Graphes et Algorithme
TD

I1 G3

17 mars 2007

Table des matières

Chapitre 1

Implémentation de graphes

1.1 Exercice 12

(Graphe complémentaire) Le graphe complémentaire d'un graphe simple orienté (V,E) est le graphe (V, V*V - E). Selon que le graphe est représenté par sa matrice d'adjacence ou par des listes de successeurs, écrire un algorithme calculant le complémentaire de tout graphe. Evaluer sa complexité.

1.1.1 Matrice d'adjacence

Commençons tout d'abord par le cas où le graphe est représenté par sa matrice d'adjacence :

Exemple

Pour cela nous allons traiter un exemple afin de mieux comprendre le problème.
La figure 1.1 montre le graphe que nous allons traiter :

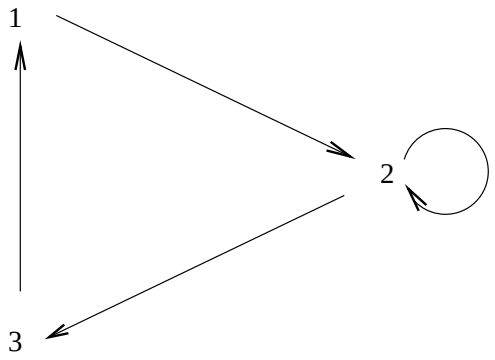

FIG. 1.1 – Graphe G

Le graphe complémentaire associé est donné par la figure 1.2 :

1.1.2 Matrice d'adjacence

Nous allons maintenant nous interresser à leur matrice d'adjacence.
La matrice d'adjacence du graphe G est :

	1	2	3
1	f	v	f
2	f	v	v
3	v	f	f

Celle du complémentaire de G est la suivante :

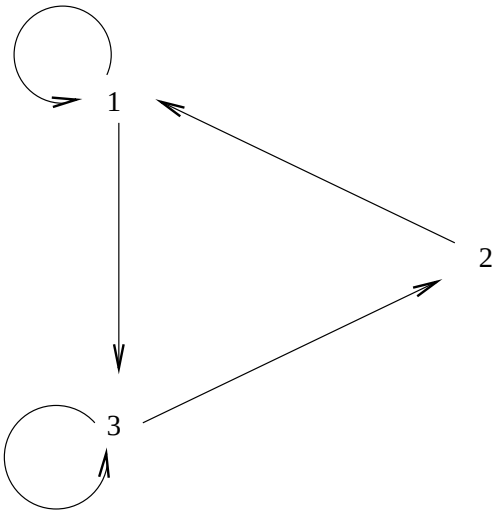

FIG. 1.2 – Complémentaire G' de G

	1	2	3
1	v	f	v
2	v	f	f
3	f	v	v

On remarque que lorsque l'on passe d'une matrice à une autre, on modifie tous les booléens vrai en booléens faux et réciproquement. Cette propriété est logique puisque lorsque nous passons d'un graphe à son complémentaire, toutes les liaisons existantes sont supprimées, et toutes celles qui n'existaient pas sont créées.

L'algorithme

Maintenant que l'on a compris l'idée générale nous allons essayer de résoudre le problème suivant :

Problème : Complémentaire
Entrée : Un graphe g représenté par une matrice d'adjacence
Sortie : Le graphe complémentaire de g

Voici un algorithme qui peut résoudre ce problème :

```
fonction Complémentaire (g : graphe) : graphe
début
    M ← matrice(G)
    n ← nbreSommets(G)
    pour i ←1 à n faire
        pour j ←1 à n faire
            M[i][j] ← non(M[i][j])
        finpour
    finpour
    retourner g
fin
```

Complexité

En ce qui concerne la compléxité en temps, elle est quadratique, et celle en espace constante.
Cependant, on peut se demander si la représentation par matrice d'ajacence est la meilleure représentation possible d'un graphe. Nous allons donc nous intéresser à une autre représentation : un tableau de listes de successeurs.

1.1.3 Listes de successeurs

Représentation

Voici la nouvelle représentation de notre graphe G :

(2)
(2,3)
(1)

Voici celle du graphe complémentaire :

(1,3)
(1)
(2,3)

L'algorithme principal

Ainsi, nous devons répondre au problèmes suivant :

Problème : Complémentaire
Entrée : Un graphe G représenté par un tableau de liste d'entiers
Sortie : Un graphe complémentaire de G complémentaire de L dans [1,n].

Voici , un algorithme résolvant ce problème :

```
fonction Complémentaire (G : graphe) : graphe
début
    tabL ← tabListeAdj(G)
    n ← nbreSommets(G)
    pour i ←1 à n faire
        pour j ←1 à n faire
            tabL[i] ← ListeComplémentaire(tabL[i], n)
        finpour
    finpour
    retourner G
fin
```

Liste complémentaire

Il nous reste à trouver l'algorithme ListeComplémentaire résolvant le problème suivant :

Problème : ListeComplémentaire
Entrée : Une liste d'entier L, un entier n avec L $\subseteq$ [1,n]
Sortie : Une liste d'entier complémentaire à L dans [1,n]

Deux algorithmes ont été imaginés pour résoudre ce problème. L'un utilisant une boucle *tantque* et l'autre une boucle *pour*.

Voici l'algorithme pour la boucle *pour*

```
fonction listeComplémentaire (L : liste, n : Entier) : liste
début
    LC ← listeVide()
    L ← tri(L)
    pour i ←1 à n faire
        si estVide(L) alors
```

```
            LC ← ajouterFin(LC, i)
        sinon
            si i < premier(L) alors
                LC ← ajouterFin(LC, i)
            sinon
                L ← enleverPremier(L)
            finsi
        finsi
    finpour
    retourner LC
fin
```

Voici l'algorithme pour la boucle *tantque*

```
fonction listeComplémentaire (L : liste, n : Entier) : liste
début
    LC ← listeVide()
    L ← tri(L)
    i ← 1
    tant que i < n faire
        si estVide(L) alors
            LC ← ajouterFin(LC, i)
        sinon
            si i < premier(L) alors
                LC ← ajouterFin(LC, i)
            sinon
                L ← enleverPremier(L)
            finsi
        finsi
        i ← i + 1
    fintantque
    retourner LC
fin
```

Voici un exemple, pour les différentes fonctions utilisées dans *listeComplémentaire* :

- ajouterFin((3,7,18),2) = ((3,7,18,3))
- premier((3,7,18))= 3
- enleverPremier((3,7,18))=(7,18)

Correction

Nous allons prouver la correction de la seconde version de l'algorithme, la première étant laissée au lecteur. La preuve est similaire avec une petite difficulté à cause de la boucle *pour*...

La terminaison est évidente, la suite formée des valeurs successives prises par i est strictement croissante et bornée. L'invariant est : L est triée, LC est triée, $L \neq \emptyset \Rightarrow i \leq$ premier(L), $LC \neq \emptyset \Rightarrow i >$ dernier(L), $A \cup LC \supseteq [1, i-1]$ et $A \cap LC = \emptyset$.

- L est triée :
  Ceci est toujours vrai vu que les seules opérations effectuées sont de choisir et de retirer le premier élément de la liste. Celles-ci ne modifient pas le fait que L soit triée.
- $LC \neq \emptyset \Rightarrow i >$ dernier(L) :
  Au début, LC est vide, donc la proposition est vraie. Ensuite, si nous ajoutons i à LC, alors nous incrémentons i à la fin de la boucle, donc la proposition reste vraie même si i n'est pas ajoutée.
- LC est triée :
  Au départ, LC est vide donc triée. A chaque tour de boucle, si nous ajoutons i à LC, d'après la proposition précédente et vu que nous ajoutons à la fin de la liste, LC reste triée.

- $L \neq \emptyset \Rightarrow i \leq$ premier(L) :
  Au début, i = 1, or L $\subseteq$ [1, n] donc la proposition est vraie. Ensuite, à chaque tour de boucle, nous enlevons le premier de L si i est supérieur ou égal à celui-ci. Or, i $\leq$ premier(L), donc le premier est retiré lorsque i est égal au premier de L. De plus, nous incrémentons i à la fin de la boucle, donc la proposition est vrai dans ce cas. Maintenant si nous n'enlevons pas le premier donc i < premier(L) et i est incrémenté donc i $\leq$ premier(L).
- A$\cup$LC $\supseteq$ [1,i-1] :
  Cette proposition est vérifiée car les boucles précédant celles en cours ont placées les entiers appartenant à [1,i-1], dans la liste LC s'ils n'appartenaient pas à L.
- A$\cap$LC = $\emptyset$:
  Pour cette proposition, comme LC au début est la liste vide et que l'on ajoute un entier à LC seulement si cet entier n'appartient pas à L, A et LC n'ont pas d'entiers en commun.

Autre interprétation de listeComplémentaire

Maintenant nous allons essayer d'éviter les effets de bord, c'est-à-dire éviter la modification de la liste passée en paramètre.

Pour cela il faudrait mettre en place un curseur. Ainsi, nous allons envisager une nouvelle représentation, c'est-à-dire un couple avec pour premier élément la liste, et en second le curseur.

Supposons n le nombre d'éléments de la liste et c la valeur du curseur, alors c $\in$ [1, n+1].

Nous allons décrire les fonctions à modifier afin d'avoir la nouvelle version de *listeComplémentaire*, l'écriture de celle-ci étant laissée au lecteur. Tout d'abord, les fonctions *premier* et *enleverPremier* sont remplaçées respectivement par *eltCurseur* et *avancerCurseur*. De plus, à première vue, la correction de la fonction serait plus simple, car il n'est pas nécessaire d'avoir une copie de la liste L.

Voici un exemple, pour les différentes fonctions :
- estVide((3,7,18),3) -> faux
  estVide((3,7,18),4) -> vrai
- avancerCurseur(((3,7,18),2)) -> (((3,7,18),3))
- eltCurseur((((3,7,18),2)))-> 7

Complexité

En ce qui concerne la complexité en temps, celle-ci est quadratique, et de même pour la complexité en espace. Néanmoins, la représentation d'un graphe avec un tableau de liste de successeurs est plus avantageuse, vu que nous prenons que l'espace nécessaire. Cette propriété ne doit pas être omise lors de l'implémentation de cette représentation, car il s'agit d'une propriété intéressante.

1.1.4 Conclusion

Parmi, les deux représentations étudiées aucunes ne semblent être avantageuses à l'autre concernant les complexités en temps.