Projet de graphe et réseaux : gogol-car

Samuel Buchet Dorian Dumez

Nomvembre 2016

1 Introduction

Commandes de compilations :

- se placer dans le dossier src pour utiliser toutes les commandes suivantes
- pour compiler gogol s : javac gogol_s/gogol_car_s.java
- pour compiler gogol l : javac gogol_l/gogol_car_l.java
- pour compiler gogol xl (requiers gogol l) : javac gogol_xl/gogol_car_xl.java
- pour compiler le menu (requiers tous les autres) : javac main/Main.java
- pour tout compiler d'un coup : javac */*.java
- exécution depuis le menu : java main/Main

Les instances se situent dans le fichier instances. Donc voici les chemin pour y accéder depuis le menu :

- ./../instances/antCity.txt : ville avec un circuit eulerien qui posait probleme selon la numérotation
- ./../instances/euler_city.txt : ville qui contiens un circuit eulerien
- ./../instances/NantesPasEuler.txt : ville qui ne contiens pas un circuit eulerien
- ./../instances/test.txt : ville qui contiens un circuit eulerien
- ./../instances/Ville1.txt : la première ville fournie
- ./../instances/Ville2.txt : la deuxième ville fournie

Commande de génération de la javadoc (depuis le dossier src) : javadoc -d ../doc -encoding UTF8 -docencoding UTF8 -charset UTF8 *

2 Gogol s

2.1 Introduction

Cet algorithme doit fonctionner sur n'importe quelle ville connexe, il n'y a pas d'autre conditions.

Le but de cette procédure est de fournir un itinéraire qui permet de parcourir toutes les rues dans les 2 sens. Et cet itinéraire doit être le plus court possible, c'est à dire que l'on doit pas parcourir une rue inutilement.

2.2 Représentation

Nous utilisons un graphe représenté par liste de successeur. C'est à dire que pour chaque noeud on dispose de la liste des sommets auxquels il est connecté.

Mais dans ce graphe les sommets sont les rues et les arc des places. Une place connecte deux rue si et seulement si ces 2 rues ont pour extrémité cette place.

2.3 Algorithme

La première partie de l'algorithme est la génération du graphe pendant la lecture du fichier.
Dans ce dernier nous commençons par lire le nom de toutes les places. Pendant ce procédé on crée une table de hachage qui associe chaque place à l'ensemble des rues qui y débouchent (initialisé a l'ensemble vide).
Dans un second temps on lit chaque rue, son nom, et ses deux extrémités. On alloue donc un nouveau sommet avec une liste de successeur vide. Ensuite pour tous les sommets contenus dans les ensembles associés à ses deux extrémités on les ajoute à sa liste de successeur et lui est ajouté à la liste de successeur de tous ces sommets. On termine toujours cela par l'ajout de ce nouveau sommet aux ensembles de ses deux extrémités.
A la fin de l'algorithme on choisit, de manière arbitraire, le dernier sommet créé comme point d'entrée dans le graphe.

Une fois le graphe précédent généré un parcours en profondeur de celui ci nous fournit un itinéraire qui respecte les propriétés souhaitées.

2.4 Preuve et complexité

Pour la première partie nous utilisons une table de hachage, le coût amorti d'un accès étant constant nous le considérerons donc toujours comme tel. Ensuite pour les ensembles nous utilisons des listes chaînées donc l'ajout d'un élément

Algorithm 1 Création du graphe pour gogol s

```
1: table de hachage ( string , ensemble ) noeuds
2: for chaque places do
3:     ajouter le nom de la place associé à un ensemble vide à noeuds
4: end for
5: for chaque rue do
6:     créer un noeud n contenant cette rue
7:     for chaque rue x de noeuds[n.début] do
8:         n.ajouterSuccesseur(x)
9:         x.ajouterSuccesseur(n)
10:    end for
11:    noeuds[n.début].ajouter(n)
12:    for chaque rue x de noeuds[n.fin] do
13:        n.ajouterSuccesseur(x)
14:        x.ajouterSuccesseur(n)
15:    end for
16:    noeuds[n.fin].ajouter(n)
17: end for
18: retourner n
```

se fait en $O(1)$. Enfin pour les listes de successeurs on utilise aussi des listes chaînées.

Pour la suite on pose n le nombre de place et m le nombre de rue.
On commence par initialiser la table de hachage en $O(n)$. La suite consiste en l'ajout de tous les arcs, on remarquera qu'ils sont ajoutés deux fois. L'ajout d'un arc s'effectue en $O(1)$ car l'ajout se fait toujours en tête. En graphe complet comportant $O(m^2)$ arc, on peut dire que l'ajout de tous les arcs est en $O(m^2)$.

Ensuite on effectue un parcours en profondeur de manière tout à fait classique, donc en $O(m + m^2)$ car le graphe peut être complet.

L'algorithme dans son ensemble est donc en $O(n+m^2)$. On peut aussi noter que la complexité spatiale est en $O(n + m^2)$ car on utilise une table de hachage sur les places puis le graphe peut être complet donc comprendre m^2 arc, et le nombre de sommet est toujours m.

Les sommets du graphe étant les rues, le parcours en profondeur nous garantit que toutes les rues sont parcourues une unique fois. Mais l'itinéraire indique de parcourir la rue dans un sens quand on arrive sur ce noeud et dans l'autre sens quand tous les appels récursifs sont terminé. Donc sur un noeud pour lequel il n'y a pas de successeur à visiter on parcourt la rue dans les 2 sens directement. Pour les autres, d'autres rues seront parcourues avant que le sens retour ne soit effectué.
De plus on est toujours certain que cet itinéraire peut être suivi car deux rues

ne sont reliées par un arc que si elles ont une place en commun.

3 Gogol l

3.1 Introduction

Dans la variante gogol l, on considère que le graphe fourni est eulerien. Il faut donc calculer l'itinéraire en sachant que la gogol car n'a besoin que d'un passage dans une rue pour en photographier l'intégralité. Le programme doit donc calculer un circuit eulerien dans le graphe de la ville. Pour ce faire, l'algorithme fourni dans le sujet a été utilisé.

3.2 Représentation

Dans cette variante, le graphe est représenté en mémoire par une liste de successeurs. Bien que le graphe est non orienté, il est représenté par un graphe orienté pour les besoins de l'algorithme. Cette représentation a été choisie car certains calculs de l'algorithme se font sur les arcs du graphe. De plus, il n'est presque jamais nécessaire d'accéder à un arc particulier, l'accès direct aux arcs n'est donc pas nécessaire.

3.3 Algorithmes

3.4 Déterminer si le graphe est eulerien

L'algorithme consistant à déterminer si le graphe est eulerien, ou non, est très simple. En effet, les degrés des sommets sont calculés lors de la création du graphe et l'algorithme consiste donc à parcourir l'ensemble des sommets et à vérifier le degré. Si le degré d'au moins un sommet n'est pas pair, alors le graphe n'est pas eulerien.

3.4.1 Arborescence

Le calcul de l'arborescence recouvrante du graphe est un simple parcours en profondeur. L'arborescence est gérée par une structure de donnée récursive. Cette structure contient une liste de pointeurs vers les arborescences fils et un pointeur vers l'arborescence père. Ainsi, elle permet de se déplacer de père en fils mais également de fils en père. Cela nous permet de créer explicitement la structure de donnée de l'anti-arborescence. En effet, il suffit de travailler sur l'arborescence en considérant que le père est désormais l'unique fils et que les fils sont les pères du sommet.

L'arborescence est créée par un parcours en profondeur, comme vu en cours. Ainsi, tant que tous les sommets ne sont pas présents dans l'arborescence, on

itère sur chaque fils du dernier sommet ajouté et une sous arborescence est récursivement créee sur ce sommet puis ajoutée en fils de l'arborescence actuelle.

Algorithm 2 Création d'une arborescence recouvrante

```
1: procedure CRÉER ARBORESECENCE(Graphe G) visités : Liste de booléens
2:     for Chaque sommet s de G do
3:         visité[s] ← faux
4:     end for
5:     for Chaque sommet s de G do
6:         if non visité[s] then
7:             creerArborescenceRec(s, visités)
8:         end if
9:     end for
10: end procedure

11: function CRÉERARBORESECENCEREC(Graphe G, Sommet s, liste de
    booléens visités)
12:     Arborescence arbo ← nouvelleArborescence(s)
13:     visités[s] ← vrai
14:     for Chaque successeurs s' de s dans G do
15:         if non visités[s] then
16:             ajouterFils(arbo, creerArborescenceRec(G, s', visités))
17:         end if
18:     end for
19:     return arbo
20: end function
```

3.4.2 Numérotation

La numérotation des arêtes a été réalisée comme indiqué dans le sujet du projet. Des règles ont été ajoutées pour numéroter les arcs présents dans l'anti arborescence avec les plus grands numéros, comme discuté en TP. Ainsi, l'arc de l'anti-racine prend systématiquement le plus grand numéro. Les autres arcs appartenant à l'anti-arborescence prennent les plus grands numéros possibles. Enfin, les autres sommets sont numérotés de manière quelconque.

Les numéros des arcs sont présents dans le chaînage de la liste des successeurs. Cela donne donc un accès direct vers les numéros lorsque l'on itère sur les successeurs.

L'algorithme de numérotation a été réalisé de manière récursive. En effet, la numérotation nécessite de disposer d'un pointeur vers le sommet de arborescence et ce pointeur est ainsi obtenu en effectuant un parcours en profondeur de l'arborescence.

Algorithm 3 Numérotation du graphe

```
    procedure NUMÉROTER(Graphe G, Arborescence arbo)
 2:     sommet s ← arbo.sommet
        entier numMin ← 1
 4:     entier numMax ← G.degréSortant(s)
        if arbo.pere() ≠ null then
 6:         numMax ← numMax-1
        end if
 8:     for chaque successeur arbo' de arbo do
            if arbo'.sommet() = arbo.pere() then
10:             G.numéroter(s, arbo'.sommet, G.degréSortant(s))
            else if arbo' ∈ arbo.fils() then
12:             G.numéroter(s, arbo'.sommet, numMax)
                numMax ← numMax-1
14:         else
                G.numéroter(s, arbo'.sommet, numMin)
16:             numMin ← numMin+1
            end if
18:     end for
        for chaque fils arbo' de arbo do
20:         Numéroter(arbo')
        end for
22: end procedure
```

3.4.3 Calcul du cycle eulerien

Le calcul du cycle eulerien a été encore une fois réalisé à partir de l'algorithme donné en cours. Cependant, dans cet algorithme, il est nécessaire de marquer des arcs comme étant parcourus. Pour ce faire, un booléen de marquage est conservé dans chaque maillon du chaînage de la liste des successeurs. Cela permet de marquer un arc avec une complexité constante à partir du moment où l'on a obtenu le sommet avec la liste des successeurs. Cependant, marquer l'arc inverse est plus difficile car il faut parcourir tout le chaînage à partir du sommet successeur.

3.5 Complexité

3.5.1 Construction de l'arborescence

Comme dit précédemment, l'arborescence est construite à partir d'un parcours en profondeur. Cet algorithme parcourt donc chaque sommet et pour un sommet donné, il parcourt au plus tous les successeurs de ce sommet. La complexité est donc de $O(|S| + |A|)$ où S les l'ensemble des arêtes du graphe et A l'ensemble des arcs, donc en $O(n + m)$.

Algorithm 4 Numérotation du graphe

```
    function CYCLEEULERIEN(Graphe G, Sommet racine)
2:      sommet s ← racine
        liste de place chemin ← G.nomPlace(racine)
4:      while Il existe un arc sortant de s non marqué do
            s' ← successeur qui est associé au plus petit numéro sur un arc non
    marqué
6:          G.marquerArc(s, s')
            G.marquerArc(s', s)
8:          chemin.ajouter(G.nomPlace(s'))
        end whilereturn chemin
10: end function
```

3.6 Numérotation des arcs

La numérotation des arcs consiste cette fois en un parcours en profondeur de l'arborescence. Étant donné que l'arborescence contient un nombre d'arc égal au nombre de sommet -1, cette partie de l'algorithme a donc une complexité en $O(n)$. Ensuite, pour chaque sommet de l'arborescence et donc du graphe, l'ensemble des successeurs sont parcourus. L'algorithme utilise les degrés sortants des sommets mais cette information est calculée au début du programme. De plus, comme expliqué précédemment, la numérotation d'un arc est en temps constant car le numéro est contenu dans le chaînage et c'est le chaînage qui est parcouru. La complexité de l'algorithme est donc encore une fois $O(|S| + |A|)$ donc $O(n + m)$.

3.7 Calcul du cycle eulerien

Le calcul de cycle eulerien consiste en un parcours du graphe en suivant les plus petits numéros des arcs. Tous les sommets sont ainsi parcourus et pour chaque sommet, l'ensemble des arcs sont parcourus. On a donc encore une fois une complexité de $O(|S| + |A|)$. Cependant, bien que le marquage de l'arc sortant soit direct, comme expliqué plus haut, le marquage de l'arc inverse implique de parcourir les successeurs du sommet de l'arc. Cette opération est réalisée à la fin du parcours des arcs du sommets, une fois que le sommet de l'arc du plus petit numéro est trouvé. La complexité est donc encore une fois de $O(|S| + |A|)$.

4 Gogol XL

4.1 Introduction

Dans la variante gogol XL, la gogol car peut encore une fois photographier les rues en un seul passage. Cependant, le graphe n'est, cette fois-ci, pas forcément eulerien. L'algorithme proposé dans le graphe ne s'applique donc plus.

L'idée de départ, prise à partir de recherches sur le postier chinois, consiste à rendre le graphe eulerien en dédoublant des arcs. On commence donc d'abord par identifier les sommets de degré impairs (c'est à dire ceux qui rendent le graphe non eulérien), puis ces sommets sont deux à deux reliés par un chemin en dédoublant les arcs. En faisant cela, les sommets des chemins restent de degré pair car on ajoute deux arcs.

La difficulté consiste donc à déterminer les sommets de degré impairs à relier entre eux de façon à dédoubler le moins d'arête possible. C'est donc un problème de couplage de poids minimum. Étant donnée que l'algorithme exacte n'était pas demandé, nous avons implémenté une simple heuristique pour ce problème. Le couplage obtenu n'est donc pas optimal.

4.2 Représentation mémoire

Pour cette partie, les algorithmes implémentés dans gogol l ont été réutilisés. De ce fait, la représentation mémoire est la même que celle de gogol l. Cependant, pour les besoins de l'algorithme de plus court chemin, une représentation comme matrice d'adjacence est aussi utilisée.

4.3 Algorithmes

4.3.1 Calcul des sommets de degrés impairs

Calculer les sommets de degrés impairs est très simple car, comme pour gogol l, les degrés des sommets sont calculés et stockés en mémoire à la création du graphe. L'algorithme consiste à parcourir tous les sommets et à ajouter dans une liste ceux pour lesquels le degré n'est pas pair. La complexité est donc de $O(n)$.

4.3.2 Couplage des sommets

Avant de coupler les sommets on doit avoir la distances minimale entre tous les sommets de degré impair. Pour avoir cela on calcule la distance minimale entre toutes les paires de sommet grâce à l'algorithme de Floyd-Warshall. Mais ce dernier utilise une représentation d'un graphe sous la forme d'une matrice d'adjacence. Pour l'avoir on parcourt tous les sommets et tous les arcs pour noter quand il y en a un entre deux sommets. De plus on a besoin du trajet calculé par floyd-warshall pour ensuite pourvoir dédoubler les arcs, donc on utilise la matrice successeur qui nous indique le chemin à suivre. Cette dernière est initialisée en même temps que la création de la matrice d'adjacence.

Niveau complexité Floyd-Warshall est en $O(m^3)$ et la transformation du graphe sous forme d'une matrice d'adjacence en $O(m^2)$ car chaque sommet peu avoir au plus m successeur si le graphe est complet. La complexité spatiale est en $O(m^2)$ car on ne travaille que sur des matrices $m \times m$.

Algorithm 5 transformation d'un graphe en matrice d'adjacence

```
1: soit matriceAdjacence une matrice m × m
2: soit successeur une matrice m × m
3: for i de 1 à m do
4:     for j de 1 a m do
5:         matriceAdjacence[i] [j] = +∞
6:         successeur[i] [j] = −1
7:     end for
8:     matriceAdjacence[i] [i] = 0
9: end for
10: for i de 1 à m do
11:     for  j ∈ successeur de i do
12:         matriceAdjacence[i] [j] = 1
13:         successeur[i] [j] = j
14:     end for
15: end for
```

Une fois le calcul des distances entre les sommets impairs effectué, l'algorithme se contente de parcourir toutes les distances et sélectionne la distance minimale à chaque fois. Le processus est répété tant que tous les couples n'ont pas été formés. Le double parcours des distances pour trouver celle qui est minimum est donc effectué i/2 fois, avec i le nombre de sommet de degré impair. La complexité est donc de $O(i^3)$ ou encore $O(n^3)$ en pire cas.

Il a également été songé de trier les distances au préalable mais cela ne facilite pas l'implémentation car une fois qu'un couple est choisi, il faut parcourir les couples à choisir pour supprimer ceux qui contiennent un des sommets choisi.

4.3.3 Dédoublement des arêtes

Maintenant que l'on a un couplage des sommets de degré impair et le chemin le plus court entre eux on va dédoubler les arêtes le long des chemins entre les sommets d'un même couple.

Algorithm 6 Dedoublage des arcs

```
1: for couple (premier,deuxième) du couplage  do
2:     parcrec = premier
3:     while parcrec ≠ successeur[parcrec] [deuxième] do
4:         precrec = parcrec
5:         parcrec = successeur[parcrec][deuxième]
6:         ajouter l'arête (precrec , parcrec)
7:     end while
8:     ajouter l'arête (precrec , deuxième)
9: end for
```

Le couplage n'étant pas optimal nous n'avons pas beaucoup d'hypothèses sur celui ci, on peut juste dire qu'une chemin a, au plus, une longueur de m. De plus on sait qu'il y a au plus $\frac{n}{2}$ couples. De plus on notera que l'ajout d'arêtes se fait en $O(m)$ car il faut retrouver les deux sommets extrémité ainsi que l'arc initial pour avoir le nom de la rue correspondante. Cet algorithme est donc en $O(n.m^2)$ même si ce cas arrivera rarement car il n'y a pas forcement beaucoup de sommets de degré impair, et si il en a beaucoup les chemins devraient être court.

On sait qu'un graphe contient un circuit eulérien si et seulement si tous ses sommets sont degré pair. Or quand on duplique un chemin tous les sommets le constituant sont de degré pair (ou vont le devenir). Donc si on ajoute 2 à son degré il est toujours pair, et on ajoute toujours 2 car si il est le long du chemin il a un nouvel arc entrant et un nouvel arc sortant. Et si il est au bout on ne fait que lui ajouter 1 à son degré, or les extrémités du chemin sont des sommets de degré impair, donc ils ne le sont plus. Enfin on sait que l'on pourra toujours avoir un couplage de tous les sommets de degré impair car selon le lemme des poignées de main il y en a toujours un nombre pair.