Victor Chmiel Pierre-Yves Nicolle

Etude de compléxité : Recherche de Circuit

Table des matières

Introduction

Ce TP a pour but de mettre en place quelques algorithmes afin de détecter des circuits dans des graphes non orientés. Il faudra ensuite évaluer la complexité des algorithmes proposés.

On a testé ces algorithmes sur quelques graphes relativement petits.

Représentation des graphes

Nous avons décidé de représenter les graphes par liste d'adjacence, puisque c'est la méthode qui minimise l'espace de stockage d'un graphe (pour la méthode de matrice d'adjacence, on a une structure qui est de taille proportionnelle à n^2 avec n le nombre de sommets du graphe, alors que la liste d'adjacence ne requiert qu'un espace de taille proportionnelle au nombre d'arrêtes dans le graphe.

La représentation par liste d'adjacence consiste à créer la liste des voisins de chaque sommet du graphe. Ainsi, G[i] stockera la liste des voisins du sommet i.

Recherche exhaustive

Cette méthode consiste à énumérer tous les sous-graphes connexes, et à évaluer si l'un de ces sous-graphes est un circuit. Afin de faire cela, il suffit de vérifier s'il y a **plus d'arrêtes que de sommet**. Le plus long sera de créer l'ensemble des sous-graphes connexes, en supprimant des sommets et les arrêtes menant vers ces sommets sur l'ensemble des sous-graphes de taille m.

```
fonction exhaustif(graphe)
begin
        L = sous_graphes(graphe)
        pour sG dans L
                si nombre_arrêtes >= nombre_sommets
                alors
                        retourner Vrai
        finPour
        retourner Faux
fin
```

Cet algorithme est clairement un algorithme qui va être très mauvais en terme de complexité. Si le graphe possède n sommets, on a alors un nombre de sous-graphes à calculer qui évolue exponentiellement par rapport à n.

Ci-dessous, un tableau avec le temps d'exécution pour des graphes de différentes tailles :

Nombre de sommets	10	11	12	13
Temps d'exécution (s)	1,5	4,2	13,8	62

A partir de 14 sommets, il devient impossible d'utiliser cette méthode. De plus, il y a certains exemples de faux positifs… On a donc une méthode beaucoup trop naïve, on va plutôt préférer une autre méthode plus astucieuse.

Recherche par parcours en largeur

La deuxième méthode que nous avons décidé d'implémenter repose sur un parcours en largeur. On part du premier sommet ayant au moins un voisin. Ce sommet sera de niveau 0. Ensuite on va commencer le parcours à partir du sommet de niveau 0. Si l'on rencontre un voisin qui n'a pas encore été visité (ie son niveau n'a pas encore été défini), alors on lui donne un niveau incrémenté par rapport au sommet précédemment visité. Si le voisin a déjà été visité (ie de niveau strictement inférieur a celui de son prédécesseur) on a trouvé un cycle. Si l'on parcours tous les sommets, le graphe ne possède pas de cycles.

La complexité temporelle est en $O(Card(S))$ (on va parcourir toutes les arrêtes n supposant que le graphe soit connexe.

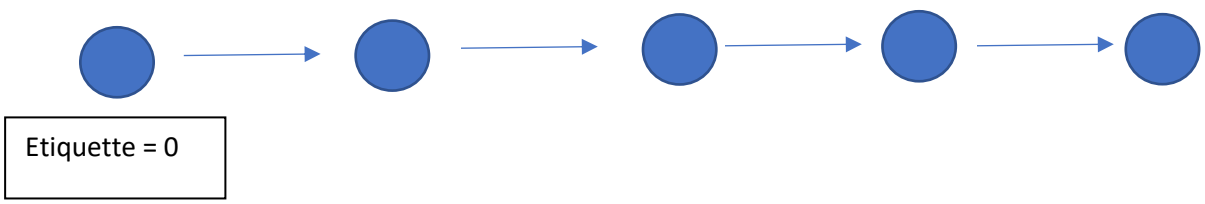
```
fonction parcours(graphe)
begin
        n = nombre_sommets(graphe)
        file = []
        niveau = [Null, Null, … Null]    /* de la meme taille que le graphe
        pour k allant de 0 a n – 1
                tant que file est vide
                        /* on cherche le premier sommet ayant au moins un voisin
                        si graphe[k] non vide
                                alors mettre k dans la file
                fin tant que
        fin pour
        tant que file est non vide
                u = sortir le premier element de file
                pour tous les voisins de u
                        si niveau[voisin] == Null
        `                       alors niveau[voisin] = niveau[u] + 1
                                mettre voisin dans file
                        sinon
                                retourner Vrai
                fin pour
        fin tant que
        retourner Faux
fin
```

Cet algorithme s'exécute presque instantanément, en tout cas avec les exemples que nous avons utilisés.

Recherche par label

Cette dernière méthode consiste à associer une étiquette (un entier) à chaque sommet, initialisée à 0. On met une étiquette à jour dès que le sommet A associé est voisin d'un autre sommet B, de la valeur de l'étiquette du sommet B + 1. On effectue cette mise à jour de l'étiquette pour tous les sommets qui sont des voisins (ie pour toutes les arrêtes). On répète cela autant de fois qu'il y a de sommets.

Sur une chaine de longueur n, à la première itération, le premier sommet de la chaine simple restera à 0, puisqu'il n'est voisin de personne.



En réitérant, on aura donc toujours les mêmes valeurs d'étiquette, qui seront forcément inférieures à n.

Dans le cas d'un cycle, aucun sommet n'aura d'étiquette à 0, et à chaque itération, chaque étiquette sera au moins incrémentée de 1. Au bout de n itérations, on aura donc des étiquettes supérieures ou égales n.

Il suffit donc de vérifier si il existe une étiquette supérieure ou égale à n.

```
fonction label(graphe)
begin
        n = nombre_sommets(graphe)
        label = [0, 0, … 0]  /* de taille n
        pour k allant de 1 a n
                pour i allant de 0 à n – 1
                        pour j allant de 0 à n – 1
                                si j est voisin de i et label[i] >= label[j]
                                        alors label[j] = label[i] + 1
                        fin pour
                fin pour
        fin pour
        estCyclique = Faux
        pour k allant de 0 à n – 1
                si label[k] >= n
                        alors estCycle = Vrai
        fin pour
        retourner estCyclique
fin
```

Cet algorithme a une complexité temporelle en $O(n^3)$, avec n le nombre de sommets dans le graphe. Cette complexité cubique s'explique par la triple boucle tant que.

Tout comme pour l'algorithme par parcours, cet algorithme par étiquette s'exécute presque instantanément pour de petites valeurs de n.

Conclusion

Nous avons mis en place un algorithme naïf qui ne respectait pas le cahier des charges (à cause des faux positifs) et beaucoup trop long à s'exécuter. Par contre, les deux autres algorithmes plus évolués sont capables de détecter des cycles pour tous les graphes, même si leur complexité est polynomiale.