HÉCART Olivier
MÉRÉ Aurélien

Projet d'algorithmique

Spécifications

Table des matières

Projet d'algorithmique 1
Table des matières 1
Algorithme 2
  Première partie 2
    Preuve 2
  Deuxième partie 3
    Preuve 4
  Complexité 4
Types de données et fonctions 5
  Classe Sommet 5
    AjouteLien 6
    SupprimeLien 6
    RécupèreLien 6
    Degré 6
    RécupèreNom 7
  Listes chaînées 7
    AjouteSommet 7
    AfficheListe 7
  Format de fichier 7

Algorithme

Ces spécifications définissent le fonctionnement de notre algorithme de recherche d'un cycle eulérien dans un graphe non orienté.

L'algorithme de recherche de cycle eulérien nécessite un graphe en entrée. Ce graphe est modélisé à l'aide d'un tableau de sommets (voir modélisation du graphe). Chaque sommet du graphe est affecté d'un degré, sachant que le degré d'un sommet est le nombre d'arêtes qui le lient à d'autres sommets.

L'algorithme se décompose en deux parties distinctes.

Première partie

La première partie de l'algorithme consiste à tester la parité du degré de chaque noeud dans le but de déceler la présence d'un cycle eulérien. En effet, un graphe non eulérien possède au moins un sommet dont le degré est impair. (voir preuve).

La fonction associée prend un tableau de sommets en paramètre et retourne un booléen : vrai si un cycle eulérien est détecté, faux sinon.

```
Fonction EstEulérien(graphe : tableau de sommets) retourne booléen
Début
        /* On parcours les sommets du graphe * /
        Pour i allant de 0 à NbDeSommets faire
                /* si un sommet est de degré impair, le graphe n'est pas eulérien */
                Si Degré(graphe[ i ] ) mod 2 = 1 alors
                        Retourner(faux) ;
                Fin si
        Fin pour
        Retourner(Vrai) ;
Fin
```

A la fin de cette première partie, le programme peut déclarer la présence d'un cycle eulérien à l'utilisateur et arrêter le processus en cas d'absence.

Preuve

Si notre graphe admet un cycle eulérien, alors chaque arête passant par un sommet s ajoute 2 au degré de s. Comme chaque arête est traversée une seule fois, le degré de chaque sommet est une somme de 2, soit un nombre pair. Par extension, tout graphe connexe dont les nœuds ont tous un degré pair admet un cycle eulérien.

HÉCART Olivier
MÉRÉ Aurélien

Deuxième partie

La deuxième partie de l'algorithme consiste à déterminer un cycle eulérien.

La fonction associée prend un tableau de sommets en paramètre (le graphe) et retourne une liste chaînée de sommets (en réalité une liste de pointeurs sur sommet) représentant le cycle eulérien.

L'algorithme consiste à :

- Choisir un sommet arbitrairement.
- Parcourir le graphe de sommet en sommet en les choisissant arbitrairement.
  - La liste des sommets parcourus est gardée en mémoire
  - Et les arcs parcourus sont effacés du graphe ainsi que le degré des sommets mis à jour.
    (Une copie du graphe d'origine peut être conservée dans plusieurs buts dont un réaffichage)
- Si l'on ne peut plus emprunter d'arête et que l'on est pas revenu au sommet d'origine, le graphe examiné n'est pas eulérien.
- Si l'on revient au sommet d'origine, un premier cycle est créé.
  (On peut ici comparer le nombre d'arêtes parcourues avec le nombre d'arêtes total. En cas d'égalité, le cycle créé est eulérien)
  => on parcourt le cycle créé à la recherche d'un sommet de degré non nul (encore relié à une partie du graphe non parcourue). A chaque sommet de degré non nul rencontré, on re-exécute le présent algorithme et on insère le cycle retourné dans le cycle déjà créé (et au bon endroit : le sommet en question). Cela forme un nouveau cycle composé des deux précédents.

- On obtient, après un ou plusieurs appels récursif, une liste de sommets représentant un cycle eulérien.

On considère une variable globale graphe : tableau de sommets NbAretesTotal et NbAretes : entier.
NbAretes est placée à 0.

```
Fonction cycle(UnSommet : pointeur sur sommet) : ListeDeSommets
Début
        Liste, TeteDeListe : pointeurs sur ListeDeSommets ;
        SommetCourant : pointeur sur sommet ;

        SommetCourant = Un_Des_Sommets_Relies_A(UnSommet) ;
        Liste = TeteDeListe = NouveauNoeud(n , NULL) ;

        /* Parcourir le graphe de sommet en sommet en les choisissant arbitrairement. */
        TantQue (SommetCourant ≠ UnSommet) faire
                Effacer_Lien_Entre(SommetCourant, Liste->sommetPointé) ;
                /* Cette commande efface un lien et met à jour le degré des sommets dans le graphe */.
                NbAretes = NbAretes + 1 ;
                /* La liste des sommets parcourus est gardée en mémoire */
                AjouterDans(Liste, SommetCourant) ;
                SommetCourant = Un_Des_Sommets_Relies_A(SommetCourant) ;
        Fin TantQue
        Effacer_Lien_Entre(SommetCourant, UnSommet) ;

        /* On referme la liste chaînée afin qu'elle forme un cycle */
        TeteDeListe->suivant = Liste ;

        /* On peut ici comparer le nombre d'arêtes parcourues avec le nombre d'arêtes total. En cas d'égalité, le
        cycle créé est eulérien. Ce test est facultatif mais peut améliorer la complexité au mieux et la complexité
        moyenne */
        If (NbAretes ≠ NbAretesTotal) alors
                Retourner(TeteDeListe) ;
        Fin si
```

```
        /* on parcourt le cycle créé à la recherche d'un sommet de degré non nul */
        TantQue (Liste ≠ TeteDeListe)
                /* A chaque sommet de degré non nul rencontré, on re-exécute le présent algorithme et on
                insère le cycle retourné dans le cycle déjà créé (et au bon endroit : le sommet en question) */
                Si (degré(Liste->sommetPointé) ≠ 0) alors
                        NoeudTemporaire1 = ChercheCycle(Liste->sommetPointé) ;
                        NoeudTemporaire2 = Liste->suivant ;
                        Liste->suivant = ChercheCycle(Liste->sommetPointé) ->suivant ;
                        NoeudTemporaire1->suivant = NoeudTemporaire2 ;
                        Liste = NoeudTemporaire1 ;

                        /* On peut ici comparer le nombre d'arêtes parcourues avec le nombre d'arêtes total.
                        En cas d'égalité, le cycle créé est eulérien. Ce test est facultatif mais peut améliorer la
                        complexité au mieux et la complexité moyenne*/
                        If (NbAretes ≠ NbAretesTotal) alors
                                Retourner(TeteDeListe) ;
                        Fin si
                Liste = Liste->suivant ;
        Fin TantQue
        Retourner(TeteDeListe) ;
Fin
```

Preuve

Nous allons raisonner par l'absurde.

Supposons que le graphe retourné par l'algorithme n'est pas eulérien.
Dans ce cas, et par définition, deux cas sont possibles :

- (1) Soit toutes les arêtes du graphe n'ont pas été parcourues
- (2) Soit une arête a été parcourue plus d'une fois

(1) Si toutes les arêtes n'ont pas été parcourues, cela implique qu'il reste des sommets de degré non nul. Or l'algorithme continue de fonctionner tant qu'il reste des sommets non nuls et que ceux-ci n'ont pas été intégrés dans la liste des sommets composant le graphe eulérien. En conséquence, cette situation ne peut se présenter.

(2) Lorsqu'une arête A est ajoutée dans la liste des sommets composant le graphe eulérien, cette arête est supprimée du graphe d'origine et le degré des nœuds reliés par cette arête est mis à jour. En conséquence, elle n'est plus exploitable par les itérations suivantes de l'algorithme, et ne peut être parcourue plus d'une fois. Ce cas ne peut donc se présenter.

Complexité

La complexité de notre première partie de l'algorithme est de n comparaisons, où n représente le nombre de sommets composant le graphe. Effectivement, on vérifie le degré de chaque sommet, or comme il y a n sommets, cela implique un total de n comparaisons. On a alors une complexité en O(n).

La complexité de la seconde partie est beaucoup plus délicate, de par le fait que le graphe que l'on utilise varie au cours du temps (en raison de la suppression des arcs à chaque itération de l'algorithme), et qu'en conséquence, la complexité elle-même change au cours du temps.

Il est toutefois possible de calculer une complexité minimale, et une complexité majorant la complexité réelle, à savoir une complexité supérieure à la complexité maximale, mais dont le calcul est beaucoup plus recevable. Les expérimentations préliminaires montrent que la complexité moyenne est beaucoup plus proche de la complexité minimale que de la complexité maximale.

La complexité se décompose en deux parties : la complexité pour la suppression des arêtes, et la complexité pour la recherche des cycles.

Complexité minimale

La complexité minimale se calcule dans le cas où l'on dispose d'un graphe connexe dont le degré de chaque sommet est 2. Dans ce cas, le parcours du graphe s'effectue en v, où v est le nombre d'arêtes du graphe. Comme chaque sommet ne dispose que de deux liens, la suppression d'une arête se fera en O(1) suivant l'algorithme mis en place, présenté dans la section suivante. Comme il faut supprimer v arêtes, la suppression se fera en 2v comparaisons (le lien est à supprimer sur chaque nœud). La complexité sera donc de l'ordre de 3v, soit en O(v).

Complexité maximale

Suppression des arêtes : il y a v arêtes à supprimer, et la liste des arêtes contient au maximum (n-1) pointeurs dans le cas des graphes que nous étudions, où n est le nombre de sommets du graphe. La complexité maximale sera donc de v*(n-1). En réalité, elle est beaucoup plus faible, car d'une part le nombre d'arêtes liées à un sommet diminue à chaque passage sur ledit sommet, et d'autre part (n-1) est le maximum que peut supporter un sommet. En moyenne elle sera deux fois moindre.

Recherche de sommets dont le degré n'est pas nul : Il y a n sommets à vérifier, chacun pouvant conduire à un nouvel appel récursif à la fonction. Toutefois, comme il y a un maximum de v arêtes sur le graphe, et que chaque nouvel appel produit un cycle eulérien d'un minimum de 3 arêtes, on aura un maximum de v/3 appels pour n sommets, soit n*v/3. Toutefois, comme les cycles trouvés par l'algorithme ont en pratique une taille supérieur à 3 arêtes, la complexité est beaucoup plus faible.

En tous les cas, la complexité maximale de l'algorithme sera donc de l'ordre de (n-1)*v+n*(v/3), soit une complexité de l'ordre de O(v*n).

Complexité moyenne

Notre complexité moyenne sera donc de l'ordre de : $O(v) < C << O(v*n)$
Avec v le nombre d'arêtes du graphe, et n le nombre de sommets.
En pratique on trouve de l'ordre de a.v.log(v), où a est une variable dépendante de v et de n, relativement faible. Notons toutefois qu'il s'agit d'une appréciation purement spéculative.

Types de données et fonctions

Classe Sommet

Notre graphe est réalisé par l'utilisation d'une classe Sommet. Une instance de cette classe contient toutes les fonctions nécessaires pour l'ajout de liens avec d'autres sommets, la suppression de liens, la recherche de liens éventuels, etc…

Avant de décrire la classe Sommet, précisons que dans notre routine principale, le graphe est représenté sous la forme d'un tableau de classes Sommet, ce qui nous permet d'effectuer des traitements directement sur les sommets, mais aussi sur l'ensemble avec une complexité en n.

La classe Sommet contient un tableau de pointeurs vers d'autres sommets. Ce tableau est mis à jour lors de l'ajout et la suppression de liens. Une variable donne l'indice du dernier élément du tableau. Voici la liste des fonctions prises en charge par une instance de la classe sommet :

- *Constructeur* : Sommet (nom : chaîne de caractères)
- AjouteLien (s :Pointeur sur sommet, liendouble : booléen)
- SupprimeLien (s : Pointeur sur sommet, liendouble : booléen)
- SupprimeLien (i : indice du sommet, liendouble : booléen)
- RecupereLien () : Pointeur sur sommet
- Degre () : entier
- RecupereNom () : Chaîne de caractères

En outre, elle possède les variables privées suivantes :

- Liens : Tableau de Pointeurs sur sommet
- Degré : Entier
- Nom : Chaîne de caractères

AjouteLien

La procédure AjouteLien crée une arête entre le sommet utilisé, et le sommet donné en argument. Si le booléen liendouble est positionné à vrai, le lien est créé dans les deux sens (ce qui est toujours le cas dans un graphe non orienté). Pour créer ce lien double, la procédure appelle la procédure AjouteLien de l'autre sommet, en se donnant comme destination.

L'algorithme de création de lien est en O(1), car on connaît l'indice du tableau dans lequel le nouveau sommet va être ajouté. En conséquence, il s'agit d'une simple écriture dans le tableau, et d'une mise à jour du degré du sommet.

SupprimeLien

La procédure SupprimeLien existe en deux variantes. La première recherche le sommet donné en argument dans la liste des liens de notre sommet, et une fois trouvé le supprime. Si la valeur de liendouble est positionnée à vrai, la suppression est également effectuée dans l'autre sens. Comme il s'agit d'une recherche dans un tableau contenant au maximum n valeurs, où n est le degré de notre sommet, la complexité de cette routine est alors au pire de 2n.

La variante de cette routine consiste dans le fait que l'on connaisse déjà l'un des indices du tableau, et qu'il ne faut en conséquence faire une recherche que dans un seul des sommets, et non dans les deux. En conséquence, la suppression se fait en O(1) pour un sommet, et en O(n) pour l'autre. La complexité de cette routine est alors au pire de n.

La suppression du lien se fait par remplacement de l'élément à supprimer par le dernier élément. Le dernier élément est ensuite mis à 0, et le degré du sommet est mis à jour.

RécupèreLien

La fonction RécupèreLien renvoie le premier lien de notre sommet vers un autre qu'elle trouve. Il s'agit en fait, si le degré du sommet est strictement positif, du premier élément de notre tableau de liens. La complexité de cette procédure est donc en O(1).

Degré

La fonction Degré renvoie le degré du sommet courant. Le degré est mis à jour lors des ajouts et suppressions d'arêtes, et stocké dans une variable privée de la classe. En conséquence, la complexité de cette routine est en O(1), étant donné qu'il s'agit juste de renvoyer la valeur de cette variable.

RécupèreNom

La fonction RécupèreNom renvoie le nom du sommet, défini lors de la création de celui-ci.

Listes chaînées

Afin de stocker le graphe eulérien, nous utilisons une structure de liste chaînée. Celle-ci est définie très simplement par :

Structure ListeChaînee {
- S : Pointeur sur sommet
- Suivant : Pointeur sur une liste chaînée

}

Afin d'exploiter ces listes chaînées, une fonction AjouteSommet() a été implémentée.

AjouteSommet

AjouteSommet(source : Pointeur sur ListeChaînee, sommet : Pointeur sur sommet) : Pointeur sur liste chaînée

On spécifie le sommet à ajouter à la liste chaînée source ; celui-ci est ajouté au début de la liste chaînée. On renvoie ensuite la liste chaînée résultante.

AfficheListe

AfficheListe(source : Pointeur sur ListeChaînée)

Cette procédure affiche le contenu de la liste chaînée par un parcours itératif jusqu'à ce que l'élément suivant soit nul.

Format de fichier

Afin de préserver notre graphe pour une réutilisation ultérieure, il est possible de sauvegarder et restaurer, à l'aide de fichiers, notre graphe. Ces fichiers, tels qu'ils ont été définis dans le cahier des charges, sont composés de deux parties. D'une part les propriétés relatives aux sommets, à savoir leur nom et leur position sur la feuille de dessin, chaque paramètre étant séparé entre virgules ; et d'autre part les propriétés relatives aux arêtes, et notamment la liste des arêtes sources et destinations, séparés par des virgules. Chaque sommet et chaque arête sont séparés par un saut de ligne, et les chaque partie est séparée par le caractère ASCII terminal $.