

Victor Vuillard – Thomas Petazzoni

---

# Algorithmes sur les graphes

## *Rapport Travaux Pratiques 2 – AG51*

---

Semestre d'automne 2002-2003

Université de Technologie  
de Belfort Montbéliard

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Représentation des graphes en Java</b>	<b>3</b>
1.1 Interface <b>Graphe</b>	3
1.2 Deux représentations des graphes	4
1.2.1 La représentation par matrice d’adjacences	4
1.2.2 Représentation par listes d’adjacences	5
1.2.3 Graphes non orientés	6
1.3 Code source	6
1.3.1 Graphe orienté par matrice d’adjacence	6
1.3.2 Graphe orienté par listes chaînées	10
1.3.3 Graphe non orienté par matrice d’adjacence	13
1.3.4 Graphe non orienté par listes chaînées	14
<b>2 Algorithmes</b>	<b>14</b>
2.1 Parcours en largeur	15
2.2 Recherche du plus court chemin	16
2.3 Parcours en profondeur	17
2.4 Recherche des composantes fortement connexes	17
2.5 Code source	19
<b>3 Analyse temporelle des algorithmes</b>	<b>22</b>
3.1 Parcours en largeur d’un graphe orienté	23
3.1.1 Mesures expérimentales	23
3.1.2 Analyse théorique	24
3.2 Recherche du plus court chemin	24
3.2.1 Mesures expérimentales	24
3.2.2 Analyse théorique	25
3.3 Parcours en profondeur	26
3.3.1 Analyse théorique	26
3.4 Recherche des composantes fortement connexes	27
3.4.1 Mesures expérimentales	27
3.4.2 Analyse théorique	28

# Introduction

Les objectifs du TP étaient :

- Réaliser des fonctions de manipulation des graphes (en utilisant au choix une représentation par matrice d'adjacences, FS/APS ou listes d'adjacences).
- Implémenter les algorithmes de parcours en largeur, parcours en profondeur, recherche du plus court chemin et recherche des composantes fortement connexes.
- Mesurer les couts de ces algorithmes de manière expérimentale et comparer les résultats obtenus avec les valeurs théoriques.

## Notations

Dans l'ensemble du rapport, on notera :

- $G$  un graphe
- $S$  l'ensemble des sommets du graphe
- $A$  l'ensemble des arcs du graphe

## 1 Représentation des graphes en Java

### 1.1 Interface Graphe

Etant donné que nous pouvions potentiellement réaliser plusieurs implémentations des graphes, en utilisant les différentes représentations possibles, nous avons décidé d'écrire une *interface Java Graphe*, listant les méthodes à implanter dans chaque classe permettant de représenter les graphes.

Voici le code de cette interface :

Listing 1 – Interface Graphe

```
interface Graphe
{
    public boolean est_vide();
    public boolean est_arc(int s1, int s2);
    public int     degre_plus(int s);
    public int     degre_moins(int s);
    public void    ajouter_arc(int s1, int s2);
    public void    supprimer_arc(int s1, int s2);
    public int     ieme_successeur(int s, int i);
    public int     ieme_predecesseur(int s, int i);
    public int     premier_successeur(int s);
    public int     successeur_suivant(int s1, int s2);
}
```

<pre> public int      getNbSommet(); } </pre>
---

**est\_vide** Indique si le graphe est vide ou non.

**est\_arc** Indique si il existe un arc entre deux sommets

**degre\_plus** Donne le nombre de sommets auxquels on peut accéder depuis le sommet donné.

**degre\_moins** Donne le nombre de sommets qui permettent d'accéder au sommet donné.

**ajouter\_arc** Ajoute un arc entre deux sommets donnés.

**supprimer\_arc** Supprime un arc entre deux sommets donnés.

**ieme\_successeur** Donne le  $i^{eme}$  successeur d'un sommet donné.

**ieme\_predecesseur** Donne le  $i^{eme}$  predecesseur d'un sommet donné.

**premier\_sucesseur** Donne le premier successeur d'un sommet donné.

**successeur\_suivant** A partir d'un sommmet et d'un successeur de ce sommet, cette fonction retourne le successeur suivant du sommet.

**getNbSommet** Retourne le nombre de sommets du graphe.

Les algorithmes de parcours en largeur, de parcours en profondeur, de recherche de plus court chemin et de recherche des composantes fortement connexes seront implémentés en utilisant uniquement les méthodes de l'interface **Graphe**. Ceci permettra de tester très simplement ces différents algorithmes sur les différentes représentations des graphes.

## 1.2 Deux représentations des graphes

Deux représentations des graphes ont été implémentées :

- Par matrice d'adjacences
- Par liste d'adjacences

Les deux représentations seront exposées pour des graphes orientés. L'implémentation des graphes non orientés sera exposée plus loin.

### 1.2.1 La représentation par matrice d'adjacences

La représentation par matrice d'adjacences d'un graphe  $G = (S, A)$  consiste en une matrice  $|S| \times |S|$ ,  $M = (a_{ij})$ , telle que :

$$a_{ij} = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{sinon} \end{cases}$$

La matrice d'adjacences d'un graphe nécessite donc une quantité de mémoire en  $\Theta(S^2)$ . Cette représentation des graphes est donc pratique pour des petits graphes, mais devient très couteuse en mémoire pour des grands graphes. Pour des grands graphes *peu denses*, c'est à dire comportant peu d'arcs, cette représentation est très inefficace, car une grande partie de la matrice d'adjacences sera remplie de 0. Enfin, cette représentation est peu avantageuse lorsque l'on doit ajouter ou supprimer des sommets (réallocation et recopie de l'ensemble de la matrice).

La classe `GrapheMatrix` implémente toutes les méthodes de l'interface `Graphe`, en utilisant une représentation par matrice d'adjacences. La matrice est représentée en mémoire par un tableau bi-dimensionnel d'entiers (`int [][]`). Le constructeur de la classe `GrapheMatrix`, prend un entier en paramètre, qui fixe le nombre de sommets du graphe. Dans l'implémentation réalisée, ce nombre ne peut pas changer, mais cela serait réalisable très simplement.

### 1.2.2 Représentation par listes d'adjacences

La représentation par listes d'adjacences d'un graphe  $G = (S, A)$  consiste en un tableau  $l$  de  $|S|$  listes, une pour chaque sommet de  $G$ . Pour chaque  $s \in S$ , la liste d'adjacences  $l[s]$  est une liste des sommets  $t$ , tels que l'arc  $(s, t) \in A$ . Autrement dit,  $l[s]$  est constituée de tous les sommets adjacents à  $s$  dans  $G$ . Cette représentation est nettement plus avantageuse que la précédente dans le cas de graphes *peu denses*, et ne consomme qu'une quantité de mémoire en  $O(S + A)$ . Toutefois, pour déterminer si un arc donné est présent dans le graphe, il n'existe pas d'autre moyen que de parcourir la liste d'adjacence associée au sommet d'origine de l'arc.

La classe `GrapheListeChaine` implémente toutes les méthodes de l'interface `Graphe`, en utilisant une représentation par liste d'adjacences. Les listes chaînées sont représentées par des `LinkedList`, classe fournie par le JDK de Sun. On utilise donc un tableau de listes chaînées pour représenter le graphe (`LinkedList []`). Le constructeur de la classe `GrapheListeChaine` prend là aussi un entier en paramètre, qui fixe le nombre de sommets du graphe. La possibilité d'ajouter et de supprimer des sommets n'a pas été implémentée.

### 1.2.3 Graphes non orientés

Enfin, les classes `GrapheMatrixNonOriente` et `GrapheListeChaineNonOriente` permettent à partir des deux classes précédentes d'obtenir une implémentation pour les graphes non orientés. L'idée est très simple : un graphe non orienté est simplement un graphe orienté dans lequel on ajoute 2 arcs au lieu d'un. Par exemple si on souhaite ajouter un arc entre  $s1$  et  $s2$  sur un graphe non orienté, il suffit d'ajouter dans le graphe orienté correspondant un arc entre  $s1$  et  $s2$  et un autre arc entre  $s2$  et  $s1$ . Ces deux classes sont donc implémentées en héritant des classes `GrapheMatrix` et `GrapheListeChaine` et en surchargeant les méthodes `ajouter_arc` et `supprimer_arc`.

## 1.3 Code source

### 1.3.1 Graphe orienté par matrice d'adjacence

Listing 2 – Graphe orienté par matrice d'adjacence

```
public class GrapheMatrix implements Graphe
{
    private int nb_sommet;
    private int ident;

    private int [][] matrix;

    public GrapheMatrix(int sommets)
    {
        nb_sommet = sommets;
        ident = 0;
        matrix = new int[sommets][sommets];
    }

    public int getNbSommet()
    {
        return nb_sommet;
    }

    public void ajouter_arc(int s1, int s2)
    {
        if(s1 < 0 || s2 < 0)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return;
        }
    }
}
```

```

        if ( est_arc(s1, s2) == true)
        {
            System.out.println("Cet arc existe déjà");
            return;
        }

        matrix[s1][s2] = 1;
    }

    public void supprimer_arc(int s1, int s2)
    {
        if(s1 < 0 || s2 < 0)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return;
        }

        if(est_arc(s1, s2) == false)
        {
            System.out.println("L'arc n'appartient pas au graphe");
            return;
        }

        matrix[s1][s2] = 0;
    }

    public boolean est_vide()
    {
        return (nb_sommet == 0);
    }

    public boolean est_sommet(int s)
    {
        if(s > 0 && s < nb_sommet)
            return true;
        else
            return false;
    }

    public boolean est_arc(int s1, int s2)
    {
        int i;

        if(s1 < 0 || s1 > nb_sommet || s2 < 0 || s2 > nb_sommet)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return false;
        }
    }

```

```

        return (matrix[s1][s2] != 0);
    }

    public int degre_plus(int s)
    {
        int cpt = 0;

        if(s < 0 || s > nb_sommet)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return -1;
        }

        for(int i = 0; i < nb_sommet; i = i + 1)
        {
            if (matrix[s][i] != 0)
                cpt = cpt + 1;
        }
        return cpt;
    }

    public int degre_moins(int s)
    {
        int cpt = 0;

        if(s < 0 || s > nb_sommet)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return -1;
        }

        for(int i = 0; i < nb_sommet; i = i + 1)
        {
            if (matrix[i][s] != 0)
                cpt = cpt + 1;
        }
        return cpt;
    }

    public int ieme_successeur(int s, int num)
    {
        int i, cpt = 0;

        if(s < 0 || s > nb_sommet)
        {
            System.out.println("Ces sommets n'appartiennent pas au graphe");
            return -1;
        }
    }

```



```

    for(i = 0; i < nb_sommet; i++)
    {
        if (matrix[s][i] != 0 && cpt == num)
            return i;
        if (matrix[s][i] != 0)
            cpt++;
    }

    return -1;
}

public int ieme_predecesseur(int s, int num)
{
    int i, cpt = 0;

    if(s < 0 || s > nb_sommet)
    {
        System.out.println("Ces sommets n'appartiennent pas au graphe");
        return -1;
    }

    for(i = 0; i < nb_sommet; i++)
    {
        if (matrix[i][s] != 0 && cpt == num)
            return i;
        if(matrix[i][s] != 0)
            cpt++;
    }

    return -1;
}

public int premier_successeur(int s)
{
    return ieme_successeur(s, 0);
}

public int successeur_suivant(int s1, int s2)
{
    int i;

    if(s1 < 0 || s1 > nb_sommet || s2 < 0 || s2 > nb_sommet)
    {
        System.out.println("Ces sommets n'appartiennent pas au graphe");
        return -1;
    }

    for (i = s2 + 1; i < nb_sommet; i++)
    {

```

```

        if (matrix[s1][i] != 0)
            return i;
    }

    return -1;
}
}

```

### 1.3.2 Graphe orienté par listes chaînées

Listing 3 – Graphe orienté par listes chaînées

```

import java.util.*;

public class GrapheListeChaine implements Graphe
{
    LinkedList[] l;
    private int nbsommets;

    public GrapheListeChaine(int sommets)
    {
        l = new LinkedList[sommets];

        for (int i = 0; i < sommets; i++)
            l[i] = new LinkedList();

        nbsommets = sommets;
    }

    public boolean est_vide()
    {
        return (nbsommets == 0);
    }

    public boolean est_arc(int s1, int s2)
    {
        if (s1 < 0 || s2 < 0 || s1 > nbsommets || s2 > nbsommets)
            return false;

        for (int j = 0; j < l[s1].size(); j++)
        {
            if (((Integer) l[s1].get(j)).intValue() == s2)
                return true;
        }

        return false;
    }
}

```

```

public int degre_plus(int s)
{
    int i;

    for (i = 0; ieme_successeur(s, i) != -1; i++)
        ;

    return i;
}

public int degre_moins(int s)
{
    int i;

    for (i = 0; ieme_predecesseur(s, i) != -1; i++)
        ;

    return i;
}

/**
Ajoute l'arc de valuation <i>val</i> reliant <i>s1</i> à <i>s2</i>
*/
public void ajouter_arc(int s1, int s2)
{
    if(s1 < 0 || s1 > nbsommets || s2 < 0 || s2 > nbsommets)
        return;

    if(est_arc(s1, s2))
        return;

    l[s1].addFirst(new Integer(s2));
}

public void supprimer_arc(int s1, int s2)
{
    if(s1 < 0 || s1 > nbsommets || s2 < 0 || s2 > nbsommets)
        return;

    for (int i = 0; i < l[s1].size(); i++)
    {
        if(((Integer) l[s1].get(i)).intValue() == s2)
        {
            l[s1].remove(i);
            break;
        }
    }
}

```

```

public int ieme_successeur(int s, int cpt)
{
    if(s < 0 || s > nbsommets)
        return -1;

    for (int i = 0; i < l[s].size(); i++)
    {
        if(cpt == i)
            return ((Integer) l[s].get(i)).intValue();
        cpt++;
    }

    return -1;
}

public int ieme_predecesseur(int s, int ieme)
{
    int cpt = 0;

    for (int i = 0; i < nbsommets; i++)
    {
        for (int j = 0; j < l[i].size(); j++)
        {
            if ( ((Integer) l[i].get(j)).intValue() == s)
            {
                if(cpt == ieme)
                    return i;
                else
                    cpt++;
            }
        }
    }

    return -1;
}

public int premier_successeur(int s)
{
    if(s < 0 || s > nbsommets)
        return -1;

    if(l[s].size() > 0)
        return ((Integer) l[s].get(0)).intValue();
    else
        return -1;
}

public int successeur_suivant(int s1, int s2)

```

```

{
    if(s1 < 0 || s2 < 0 || s1 > nbsommets || s2 > nbsommets)
        return -1;

    for (int i = 0; i < nbsommets; i++)
    {
        for (int j = 0; j < l[s1].size(); j++)
        {
            if ( ((Integer) l[s1].get(j)).intValue() == s2)
            {
                if(j+1 == l[s1].size())
                    return -1;
                else
                    return ((Integer) l[s1].get(j+1)).intValue();
            }
        }
    }

    return -1;
}

public int getNbSommet()
{
    return nbsommets;
}
}

```

### 1.3.3 Graphe non orienté par matrice d'adjacence

Listing 4 – Graphe non orienté par matrice d'adjacence

```

public class GrapheMatrixNonOriente extends GrapheMatrix
{
    public GrapheMatrixNonOriente(int n)
    {
        super(n);
    }

    public void ajouter_arc(int s1, int s2)
    {
        super.ajouter_arc(s1, s2);
        super.ajouter_arc(s2, s1);
    }

    public void supprimer_arc(int s1, int s2)
    {
        super.supprimer_arc(s1, s2);
    }
}

```

```

    super.supprimer_arc(s2, s1);
}
}

```

### 1.3.4 Graphe non orienté par listes chaînées

Listing 5 – Graphe non orienté par listes chaînées

```

public class GrapheListeChaineNonOriente extends GrapheListeChaine
{
    public GrapheListeChaineNonOriente(int sommets)
    {
        super(sommets);
    }

    public void ajouter_arc(int s1, int s2)
    {
        super.ajouter_arc(s1, s2);
        super.ajouter_arc(s2, s1);
    }

    public void supprimer_arc(int s1, int s2)
    {
        super.supprimer_arc(s1, s2);
        super.supprimer_arc(s2, s1);
    }
}

```

## 2 Algorithmes

Tous les algorithmes ont été implantés sous forme de méthodes statiques dans la classe **GrapheAlgo**. Leur implémentation utilise uniquement les méthodes l'interface **Graphe**, sans s'appuyer sur des propriétés particulières de chacune des 2 représentations de graphes implémentées. Ceci permet d'utiliser le même code sur les différentes implémentations des graphes.

Les algorithmes implémentés n'affichent aucun message, et ne retournent aucun résultat : ils ont été écrits dans l'idée d'analyser leur efficacité temporelle sur de grands graphes. Ils ont été vérifiés et testés sur de petits graphes.

## 2.1 Parcours en largeur

Le parcours en largeur est l'algorithme de parcours des graphes le plus court, et est utilisé notamment dans les algorithmes de recherche de plus court chemin et de détermination de l'arbre couvrant minimum.

A partir d'un graphe  $G$ , et d'un sommet d'origine  $s$ , le parcours en largeur emprunte systématiquement les arcs de  $G$  pour découvrir tous les sommets accessibles depuis  $s$ . L'algorithme de parcours en largeur tire son nom du fait qu'il parcourt d'abord tous les sommets situés à une distance  $k$  de  $s$  avant de découvrir tout sommet situé à la distance  $k + 1$ .

L'algorithme utilise un tableau de booléens (**marque**) lui permettant de connaître les sommets ayant déjà été parcourus. L'algorithme présenté ne calcule pas le tableau des distances entre les sommets et l'origine, ni le tableau des parents. Un parcours en largeur modifié sera présenté pour la recherche du plus court chemin.

Listing 6 – Parcours en largeur

```
ParcoursLargeur(G,s)
  pour chaque sommet u de G faire
    marque[u] ← faux
  fin pour

  enfiler (F, s)

  tant que F non vide faire
    u ← tete(F)
    pour chaque successeur v de u faire
      si marque[v] = faux alors
        marque[v] = vrai
        enfiler (F,v)
      defile (F)
    fin si
  fpour
ftq
Fin
```

La méthode `parcours_largeur` de la classe `GrapheAlgo` implémente cet algorithme.

## 2.2 Recherche du plus court chemin

La recherche du plus court chemin dans un graphe non valué est très simple si l'on s'appuie sur le parcours en largeur. En effet, en partant d'un sommet donné, le parcours en largeur parcourt tous les sommets du graphe, du sommet le plus proche de l'origine au sommet le plus éloigné. La notion de distance est donc intimement liée au parcours en largeur.

Il suffit simplement de rajouter un tableau **distances**, qui contiendra les distances de chaque sommet à l'origine, et un tableau **parents**, qui donnera pour chaque sommet son parent dans le parcours en largeur. Pour connaître le plus court chemin d'un sommet  $a$  à un sommet  $b$ , il suffira donc de prendre la valeur **parents**[ $b$ ], puis la valeur **parents**[**parents**[ $b$ ]], etc.. jusqu'à remonter à  $a$ . Ceci donnera le chemin inverse du chemin le plus court pour aller de  $a$  à  $b$ .

Listing 7 – Recherche du plus court chemin

```
ParcoursLargeur(G,s)
  pour chaque sommet u de G faire
    marque[u] <- faux
    distances[u] <- -1
    parents[u] <- -1
  fin pour

  enfiler (F, s)

  tant que F non vide faire
    u <- tete(F)
    pour chaque successeur v de u faire
      si marque[v] = faux alors
        marque[v] = vrai
        distances[v] <- distances[u] + 1
        parents[v] <- u
        enfiler (F,v)
      defile (F)
    fin si
  fpour
ftq
Fin
```

La méthode `recherche_plus_court_chemin` de la classe `GrapheAlgo` implémente cet algorithme.



## 2.3 Parcours en profondeur

La stratégie suivie par le parcours en profondeur est de descendre plus profondément dans le graphe chaque fois que c'est possible. Lors d'un tel parcours, les arcs sont explorés à partir du sommet  $v$  découvert le plus récemment et dont on n'a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs de  $v$  ont été explorés, l'algorithme revient en arrière pour explorer les arcs qui partent du sommet à partir duquel  $v$  a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles depuis l'origine aient été découverts.

Comme pour le parcours en largeur, on utilisera un tableau de booléens (**marque**) permettant de connaître les sommets ayant déjà été visités. L'algorithme présenté ne calcule pas le tableau des dates de découverte des sommets, ni le tableau des parents.

Listing 8 – Parcours en profondeur

```
ParcoursProfondeur(G,s)
  pour chaque sommet u de G faire
    marque[u] ← faux
  fin pour

  marque[s] ← vrai
  _ParcoursProfondeur(s)
Fin

_ParcoursProfondeur(u)
  pour tous les successeurs v de u faire
    si marque[v] = faux alors
      marque[v] ← vrai
      _ParcoursProfondeur(v)
    fsi
  fin pour
Fin
```

Les méthodes `parcours_profondeur` et `_parcours_profondeur` de la classe `GrapheAlgo` implémentent cet algorithme.

## 2.4 Recherche des composantes fortement connexes

La recherche des composantes connexes est une application classique du parcours en profondeur. Une composante fortement connexe d'un graphe  $G = (S, A)$  est un ensemble maximal de sommets  $R \subseteq S$  tel que, pour

chaque paire de sommets  $u$  et  $v$  de  $R$ , les sommets  $u$  et  $v$  soient mutuellement accessibles.

Listing 9 – Recherche des composantes fortement connexes

```

CFC(G)
rang ← - 0
pour tous les sommets u de G faire
    rangs[u]      ← - 1
    theta[u]      ← - 1
    numero_cfc[u] ← - 1
    visite [u]     ← - faux
    empile[u]      ← - faux
fin pour

pour tous les sommets u de G faire
    si visite [u] = faux
        CFCDESC(G,u,rang)
    fsi
fpour
Fin

CFCDESC(G,s,rang)
Empiler(P, s)
empile[s] ← - vrai
visite [s] ← - vrai
rang      ← - rang + 1
rangs[s]  ← - rang
theta[s]  ← - rang

pour tous les successeurs u de s faire
    si visite [u] = faux
        CFCDESC(G,u,rang)
        si theta[s] >= theta[k] alors
            theta[s] ← - theta[k]
        fsi
    sinon si empile[u] = vrai
        si theta[s] >= rangs[k]
            theta[s] ← - rangs[k]
        fsi
    fsi
fpour

si theta[s] = rangs[s] alors
    faire
        u = depiler(P)
        empile[u]      ← - faux
        numero_cfc[u] ← - nb_cfc
    tant que u != s

```

```

        nb_cfc <- nb_cfc + 1
    fsi
Fin

```

## 2.5 Code source

Listing 10 – Implémentation des différents algorithmes

```

import java.util.*;

class GrapheAlgo
{
    private static void _parcours_profondeur(Graphe g, int s,
                                             boolean[] marque)
    {
        for (int i = g.premier_successeur(s);
             i != -1; i = g.successeur_suivant(s, i))
        {
            if (marque[i] == false)
            {
                marque[i] = true;
                _parcours_profondeur(g, i, marque);
            }
        }
    }

    static void parcours_profondeur(Graphe g, int s)
    {
        boolean[] marque;

        marque = new boolean [g.getNbSommet()];

        for(int i = 0; i < g.getNbSommet(); i++)
            marque[i] = false;

        marque[s] = true;
        _parcours_profondeur(g, s, marque);
    }

    static void parcours_largeur(Graphe g, int s)
    {
        LinkedList file = new LinkedList();
        boolean[] marque = new boolean [g.getNbSommet()];

        for(int i = 0; i < g.getNbSommet(); i++)
            marque[i] = false;
    }
}

```

```

    file .addFirst(new Integer(s));

marque[s] = true;

while(!( file .isEmpty()))
{
    s = ((Integer) file .removeLast()).intValue();
    for (int i = g.premier_successeur(s);
        i!= -1;
        i = g.successeur_suivant(s,i))
    {
        if (marque[i] == false)
        {
            marque[i] = true;
            file .addFirst(new Integer(i));
        }
    }
}

static void recherche_plus_court_chemin(Graphe g, int s)
{
    LinkedList file      = new LinkedList();
    boolean [] marque = new boolean [g.getNbSommet()];
    int [] distances    = new int [g.getNbSommet()];
    int [] parents      = new int [g.getNbSommet()];

    for(int i = 0; i < g.getNbSommet(); i++)
    {
        marque[i]      = false;
        parents[i]      = -1;
        distances[i]    = -1;
    }

    file .addFirst(new Integer(s));

    marque[s]      = true;
    distances[s]    = 0;
    parents[s]      = -1;

    while(!( file .isEmpty()))
    {
        s = ((Integer) file .removeLast()).intValue();
        for (int i = g.premier_successeur(s);
            i!= -1;
            i = g.successeur_suivant(s,i))
        {
            if (marque[i] == false)
            {

```

```

        marque[i]    = true;
        distances[i] = distances[s] + 1;
        parents[i]   = s;
        file .addFirst(new Integer(i));
    }
}

}

private static int cfcDesc (Graphe g, int s, int nb_cfc,
    int [] numero_cfc, int [] rangs, int [] theta,
    int rang, LinkedList pile , int n,
    boolean[] visite , boolean[] empile)
{
    pile .addFirst(new Integer(s));
    empile[s] = true;
    visite [s] = true;

    rang++;
    rangs[s] = rang;
    theta[s] = rang;

    for(int k = g.premier_successeur(s); k != -1; k = g.successeur_suivant(s,k))
    {
        if (! visite [k])
        {
            nb_cfc = cfcDesc(g,k,nb_cfc,numero_cfc,rangs,theta,rang,pile ,n
                ,visite ,empile);
            theta[s] = (theta[s] < theta[k]) ? theta[s] : theta[k];
        }
        else
        if (empile[k])
            theta[s] = (theta[s] < rangs[k]) ? theta[s] : rangs[k];
    }
    if(theta[s] == rangs[s])
    {
        int top;
        do
        {
            top = ((Integer) pile .removeFirst()).intValue();
            empile[top] = false;
            numero_cfc[top] = nb_cfc;
        } while (top != s);
        nb_cfc++;
    }
    return nb_cfc;
}

```

```

static int[] cfc (Graphe g)
{
    int nb_cfc = 0;
    int[] numero_cfc;
    int[] rangs;
    int[] theta;
    int i;
    int rang = 0;
    LinkedList pile = new LinkedList();
    int n; // nb de sommets
    n = g.getNbSommets();
    boolean[] visite;
    boolean[] empile;

    rangs = new int[n];
    theta = new int[n];
    numero_cfc = new int[n];
    visite = new boolean[n];
    empile = new boolean[n];

    for(i = 0; i < n; i++)
    {
        rangs[i] = -1;
        theta[i] = -1;
        numero_cfc[i] = -1;
        visite[i] = false;
        empile[i] = false;
    }

    for(i = 0; i < n; i++)
        if (!visite[i])
            nb_cfc = cfcDesc(g,i,nb_cfc,numero_cfc,rangs,theta,rang,pile,n,visite,empile);

    return numero_cfc;
}
}

```

### 3 Analyse temporelle des algorithmes

Chaque algorithme a été exécuté par un petit programme de test en Java, sur des graphes comportant un grand nombre de sommets. Nous avons choisi de faire des graphes de  $n$  sommets et  $2 * n$  arcs. Les arcs ont été positionnés de manière aléatoires sur le graphe.

Nous avons au départ utilisé la représentation par matrice d'adjacences, mais celle-ci ne convient pas, car la plupart des algorithmes sur les graphes

prenent alors un temps en  $O(n^2)$ , au lieu d'un temps linéaire. Les résultats présentés ont donc été réalisés en utilisant la représentation par listes d'adjacences.

Tous les algorithmes ont donc été implémentés en Java, et exécuté avec le JDK 1.4.1 sous un système Linux.

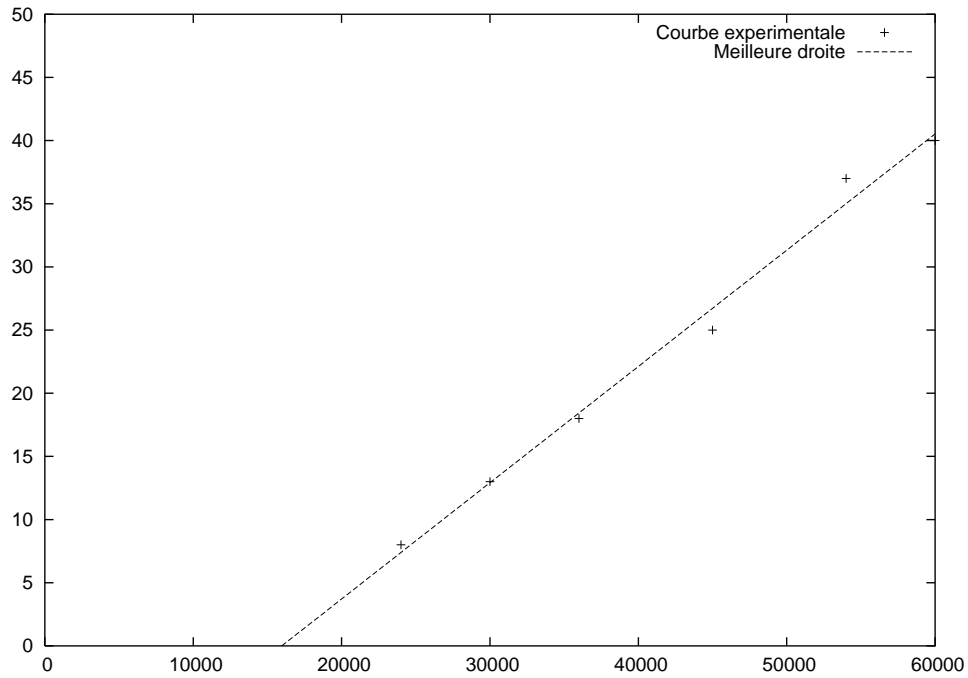
### 3.1 Parcours en largeur d'un graphe orienté

#### 3.1.1 Mesures expérimentales

Cet algorithme a été testé 15 fois pour chaque valeur du nombre de sommets, et pour chaque valeur du nombre de sommets, 100 parcours en largeur à partir de différents sommets ont été réalisés. Ceci permet d'obtenir pour chaque valeur du nombre de sommets une valeur moyenne.

$card(S)$	$card(S + A)$	Temps
8000	24000	8
10000	30000	13
12000	36000	18
15000	45000	25
18000	54000	37
20000	60000	40

Ces résultats donnent la courbe linéaire suivante :



### 3.1.2 Analyse théorique

Après l'initialisation (première boucle), tous les sommets sont marqués comme étant non parcourus, et le test `st marque[v] = faux` nous garantit que chaque sommet sera enfilé au plus une fois, et donc défilé au plus une fois. Les opérations d'enfilement et de défilement étant en  $O(1)$ , le temps total des opérations de file est  $O(S)$ . Comme la liste d'adjacences de chaque sommet n'est balayée qu'au moment où le sommet est défilé, la liste d'adjacence de chaque sommet est donc parcourue au plus une fois. La somme des longueurs de toutes les listes d'adjacences étant  $\Theta(A)$ , le temps total consacré au balayage des listes d'adjacence est  $O(A)$ . Le coût de l'initialisation est  $O(S)$  et le temps d'exécution total de l'algorithme est donc  $O(S + A)$ . Donc le parcours en largeur s'exécute en un temps qui est linéaire par rapport à la taille de la représentation par listes d'adjacences de  $G$ .

## 3.2 Recherche du plus court chemin

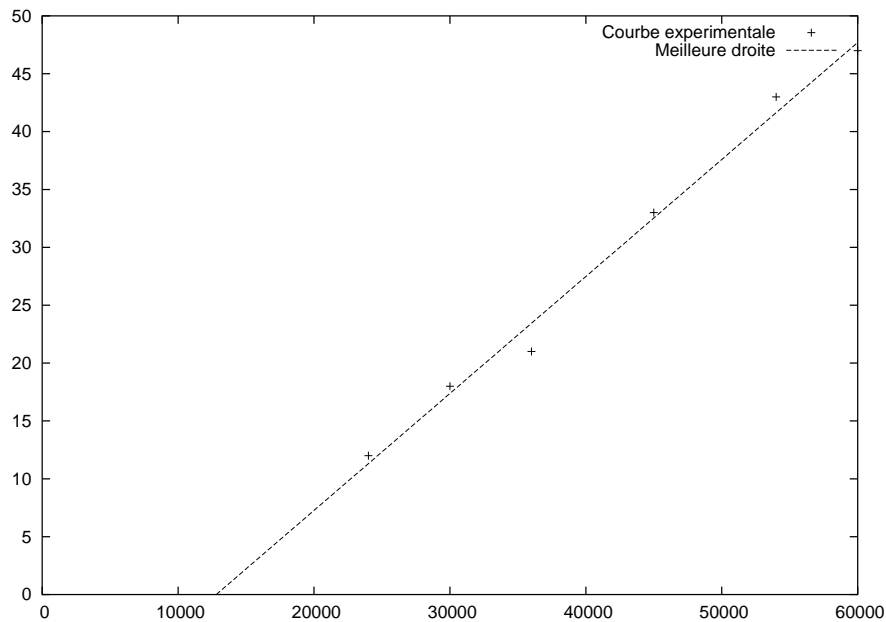
### 3.2.1 Mesures expérimentales

De la même façon que pour le parcours en largeur, cet algorithme a été testé à partir de 100 sommets différents sur 15 graphes différents pour chaque taille de graphe testée.



$card(S)$	$card(S + A)$	Temps
8000	24000	12
10000	30000	18
12000	36000	21
15000	45000	33
18000	54000	43
20000	60000	47

Ces résultats donnent la courbe linéaire suivante :



### 3.2.2 Analyse théorique

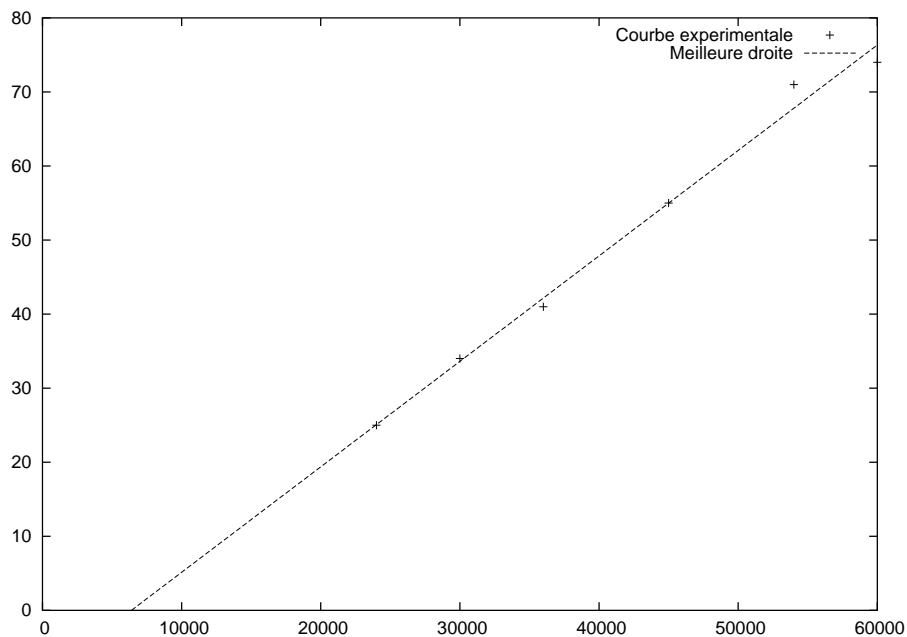
L'analyse de cet algorithme du plus court chemin est très simple, puisqu'il est strictement identique au parcours au largeur, sauf que l'on remplit un tableau **distances** et un tableau **parents** durant le parcours. L'initialisation prend toujours un temps  $O(S)$ , et le corps de l'algorithme un temps  $O(A)$ , donc l'algorithme total s'exécute en un temps  $O(S + A)$ , temps linéaire avec la taille de la représentation du graphe par listes d'adjacences.

### 3.3 Parcours en profondeur

Les tests effectués pour le parcours en profondeur sont identiques à ceux réalisés pour le parcours en largeur : pour chaque taille de graphe, on teste sur 15 graphes différents un parcours en profondeur à partir de 100 sommets différents, afin de donner une moyenne.

$card(S)$	$card(S + A)$	Temps
8000	24000	25
10000	30000	34
12000	36000	41
15000	45000	55
18000	54000	71
20000	60000	74

Ces résultats donnent la courbe linéaire suivante :



#### 3.3.1 Analyse théorique

L'exécution de la boucle d'initialisation dans la fonction `ParcoursProfondeur`, prend un temps  $O(S)$ . La fonction `_ParcoursProfondeur` est appelée une fois pour chaque sommet accessible depuis le sommet d'origine du parcours. La

boucle de cette fonction est exécutée autant de fois qu'il y a de successeurs au sommet courant. Si on somme ce nombre de successeurs, on obtient la valeur  $\Theta(A)$ , donc le coût total de la fonction `ParcoursProfondeur` est  $\Theta(A)$ . Le temps d'exécution de `ParcoursProfondeur` est donc  $\Theta(S + A)$ , ce qui est linéaire avec la taille de la représentation par listes d'adjacences du graphe.

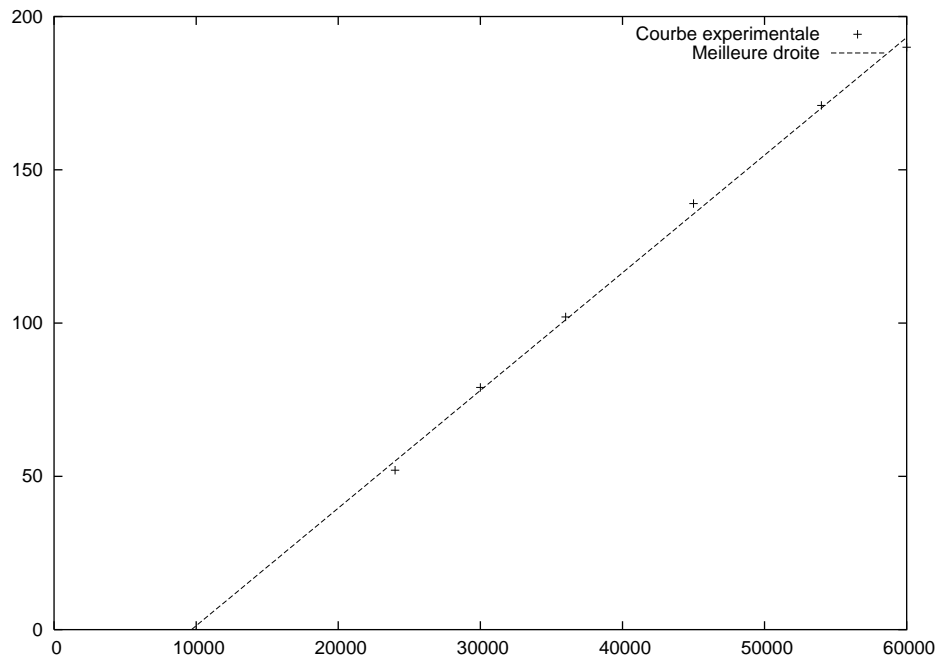
### 3.4 Recherche des composantes fortement connexes

#### 3.4.1 Mesures expérimentales

Ici, les tests ont simplement été réalisés sur 15 graphes différents pour chaque valeur du nombre de sommets.

$card(S)$	$card(S + A)$	Temps
8000	24000	52
10000	30000	79
12000	36000	102
15000	45000	139
18000	54000	171
20000	60000	190

Ces résultats donnent la courbe linéaire suivante :



### 3.4.2 Analyse théorique

La boucle d'initialisation de la fonction **CFC** prend un temps  $O(S)$ . La boucle qui suit est exécutée au plus pour tous les sommets du graphe, elle prend donc un temps  $O(S)$ . La fonction **CFCDESC** boucle sur la liste des successeurs du sommet courant. Chaque boucle fait donc autant d'itérations qu'il y a de successeurs. Il y a au total  $\Theta(A)$  arcs dans le graphe, l'exécution de cette fonction prendra donc au plus  $O(A)$ . L'algorithme de recherche des composantes connexes s'exécute donc en  $O(S + A)$ , soit un temps linéaire par rapport à la taille du graphe.

## Listings

1	Interface Graphe . . . . .	3
2	Graphe orienté par matrice d'adjacence . . . . .	6
3	Graphe orienté par listes chaînées . . . . .	10
4	Graphe non orienté par matrice d'adjacence . . . . .	13
5	Graphe non orienté par listes chaînées . . . . .	14
6	Parcours en largeur . . . . .	15
7	Recherche du plus court chemin . . . . .	16
8	Parcours en profondeur . . . . .	17
9	Recherche des composantes fortement connexes . . . . .	18
10	Implémentation des différents algorithmes . . . . .	19