# Rapport de projet d'Algorithmique

LAIGLE Yoan
LOPES Guillaume

20 Décembre 2006

# Table des matières

# 1 Présentation du projet et choix d'implémentation

Dans ce projet, nous avons implémenté les algorithmes dé Kosaraju et Tarjan vus en cours et en TD, afin de construire le graphe réduit des composantes fortement connexes.

## 1.1 Choix d'implémentation

Nous allons, tout d'abord, donner une descripton succinte des différentes classes du projet :

*class Sommet :*
Cette classe permet de représenter un sommet d'un graphe simple.Les champs de cette classe sont :

- un entier *numero*, qui représente le numéro du sommet
- un entier *dateDebut*,qui représente la date de première visite du sommet
- un entier *dateFin*,qui représente la date de dernière visite du sommet
- un entier *lowlink*,qui représente le sommet de plus court chemin
- un booleen *surpile*, qui permet de savoir si un sommet est sur la pile
- un string *couleur*,qui représente la couleur du sommet

*class GrapheMatrice :*
Cette classe ne possède qu'un seul champ aretes, qui est une Arraylist de Arraylist d'entier. Ce champ permet de représenter la matrice d'adjacence d'un graphe. On a utilisé la classe prédéfinie ArrayList en JAVA pour la représenter. En effet, cette classe est simple d'utilisation et ainsi on respecte le cahier des charges de l'énoncé, qui veut que le code du programme soit objet, c'est à dire que les expressions de type *tab[u]* soit remplacé par des expressions du type *u.tab*

*class GrapheListes :*
Cette classe ne possède qu'un seul champ aretes, qui est une ArrayList de ArrayList de sommets. Ce champ permet de représenter la liste d'adjacence d'un graphe. On utilisé aussi la classe prédéfinie ArrayList,pour les mêmes raisons que citées ci-dessus.

*abstract class Graphe :*
Cette classe permet de représenter un graphe simple. Elle possède les champs suivants :

- Une ArrayList de Sommet *sommets*, qui représente la liste des sommets du graphe

- Une ArrayList d'entier *CFC*, qui représente l'appartenance d'un Sommet à une composante fortement connexe
- Un entier *nombreComposante*, qui représente le nombre de composantes fortement connexes dans le graphe
- Un entier *compteurFin*, qui représente le compteur de date de fin
- Une pile de Sommet *pile*, qui représente la pile ou sont stockés les sommets du graphe

Nous avons créé une petite interface graphique, mais nous ne la traiterons pas ici car son explication est fastidieuse et ne présente aucun intérêt pour ce projet.

## 1.2 Fonctionnement du projet

Nous présentons dans cette partie une brève description du fonctionnement du programme.

Qu'est ce qui se passe lorsque l'utilisateur lance le programme ?
On lui demande, dans l'ordre :

1. Comment sera représenté le graphe, c'est à dire soit par des matrices, soit par des listes
2. Le nombre de sommets qu'il désire
3. Ajouter les transitions, une à une du graphe
4. Lancer l'algorithme de Kosaraju ou de Tarjan sur le graphe qu'il vient de créer

Noter bien qu'à tout moment, l'utilisateur peut revenir en arrière !
Une fois que l'un des deux algorithmes à été choisi, l'utilisateur voit apparaitre à l'écran, les informations suivantes :

- Le nombre de composantes du graphe
- Le numéro de composante de chaque sommet
- Le graphe réduit

Qu'est ce l'utilisateur ne peut pas pas faire ?

- Rentrer des chaines de caractères à la place d'entiers
- Rentrer des sommets qui n'existent pas dans le graphe, au moment des ajouts de transitions
- Rentrer deux fois le même arc
- Rentrer un arc qui contredit la notion de graphe simple

Lorsque l'utilisateur effectue l'une des actions ci-dessus, un message d'erreur précis lui indique l'erreur qu'il a commise et bloque le programme tant que l'utilisateur ne rentre pas de données correctes.

# 2 Complexité des algorithmes

Nous allons présenter dans cette partie la complexité des différentes fonctions, qui permettent de trouver la complexité de Kosaraju et Tarjan.
Dans la suite de cette partie, $n$ représentera le nombre de sommets du graphe et $m$ le nombre d'arcs du graphe.

## 2.1 Complexité de l'algorithme de Kosaraju

Voici la liste des fonctions dont la complexité varie selon la représentation :

- *transpose()*
- *voisins(Sommet s)*

### 2.1.1 GrapheMatrice

Commençons par montrer que *transpose()* est en $O(n^2)$.

```
public void transpose(){
```

*//Ces trois instructions s'effectuent en O(1).*

```
int nbSommets = this.getN();
ArrayList<ArrayList<Integer>> transGraphe =
new ArrayList<ArrayList<Integer>> (nbSommets);
```

*//Cette boucle est en O(n)*

```
for(int i=0;i<nbSommets;i++){
transGraphe.add(i,new ArrayList<Integer> (nbSommets));
}
```

*//Cette boucle est en $O(n^2)$.*

```
for(int i=0;i<nbSommets;i++){
for(int j =0;j<nbSommets;j++){
transGraphe.get(i).add(aretes.get(i).get(j));
}
}
```

*//Cette boucle est en $O(n^2)$.*

```
for(int i=0;i<nbSommets;i++){
for(int j = 0;j<nbSommets;j++){
int tmp = aretes.get(j).get(i);
transGraphe.get(j).add(i, aretes.get(i).get(j));
```

```
transGraphe.get(i).add(j,tmp);
}
}
```

*//Cette instruction s'effectue en O(1).*

```
this.setAretes(transGraphe);
}
```

La fonction *transpose()*, pour un GrapheMatrice, s'effectue donc bien en $O(n^2)$.

Démontrons que *voisins(Sommet s)* s'effectue en O(n).

```
public Iterator<Sommet> voisins(Sommet u) {
```

*//Ces trois intructions s'effectuent en O(1).*

```
ArrayList<Sommet> voisinsU =
new ArrayList<Sommet>();
int nbSommets = this.getN();
int numU = u.getNumero();
```

*//Cette boucle s'effectue en O(n).*

```
for(int i =0;i<nbSommets;i++){
if(aretes.get(numU).get(i) == 1){
voisinsU.add(this.getSommet(i));
}
}
return voisinsU.iterator();
}
```

La fonction *voisins* s'effectue donc en O(n) pour un GrapheMatrice.

### 2.1.2 GrapheListes

Commençons par montrer que *transpose()* est en O(n+m).

```
public void transpose(){
```

*//Ces trois instructions s'effectuent en O(1).*

```
int nbSommets = this.getN();
ArrayList<ArrayList<Sommet>> transGraphe =
new ArrayList<ArrayList<Sommet>> (nbSommets);
Iterator it;
```

*//Cette boucle s'effectue en O(n).*

```
for(int i =0;i<nbSommets;i++){
transGraphe.add(i,new ArrayList <Sommet> ());
}
```

*//Cette boucle s'effectue donc en O(n+m).D'après la démonstration vue en TD.*

```
for(int i =0;i<nbSommets;i++){
it = aretes.get(i).iterator();
```

*//Ce while s'effectue en O(m).*

```
while(it.hasNext()){
Sommet s = (Sommet)(it.next());
transGraphe.get(s.getNumero()).add(this.getSommet(i));
}
}
```

*//Cette dernière instruction s'effectue en O(1).*

```
this.setAretes(transGraphe);
}
```

La fonction *transpose()*, pour un GrapheListe, s'effectue donc en O(n+m).

Démontrons que *voisins(Sommet s)* s'effectue en O(1).
*//Cette fonction est trivialement en O(1).*

```
public Iterator<Sommet> voisins(Sommet u) {
int numU = u.getNumero();

return aretes.get(numU).iterator();
}
```

La fonction *voisins(Sommet s)* s'effectue en O(1) pour un GrapheListe.

Etudions maintenant la complexité de l'algorithme de parcours en profondeur de Kosaraju selon la représentation choisie :

```
public final void ParcoursProfondeurKosaraju(Sommet s){
```

*//Ces trois instructions s'effectuent en O(1)*

```
int numero = s.getNumero();
this.CFC.set(numero,nombreComposante);
s.setCouleur("gris");
//Cette instruction s'effectue en O(n) pour les GrapheMatrices
//Cette instruction s'effectue en O(1) pour les GrapheListes

Iterator adj = this.voisins(s);
//Cette boucle s'éxecute m fois au pire donc, elle est en O(m)
while(adj.hasNext()){
Sommet w = (Sommet)adj.next();
if(w.getCouleur().compareTo("blanc") == 0){
this.ParcoursProfondeurKosaraju(w);
}
}

s.setDateFin(compteurFin);
compteurFin++;

}
```

L'algorithme de parcours en profondeur de Kosaraju s'effectue en O(n+m) pour les deux représentations.

Démontrons maintenant la complexité finale de l'algorithme de Kosaraju selon la représentation

```
public final void Kosaraju(){
//Ces quatres instructions s'effectuent en O(1)
this.nombreComposante=this.compteurFin = 0;
int numSommets = this.getN();
ArrayList<Sommet> post = new ArrayList<Sommet>(numSommets);
Iterator sommetsGraphe = this.getSommets();
//Cette boucle while s'effectue en O(n)
while(sommetsGraphe.hasNext()){
Sommet s =((Sommet)(sommetsGraphe.next()));
s.setDateFin(-1);
s.setCouleur("blanc");
post.add(new Sommet(0));
}
```

*//Cette instruction s'effectue en O(1)*

```
sommetsGraphe = this.getSommets();
```

*Cette boucle while s'effectue en O(n+m)*

```
while(sommetsGraphe.hasNext()){
Sommet s = (Sommet)sommetsGraphe.next();
if(s.getDateFin() == -1){
this.ParcoursProfondeurKosaraju(s);
}
}
```

*le calcul de la transposée s'effectue en :*
*//O($n^2$) pour les GrapheMatrices*
*//O(n+m) pour les GrapheListes*

```
this.transpose();
```

*Ces deux instructions s'effectuent en O(1)*

```
System.out.println("La matrice transpose");
sommetsGraphe = this.getSommets();
```

*//Cette boucle while s'effectue en O(n)*

```
while(sommetsGraphe.hasNext()){
Sommet s = (Sommet)sommetsGraphe.next();
post.set(s.getDateFin(),s);
}
```

*Cette instruction s'effectue en O(1)*

```
sommetsGraphe = this.getSommets();
```

*//Cette boucle while s'effectue en O(n)*

```
while(sommetsGraphe.hasNext()){
Sommet s = ((Sommet)(sommetsGraphe.next()));
s.setDateFin(-1);
s.setCouleur("blanc");
}
```

*Cette instruction s'effectue en O(1)*

```
this.nombreComposante = this.compteurFin = 0;
```

*//Cette boucle for s'effectue en O(n+m)*

```
for(int date = (numSommets-1);date >=0;date--){
Sommet s = post.get(date);
if(s.getDateFin() == -1){
this.ParcoursProfondeurKosaraju(s);
nombreComposante++;
}
}
```

*le calcul de la transposée s'effectue en :*
*//$O(n^2)$ pour les GrapheMatrices*
*//O(n+m) pour les GrapheListes*

```
this.transpose();
}
```

*Donc la complexité de l'algorithme de Kosaraju est en :*
*//$O(n^2)$ pour les GrapheMatrices*
*//O(n+m) pour les GrapheListes*

## 2.2 Complexité de l'algorithme de Tarjan

Etudions la complexité de *Tarjan(Sommet s)*

```
public final void Tarjan(Sommet s){
```

*//Ces cinq instructions s'effectuent en O(1)*

```
int numero=0;
s.setDateDebut(compteurFin++);
pile.push(s);
s.setSurpile(true);
s.setLowlink(s.getDateDebut());
```

*//Cette instruction s'effectue en :*
*//O(n) pour les GrapheMatrices*
*//O(1) pour les GrapheListes*

```
Iterator sommets = this.voisins(s);
```

*//Cette boucle est effectuée n fois*

```
while(sommets.hasNext()){
Sommet v = (Sommet)sommets.next();
if(v.getDateDebut()==-1){
Tarjan(v);
}
```

*//Cette instruction s'effectue en O(1)*

```
if(v.getSurpile()&&(s.getLowlink()>v.getLowlink())){
s.setLowlink(v.getLowlink());
}
}
```

*//Cette boucle est effectuée au pire m fois*

```
if(s.getDateDebut()==s.getLowlink()){
do{
Sommet v = pile.pop();
v.setSurpile(false);
numero = v.getNumero();
this.CFC.set(numero,nombreComposante);
}while(numero!=s.getNumero());
nombreComposante++;
}

}
```

Etudions maintenant la complexité de l'algortihme de *Tarjan* :

```
public final void Tarjan(){
```

*//Ces trois instructions s'effectuent en O(1)*

```
pile = new Stack<Sommet>();
nombreComposante=compteurFin=0;
Iterator sommetsGraphe = this.getSommets();
```

*//Cette boucle while s'effectue en O(n)*

```
while(sommetsGraphe.hasNext()){
Sommet s =((Sommet)(sommetsGraphe.next()));
s.setDateFin(-1);
s.setLowlink(-1);
s.setDateDebut(-1);
s.setSurpile(false);
}
```

*//Cette instruction s'effectue en O(1)*

```
sommetsGraphe = this.getSommets();
```

*//Cette boucle while est effectuée n fois*
*//donc la complexité de cette boucle est en :*
*//O($n^2$) pour les GrapheMatrices*
*//O(n+m) pour les GrapheListes*

```
while(sommetsGraphe.hasNext()){
Sommet s = (Sommet)sommetsGraphe.next();
if(s.getDateDebut() == -1){
Tarjan(s);
}
}
}
```

Donc la complexité de l'algorihme de Tarjan est en :
O($n^2$) pour les GrapheMatrices
O(n+m) pour les GrapheListes