation de la fonction de coût FC lorsqu'on déplace la cellule C_i d'un côté de la frontière à l'autre (sans déplacer aucune autre cellule).

Le gain $G(i)$ est défini par:

$$G(i) = \sum_j A_{ij} * D_{ij}$$

avec $D_{ij} = +1$ si les cellules C_i et C_j sont de part et d'autre de la frontière
 $D_{ij} = -1$ sinon

Après avoir calculé le gain $G(i)$ pour toutes les cellules C_i , on classe les cellules dans deux listes ordonnées par gain décroissant (une liste $GAIN_L$ pour les cellules appartenant à L et une liste $GAIN_R$ pour les cellules appartenant à R).

On transfère la première cellule C_i de la liste ordonnée $GAIN_L$ vers R, on met à jour les gains des cellules adjacentes à C_i , ainsi que l'ordre des listes $GAIN_L$ et $GAIN_R$. La cellule C_i transférée dans R n'est pas rangée dans $GAIN_R$, car elle n'est plus autorisée à bouger.

On transfère la première cellule C_i de la liste ordonnée $GAIN_R$ vers L, on met à jour les gains des cellules adjacentes à C_i , ainsi que l'ordre des listes $GAIN_L$ et $GAIN_R$. La cellule C_i transférée dans L n'est pas rangée dans $GAIN_L$, car elle n'est plus autorisée à bouger.

On recommence les deux étapes ci-dessus tant que cet échange entraîne une diminution de la fonction de coût.

C. Structures de données

On a besoin de deux structures de données séparées pour représenter d'une part le graphe à partitionner, et d'autre part les listes $GAIN_L$ et $GAIN_R$. Cette seconde structure de donnée étant utilisée pour calculer les mouvements de cellules à effectuer, est appelée «gestionnaire de mouvements».

C.1. représentation du graphe

Les noeuds du graphe sont identifiés par un index compris entre 0 et (NBNODE-1).

La structure du graphe est représentée par un tableau `NODE[ ]` indexé par le numéro du noeud (la taille du tableau est donc égale à `NBNODE`). Chaque case du tableau `NODE[i]` contient un pointeur sur une liste de bi-pointeurs représentant l'ensemble des noeuds adjacents au noeud `[i]`. Il y a donc un bipointeur pour chaque arête attachée au noeud `[i]`.

Un second tableau `PART[ ]` est lui aussi indexé par le numéro du noeud. Chaque case `PART[i]` contient un entier définissant à quel sous-ensemble appartient le noeud `[i]`: (valeur 0 pour l'ensemble L, et valeur 1 pour l'ensemble R).

Le graphe est donc représenté par la structure suivante:

```
typedef struct graph_t {
    unsigned    NBNODE; // nombre de noeuds du graphe
```

```

bip_t *      * NODE; // pointeur sur le tableau NODE[NBNODE]
unsigned     * PART; // pointeur sur le tableau PART[NBNODE]
} graph_t

```

C2. représentation du gestionnaire des mouvements

On appelle arité d'un noeud du graphe, le nombre d'arêtes attachées à ce noeud. Le gain maximum $G(i)$ associé au mouvement d'un noeud $[i]$ est au plus égal à l'arité du noeud $[i]$: Ce gain peut être positif ou négatif, mais sa valeur absolue ne peut pas être supérieure au nombre des noeuds auquel il est connecté. Soit G_{MAX} la valeur maximale de l'arité (en considérant tous les noeuds du graphe). Pour tout noeud $[i]$ du graphe, on a donc $G_{MAX} < G(i) < G_{MAX}$

Pour représenter la liste ordonnée $GAIN_L$, on effectue une partition de l'ensemble des cellules appartenant à L , en regroupant dans un même sous ensemble tous les noeuds possédant le même gain. Ce sous ensemble est représenté par une liste doublement chaînée, dont chaque élément de type `bichain_t` contient un index de noeud. On définit un tableau de pointeurs $GAIN_L[]$ possédant $2 * G_{MAX} + 1$ cases, et indexé par une valeur de gain comprise entre G_{MAX} et G_{MAX} . Chaque case correspond à une valeur de gain, et contient un pointeur sur une des listes doublement chaînées définies ci-dessus.

On fait la même chose pour représenter la liste $GAIN_R$.

Au début de l'algorithme, chaque noeud du graphe est inséré dans la liste chaînée qui correspond à sa valeur de gain. Au fur et à mesure que les noeuds sont transférés, les listes se vident, car un noeud qui a subi un mouvement n'est plus autorisé à bouger. Puisque le gain des noeuds changent au cours de l'algorithme, un même noeud peut être retiré d'une liste et inséré dans une autre, ce qui justifie l'utilisation de listes doublement chaînées.

Dans la structure de donnée «graphe_t» représentant le graphe, les noeuds du graphe sont identifiés par leur index. On a donc besoin d'introduire dans la structure «move_manager_t» représentant le gestionnaire de mouvements, un tableau de pointeurs $NODES[]$, indexé par le numéro du noeud, et contenant un pointeur sur la structure `bichain_t` représentant un les noeuds du graphe dans les listes doublement chaînées $GAIN_L$ et $GAIN_R$. On utilise donc les structures suivantes:

```

typedef struct bichain_t {
    bichain_t *NEXT; // pointeur sur le noeud suivant de même gain
    bichain_t *PREV; // pointeur sur le noeud précédent de même gain
    unsigned   INODE; // index du nSud
    int        IGIN;  // index dans le tableau des gains
    unsigned   PART;  // numero de la partie 0 pour R, 1 pour L
} bichain_t

typedef struct move_manager_t {
    int      GMAX; // la taille des tableaux GAIN_R et GAIN_L est égale à 2*GMAX+1
    bichain_t **GAIN_L; // pointeur sur le tableau de pointeurs pour la partie gauche
    bichain_t **GAIN_R; // pointeur sur le tableau de pointeurs pour la partie droite
    int      LX; // index de la première case non vide de GAIN_L (init à -1)
    int      RX; // index de la première case non vide de GAIN_R (init à -1)
    bichain_t **NODES; // pointeur sur le tableau de pointeurs sur les noeuds
} move_manager_t

```

D. Fonctions d'accès aux structures de données

On dispose d'un ensemble de fonctions permettant de construire et de manipuler les deux structures de données `graph_t` et `move_manager_t`.

graph_t * parse_graph(char *filename)

Cette fonction prend pour argument un nom de fichier au format.dot représentant le graphe à partitionner, et construit en mémoire la structure graph_t correspondante, et renvoie un pointeur sur cette structure. Si aucune partition initiale n'est définie dans le fichier .dot, les noeuds d'index inférieur à NBNODE/2 sont arbitrairement rangés dans le sous-ensemble L, et les noeuds d'index supérieur à NBNODE/2 sont rangés dans le sous-ensemble R.

Un exemple de fichier au format .dot est donné ci-dessous. Les noeuds du graphe sont identifiés par un index, il y a une ligne par arête, les deux sous-graphes cluster0 et cluster1 définissent la partition initiale et sont optionnels.

```
graph
{
  // les clusters sont optionnels
  subgraph cluster0 {0 1 2 3}
  subgraph cluster1 {4 5 6 7}
  0--4
  0--5
  0--7
  1--5
  3--7
  1--2
  5--6
}
```

void drive_graph(graph_t *p, char *filename)

Cette fonction prend en paramètre un pointeur p sur une structure graph_t, ainsi que le nom du fichier de sortie, et écrit sur disque un fichier au format .dot, en conservant dans le fichier les informations de partition (sous-ensemble L et R). Ce format est affichable sous forme graphique et permet de visualiser la partition.

bichain_t * cons_bichain (bichain_t *prev, bichain_t *next, unsigned inode, int igain, unsigned part)

Cette fonction alloue la mémoire nécessaire pour créer un élément d'une liste doublement chaînée, elle initialise les différents champs de cette structure, et renvoie un pointeur sur cette structure.

void free_bichain (bichain_t *p)

Cette fonction libère la mémoire allouée lors de la construction de l'objet bichain_t par la fonction cons_bichain().

move_manager_t * cons_move_manager(graph_t *p)

Cette fonction prend pour argument un pointeur sur la structure représentant le graphe partitionné, et construit en mémoire la structure move_manager_t correspondant à cette partition.

void add_node (move_manager_t *mm, bichain_t *node)

Cette fonction prend pour arguments un pointeur sur la structure représentant le gestionnaire de mouvements, un pointeur sur un élément de liste doublement chaînée (représentant un noeud), contenant l'index du nSud dans le graphe, un entier représentant l'index dans le tableau GAIN_R ou GAIN_L, et un entier définissant le sous-ensemble d'appartenance (0 pour R, 1 pour L). Elle range ce noeud dans la liste doublement chaînée correspondante, elle met à jour le tableau NODES et les index LX ou LR. Les champs NEXT et PREV du nSud ne sont pas utilisés, ils sont initialisés par la fonction add_node elle-même.

bichain_t * del_node(move_manager_t *mm, bichain_t *node)

cette fonction prend pour arguments un pointeur sur la structure représentant le gestionnaire de mouvements, et un pointeur sur un élément de liste doublement chaînée (représentant un noeud). Elle retire cet élément de la liste doublement chaînée, et renvoie un pointeur sur la structure bichain_t pour une éventuelle libération de la mémoire.

int compute_gain (graph_t *g, unsigned index)

Cette fonction prend pour argument un pointeur sur la structure représentant le graphe, un index désignant un noeud particulier, et calcule le gain associé à ce noeud, pour la partition courante.

int global_cost (graph_t *g)

Cette fonction prend pour argument un pointeur sur la structure représentant le graphe, calcule la fonction de coût globale pour la partition courante, et renvoie la valeur calculée.

E. Travail à réaliser

Nous vous proposons un programme à trous. Vous allez pouvoir remplacer progressivement les fichiers objets fournis par vos propres fichiers objet.

1. Construction et initialisation du gestionnaire de mouvements

C'est un point important puisque que c'est la structure de base de l'algorithme. Vous disposez pour vous aider d'une fonction `dump_move_manager()` qui affiche le contenu de la structure.

2. Ecriture de l'algorithme MinCut glouton

Ecrivez la boucle réalisant les transferts de cellules tant que le transfert d'une paire de cellules améliore la fonction de coût FC.

3. Exécution et évaluation

Exécutez cet algorithme sur différents graphes que vous définirez vous-mêmes. La partition résultante est-elle toujours la partition optimale? Pourquoi? Donnez un contre-exemple.

4. Ecriture des fonctions

Ecrire les différentes fonctions d'accès aux structures de données définies dans la partie D, et validez ces fonctions en les intégrant peu à peu dans votre programme.

5. construction du graphe en mémoire

L'objectif est ici d'écrire la fonction `parse_graph()`, en utilisant `lex` et `yacc`. Nous vous suggérons de supposer qu'il n'y a pas de clusters dans le fichier, puis de les introduire. Si vous n'avez pas le temps de construire le graphe, étudiez au moins la grammaire de ce petit langage.