

# Rapport PFE

Bel Haj Ali Marwa

16 mai 2016

Ministère de l'Enseignement Supérieur et de la Recherche  
Scientifique

Université de Monastir

\*\*\*\*\*

Institut Supérieur d'Informatique et de Mathématiques  
de Monastir



## *Projet de Fin d'Etudes*

En vue de l'obtention du

Diplôme de Licence Fondamentale en Sciences Informatiques

*Tri topologique : Réalisation en C et Java*

Réalisé par

**Marwa Bel Haj Ali**

sous la direction de

**M. Mohamed Graiet**

Année universitaire 2015-2016

# Dédicace

## **Je dédie ce rapport à A ma très chère mère Amel Kacem**

Affable, honorable, aimable : Tu représentes pour moi le symbole de la bonté par excellence, la source de tendresse et l'exemple du dévouement qui n'a pas cessé de m'encourager et de prier pour moi.

Ta prière et ta bénédiction m'ont été d'un grand secours pour mener à bien mes études.

Aucune dédicace ne saurait être assez éloquente pour exprimer ce que tu mérites pour tous les sacrifices que tu n'as cessé de me donner depuis ma naissance, durant mon enfance et même à l'âge adulte.

Tu as fait plus qu'une mère puisse faire pour que ses enfants suivent le bon chemin dans leur vie et leurs études. Je te dédie ce travail en témoignage de mon profond amour. Puisse Dieu, le tout puissant, te préserver et t'accorder santé, longue vie et bonheur.

## **A mon chère père Mohamed BelHajAli**

Aucune dédicace ne saurait exprimer l'amour, l'estime, le dévouement et le respect que j'ai toujours eus pour vous.

Rien au monde ne vaut les efforts fournis jour et nuit pour mon éducation et mon bien être.

Ce travail est le fruit des sacrifices que tu as consentis pour mon éducation et ma formation.

## **A ma très chère soeur Khawla, son mari Housseem et sa future fille Gharam**

En témoignage de l'attachement, de l'amour et de l'affection que je porte pour vous.

Je vous remercie pour votre hospitalité sans égale et votre affection si sincère.

Je vous dédie ce travail avec tous mes vœux de bonheur, de santé et de réussite.

### **A mon très cher frère Yacine**

Mon cher frère qui m'est le père et la mère, les mots ne suffisent guère pour exprimer l'attachement, l'amour et l'affection que je porte pour vous.

Mon ange gardien et mon fidèle compagnon dans les moments les plus délicats de cette vie mystérieuse.

Je vous dédie ce travail avec tous mes vœux de bonheur, de santé et de réussite.

### **A ma très chère Soeur Meriem**

Ma chère petite soeur présente dans tous mes moments d'examens par son soutien moral et ses belles surprises sucrées.

Je te souhaite un avenir plein de joie, de bonheur, de réussite et de sérénité.

Je t'exprime à travers ce travail mes sentiments de fraternité et d'amour.

### **À oncles et tantes : Chadlia, Habiba, Fatma et Ali**

En témoignage de mon amour, de mon profond respect et de ma reconnaissance.

### **À mes tantes Mounira, Wided, Samia et Sonia**

### **À mes oncles Mounir et Adel**

Vous avez toujours été présents pour les bons conseils.

Votre affection et votre soutien m'ont été d'un grand secours au long de ma vie professionnelle et personnelle.

Veillez trouver dans ce modeste travail ma reconnaissance pour tous vos efforts.

Que Dieu vous protège.

**A mes cousins et cousines :Amina, Ibtissem, Mohamed, Amine, Amir, Oussama, Wassim, Youssef, Safoine, Mohamed, Rihem, Arij, Anass, Elaa, Adem, Amna, Hiba, Nada et Mohamed**

Aucune dédicace ne saurait exprimer tout l'amour que j'ai pour vous, votre joie et votre gaieté me comblent de bonheur.

Puisse Dieu vous garder, éclairer votre route et vous aider à réaliser à votre tour vos vœux les plus chers.

Veillez trouver dans ce travail l'expression de mon respect le plus profond et mon affection la plus sincère.

**À la mémoire de mes grands-parents : Mohamed, Habiba,  
Aroussi et Fattouma**

J'aurais tant aimé que vous soyez présents.  
Que Dieu ait vos âmes dans sa sainte miséricorde.

**À la mémoire de mon oncle : Abdallah**

Que Dieu vous accorde sa miséricorde.

**À tous mes amis**

**À tous mes professeurs d'école primaire, de  
l'enseignement secondaire et de l'institut supérieur  
d'informatique et de mathématique.**

**À tous ceux qui ont participé à ma formation**

Que Dieu vous bénisse  $\infty$

# Remerciements

La plus grande difficulté avec les remerciements, c'est d'essayer de penser à tout le monde. Mais durant ces trois années, il y a tellement de personnes qui m'ont aidée à traverser cette épreuve, que je ne peux pas les citer tous. Et je m'excuse donc par avance pour les noms qui ne seront pas mentionnés.

Pour commencer, je tiens à remercier mon directeur de projet de fin d'étude Monsieur *Mohamed Graiet* ainsi que Monsieur *Lazher El Hamel*, sans qui ce projet n'aurait pas pu exister, progresser et aboutir. Je les remercie de n'avoir jamais douté de mes capacités. Je tiens à vous remercier pour votre encadrement, pour votre soutien permanent, votre écoute, vos conseils, votre aide, votre patience, votre respect, votre bonne humeur et vos encouragements. La liste est encore longue... Ce dernier n'aurait pas vu le jour sans la confiance, la patience, et la générosité qu'ils ont su m'accorder. Ce fut un réel plaisir de partager toutes ces heures à vos côtés sur des problèmes passionnants. J'espère que nous continuerons à travailler ensemble.

Un immense merci également à Mademoiselle *Rim Fakhfakh* pour avoir accepté de rapporter ce manuscrit, ainsi qu'à Monsieur *Abdelwahed Berguiga* pour avoir accepté de présider le jury de ma soutenance.

Mes remerciements s'adressent aussi à tous les professeurs ainsi qu'au personnel de l'Institut supérieur d'informatique et de mathématiques pour leur soutien tout au long de mes études.

Et puis une personne que je mettrai à part, qui m'a permis d'être là où je suis aujourd'hui. Dans la vie il y a des gens qui nous marquent plus que d'autres et qui seront toujours là pour nous. Merci à Khouloud Ben Ali d'avoir été la meilleure camarade dont je ne pouvais même pas rêver et une amie hors norme sur qui on peut compter dans toutes les épreuves de la vie. Mes deux premières années de l'isimm n'auraient pas été si drôles, et l'envie d'aller à la faculté n'était pas aussi grande si elle n'avait pas été là. Merci à elle pour toutes nos discussions scientifiques et surtout celles non scientifiques, pour tous ce qu'on a partagé ensemble et bien plus encore.

Elle a toujours été là quand il fallait, malgré la distance. Enfin, il nous est agréable d'exprimer mon profonde et respectueuse gratitude et nos sincères remerciements à toutes les personnes qui ont soutenu ce travail et aidé, de près ou de loin, à le mener à bien.

Voilà, c'était long mais il y a eu tellement de personne qui ont compté et qui m'ont aidé que je leur devais bien ça.

# Resumé

Dans le cadre de notre projet de fin d'étude, je me suis intéressée à la problématique du tri topologique et la représentation d'un graphe. J'ai ainsi réalisé deux programmes permettant de représenter le graphe et le trier.

La théorie de graphes est utilisée dans des domaines variés : vie quotidienne, maths, réseaux ...

Dans ce rapport je présenterai en premier temps le premier programme implémenté en C un langage procédural et compilé. Je décrirai les structures des données utilisé pour présenter le graphe et le fonctionnement de la méthode du tri topologique.

Ensuite, je présenterai le programme "MGraph" pour ordonner les applications structurées en Graphe orienté acyclique de dépendance DAG implémenté en JAVA un langage orienté objet. Je décrirai comment ce programme a été implémenté avec une interface pour dessiner des graphes générique, tester les dags et les trier topologiquement. Je donnerais enfin quelques résultats sur ces deux programmes.

**Mots clés : dag, java, c, Tri topologique**



# Abstract

As part of our project of end of study, I became interested in the problem of topological sorting and representation of a graph.

I carried out two programs to represent the graph and sort it. The graph theory is used in various fields : daily life, math, networks ... In this report I will present in the first program first implemented in C a compiled and procedural language. I will describe the data structures used to present the graph and the operation of the topological sorting method.

Then I will present the program "MGraph" to order the structured applications in directed acyclic graph of dependence DAG implemented in Java object oriented language. I will describe how the program was implemented with an interface to draw generic graphs, test and sort dags topologically. I finally give some results on these two programs.

**Keywords : dag, java, c, Topological sort**

# Introduction générale

## Domaine abordé

La théorie des graphes a connu un essor spectaculaire ces dernières années, dû en partie aux importantes interactions entre les aspects théorique et algorithmique du domaine, et les nouveaux enjeux scientifiques. Les graphes jouent un rôle prépondérant dans la recherche en sciences et technologies de l'information, ainsi qu'en bio-informatique. Ils apparaissent également comme des composants essentiels de beaucoup de théories mathématiques plus anciennes. La théorie des graphes est une branche des mathématiques qui, historiquement, s'est développée au sein de disciplines diverses telles que la chimie (modélisation de structures), la biologie (génomique), les sciences sociales (modélisation des relations) ou en vue d'applications industrielles.

Un graphe permet de représenter simplement la structure et les connexions d'un ensemble d'entités, en exprimant les relations ou dépendances entre ses éléments (réseaux ferroviaire, arbre généalogique, coloration d'une carte). À partir de cette forte base mathématique, les graphes sont devenus des structures de données puissantes pour l'informatique.

Les applications des graphes sont nombreuses : les plus classiques concernent la recherche des chemins - par métaphore nous pouvons dire que tout problème est une recherche de chemins dans un labyrinthe AD HOC, ou la notion de chemin correspond à un plan d'actions à entreprendre. Un autre groupe important d'applications des graphes concerne les réseaux de communication. On trouve également les graphes dans l'étude de problèmes liés aux relations entre différentes entités (graphes de dépendance ou graphes d'influence), ainsi qu'à des évolutions possibles d'un système (de transitions). En intelligence artificielle, on emploie les graphes entre autres dans les problèmes de résolution, qui, par métaphore se ramènent à rechercher des chemins dans un espace d'états donné. L'intérêt des graphes apparaît lorsqu'on s'intéresse aux problèmes de nature to-

pologique, tels que les problèmes de chemins, de connexité ou de fiabilité structurelle, ainsi que dans la théorie de relations et en analyse de données. Les graphes trouvent également des applications en sciences humaines, par exemple pour modéliser les relations mutuelles entre les membres d'un groupe (sociogrammes).

Une autre catégorie d'application de graphes concerne la théorie des automates, en relation avec la théorie des langages formels. Finalement, notons que les graphes sont les plus complètes parmi les structures de données, d'où leur intérêt dans la programmation. En résumé nous pouvons dire que les graphes sont adaptés à la modélisation.

## **Problématique traitée**

Les problèmes d'ordonnancement de tâches sont toujours un sujet de recherche active aujourd'hui. On rencontre parfois des problèmes pour lesquels on recherche un ordre acceptable dans l'ordonnement de tâches dépendant les unes des autres. On peut par exemple penser à la fabrication d'une voiture sur une chaîne de montage : on peut monter de façon indépendante le moteur et la carrosserie, avant de les assembler. Un autre exemple provient des choix de cours réalisés par des étudiants. En effet, certains prérequis sont parfois nécessaires. On considérera un graphe dont les sommets sont les tâches (resp. les cours) à réaliser (resp. à suivre) et si une tâche (resp. un cours) doit être réalisée (resp. suivi) avant une autre (resp. un autre), On peut représenter chacune des tâches à effectuer par un noeud, et les dépendances entre chacune des tâches par les arêtes. On résout ce problème avec un tri topologique. Le problème s'avère être polynomial, nous proposerons un algorithme à la fois simple et efficace pour le résoudre (notamment des algorithmes en temps et espace linéaire). L'objet du présent travail est de s'intéresser à quelques problèmes de graphes pour lesquels on ne connaît pas d'algorithme polynomial pour les résoudre.

## **Organisation du rapport**

### **Présentation générale**

En plus d'une introduction et conclusion générales, ce manuscrit est organisé en deux parties :

- Partie I : État de l'art La première partie présente le contexte dans

lequel s'inscrivent nos travaux. Elle est constituée de deux parties. Le première partie a pour but d'introduire quelques concepts de la théorie des graphes et quelques notions de complexité algorithmique. La seconde partie a pour but d'expliquer le problème de tri topologique et introduire son algorithme et son intérêt à résoudre des problèmes divers.

- **Partie II : Réalisation** Dans la seconde partie du manuscrit, sont présentées les contributions de ce projet. Quant à elle, cette partie est composée de trois chapitres. Dans le deuxième chapitre, on introduit une implémentation du problème par un langage de programmation C. Puis, dans le trixième chapitre, on introduit une implémentation du problème par un langage de programmation orienté objet. Et enfin dans le quatrième chapitre, on finira par une conclusion finale .

### **Plan : Vue globale**

Introduction générale

- Chapitre 1 : Graphe
- Chapitre 2 : Algorithme en c
- Chapitre 3 : Algorithme en java
- Chapitre 4 : Conclusion

# Table des matières

<b>Introduction générale</b>	<b>8</b>
<b>I Etat d'art</b>	<b>16</b>
<b>1 Graphe</b>	<b>18</b>
1.1 Introduction . . . . .	18
1.2 Définitions et notations . . . . .	20
1.2.1 Graphe et sous graphe . . . . .	20
1.2.2 Graphe avec boucle . . . . .	20
1.2.3 Graphe orienté . . . . .	21
1.2.4 Graphe valué . . . . .	21
1.2.5 Chaîne – Cycle . . . . .	22
1.2.6 Connexité . . . . .	22
1.3 Caractéristiques des graphes . . . . .	23
1.3.1 Adjacence . . . . .	24
1.3.2 Voisinage . . . . .	24
1.3.3 Degré . . . . .	25
1.4 Représentations et stockage en mémoire . . . . .	26
1.4.1 Généralité . . . . .	26
1.4.2 Matrice d'incidence sommets-arcs . . . . .	26
1.4.3 Listes d'adjacence . . . . .	27
1.4.4 Autres représentations . . . . .	28
1.5 Graphe orienté acyclique et Tri topologique . . . . .	29
1.5.1 définition informelle . . . . .	29
1.5.2 Algorithme du Graphe sans cycle . . . . .	30
1.5.3 Algorithme de tri topologique . . . . .	33
1.5.4 Applications du tri topologique . . . . .	34
1.6 conclusion . . . . .	35

<b>II</b>	<b>Réalisation</b>	<b>36</b>
<b>2</b>	<b>Réalisation en c</b>	<b>37</b>
2.1	Introduction . . . . .	37
2.1.1	Principe du tri topologique . . . . .	37
2.2	Algorithme proposé . . . . .	39
2.2.1	Choix d'une structure de données . . . . .	40
2.3	Traduction en C . . . . .	42
2.4	Conclusion . . . . .	47
<b>3</b>	<b>Réalisation en java</b>	<b>48</b>
3.1	Introduction . . . . .	48
3.1.1	Ception de l'application . . . . .	48
3.2	Réalisation de l'application . . . . .	49
3.2.1	Implémentation des classes . . . . .	49
3.2.2	Représentation de l'interface . . . . .	54
3.2.3	Teste de l'application . . . . .	62
3.2.4	Conclusion . . . . .	65
	<b>Conclusion</b>	<b>69</b>
	<b>Bibliographie</b>	<b>71</b>

# Table des figures

1.1	plan et schéma des sept ponts de Königsberg . . . . .	18
1.2	représentation en graphe des sept ponts de Königsberg	19
1.3	sous-graphe $G'$ d'un graphe $G$ . . . . .	20
1.4	graphe $G$ formé de 2 sommets et deux arcs dont une boucle . . . . .	21
1.5	graphe orienté . . . . .	21
1.6	graphe valué . . . . .	22
1.7	Chaîne – Cycle . . . . .	22
1.8	graphe non connexe . . . . .	23
1.9	Graphe réflexif, antisymétrique, transitif et complet .	24
1.10	Graphe orienté . . . . .	25
1.11	Matrice adjascence . . . . .	26
1.12	Matrice d'incidence . . . . .	27
1.13	Listes d'adjacence . . . . .	28
2.1	Graphe représentant l'ensemble $S$ . . . . .	38
2.2	Résultat 1 après tri topologique sur $S$ . . . . .	38
2.3	Résultat 2 après tri topologique sur $S$ . . . . .	38
2.4	Respect de la transitivité . . . . .	39
2.5	Non respect de la transitivité . . . . .	39
2.6	Respect $1 < 2$ . . . . .	40
2.7	Non respect $1 < 2$ $2 < 1$ . . . . .	40
2.8	DAG représentant $S$ . . . . .	41
2.9	Représentation physique du graphe en $C$ . . . . .	41
3.1	Diagramme du classe . . . . .	48
3.2	Interface Mgraph . . . . .	55
3.3	Choix du couleur . . . . .	57
3.4	Help . . . . .	62
3.5	Sauvgarde du résultat . . . . .	63
3.6	MSommets dessinés . . . . .	63
3.7	MGraph du Make dessiné . . . . .	64
3.8	Quick view Make . . . . .	65
3.9	Résultat du tri (1) . . . . .	66

3.10	Résultat du tri (2)	66
3.11	Graphe cyclique	67
3.12	Graphe non orienté	67
3.13	Sauvgarde du tri topologique	68
3.14	Image du MGraph Make	68



# Liste des tableaux

1.1	Liste d'adjacence vs Matrice d'adjacence . . . . .	28
3.1	Différents forme du MSommet . . . . .	56
3.2	Différentes forme de la fin de la flèche d'une connexion	58
3.3	Différentes forme d'une connexion . . . . .	59

**Première partie**

**Etat d'art**

*“Un petit dessin vaut mieux  
qu’un grand discours,,  
Napoléon.*

# Chapitre 1

## Graphe

Dans ce chapitre, nous introduisons les principales définitions et notations utilisées dans ce manuscrit.

On se référera à [3] pour plus de détails sur les notions introduites dans ce chapitre.

### 1.1 Introduction

La théorie des graphes débute avec les travaux d'Euler, 1736+. L'histoire veut que Léonard Euler en visite dans la ville de Königsberg en Prusse orientale (aujourd'hui elle se nomme Kaliningrad et appartient à la fédération de Russie) ait tenté de répondre à un problème : trouver un circuit qui emprunte une seule fois chacun des sept ponts disposés comme suit dans la figure : Figure 1.1 [3].

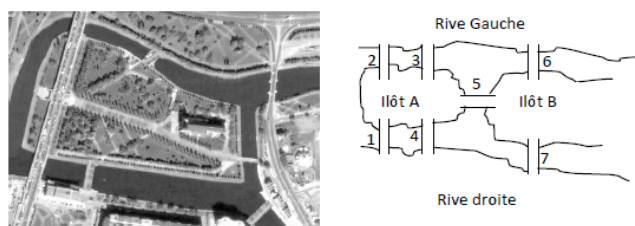


FIGURE 1.1 – plan et schéma des sept ponts de Königsberg

Euler choisit, pour étudier ce problème, de le représenter sous forme de graphe, c'est-à-dire de diagramme où il relie aux quatre lieux possibles les sept ponts de la façon suivante dans la figure : Figure 1.2 [3].

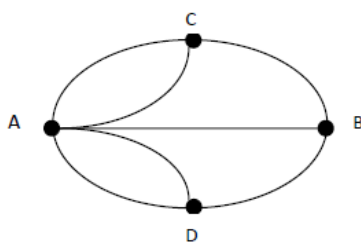


FIGURE 1.2 – représentation en graphe des sept ponts de Königsberg

Cette modélisation permet de traduire le problème initial en problème autour des propriétés du graphe : « peut-on circuler sur le graphe à partir d'un des points en empruntant une fois et une seule chaque lien ». Euler démontra que ce problème n'a pas de solution. De la même façon, de nombreuses méthodes, propriétés, procédures ont été imaginées ou trouvées à partir d'un dessin ou d'un schéma. C'est un des principes fondateurs de la théorie des graphes. En effet, un graphe permet de représenter la structure et les connexions d'un ensemble complexe en exprimant les relations entre ses éléments. Les graphes permettent de modéliser une grande variété de problèmes en se ramenant à l'étude de sommets et de liens. Après les travaux d'Euler, Kirchhoff développa, au milieu du 19ème siècle, la théorie des arbres pour l'appliquer à l'analyse de circuits électriques. La théorie des graphes a également été utilisée en chimie pour modéliser les molécules en considérant que les atomes sont des sommets et les liens entre atomes sont des arêtes. A partir de 1950, la théorie des graphes a connu un développement intense en devenant une branche à part entière des mathématiques grâce aux travaux de chercheurs tels que König, Menger, Cayley, Kuhn, Ford, Fulkerson, Roy et Erdős. Son essor en France est dû aux travaux de Claude Berge qui initia le passage à l'aire moderne de cette théorie en rassemblant les travaux épars dans la littérature dans son ouvrage « Théorie des graphes et applications ». Comme nous le verrons, la théorie des graphes présente des liens évidents avec l'algèbre linéaire, la topologie, la théorie des nombres et les statistiques. De nombreux domaines ont aujourd'hui recours aux graphes dans le but de traiter des problèmes rencontrés dans le monde réel. Parmi ces domaines, on trouve notamment ceux traitant des réseaux. Les réseaux peuvent être de diverses natures :

- Biologiques (chaînes de protéines, réseaux de gènes, topologie du cerveau,...),
- Technologiques (réseaux routiers, réseaux de télécommunication,

- Internet,...),
- Sociaux (réseaux d'affiliation, réseaux d'échange internationaux,...),
- De mots (réseaux de cooccurrence, réseaux sémantiques,...),
- ...

## 1.2 Définitions et notations

### 1.2.1 Graphe et sous graphe

Un graphe fini  $G = (V, E)$  est défini par l'ensemble fini  $V = \{v_1, v_2, \dots, v_n\}$  dont les éléments sont appelés sommets (Vertices en anglais) qu'on note aussi  $V(G)$ , et par l'ensemble fini  $E = \{e_1, e_2, \dots, e_m\}$  dont les éléments sont appelés arêtes (Edges en anglais) aussi noté  $E(G)$ . Un sommet peut-être également nommé noeud et une arête peut également être nommée arc ou lien. Si une arête  $e$  relie deux sommets  $x$  et  $y$ , on dit que l'arête est incidente aux sommets  $x$  et  $y$ . Les sommets  $x$  et  $y$  sont alors adjacents, ou incidents à  $e$ . Un graphe est dit complet si toute paire de sommets de  $G$  est une arête [6]. Le nombre de sommets de  $G$  est appelé l'ordre de  $G$  est  $|V(G)|$  et est également noté  $|G|$ .

Le sous-graphe d'un graphe  $G(V, E)$  est obtenu en enlevant un ou plusieurs sommets de  $G$  ainsi que toutes les arêtes incidentes aux noeuds supprimés comme l'exemple suivant dans la figure : Figure 1.3 [3] :

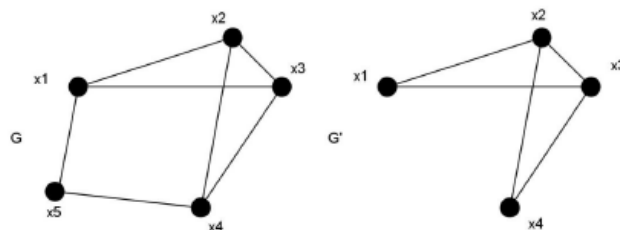


FIGURE 1.3 – sous-graphe  $G'$  d'un graphe  $G$   
 $G'$  est le sous graphe engendré par  $\{x_1, x_2, x_3, x_4\}$ .

### 1.2.2 Graphe avec boucle

il est possible d'enrichir la définition précédente pour prendre en compte l'idée de boucle sur un sommet [3], c'est-à-dire le fait que l'arc relie le sommet à lui-même comme l'exemple suivant dans la figure : Figure 1.4 [3] :



FIGURE 1.4 – graphe  $G$  formé de 2 sommets et deux arcs dont une boucle  
 $G(V=\{x_1, x_2\}; E=\{(x_1, x_2); (x_1, x_1)\})$   
 L'arc de  $G(x_1, x_1)$  est appelé une boucle.

### 1.2.3 Graphe orienté

Un graphe peut être enrichi par des informations topologiques additionnelles. En particulier, si l'on fait une distinction entre  $(x,y)$  et  $(y,x)$  pour  $x$  et  $y$  dans  $V$ . Si le sommet  $x$  a un lien vers le sommet  $y$  et que cela n'implique pas que  $y$  en ait un vers  $x$ , alors le graphe est dit orienté. Les graphes orientés sont parfois appelés des digraphes. L'orientation d'une arête se traduit par une flèche indiquant un sens de parcours comme sur l'exemple suivant dans la figure :

Figure 1.5 [3] :

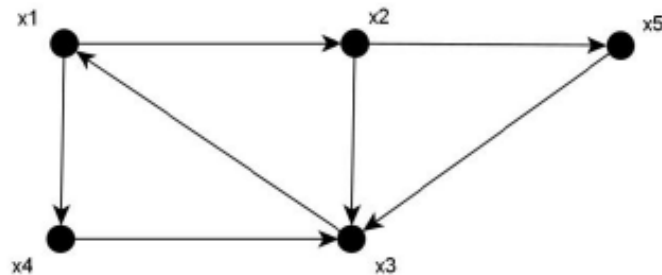


FIGURE 1.5 – graphe orienté  
 $G(V=\{x_1, x_2, x_3, x_4, x_5\};$   
 $E=\{(x_1, x_2); (x_1, x_3); (x_2, x_5); (x_5, x_4); (x_4, x_1); (x_2, x_4); (x_3, x_4)\})$

### 1.2.4 Graphe valué

Si on associe une valeur  $w(x,y)$  (par exemple un poids) à chaque arête  $(x,y)$  de  $E$ , alors le graphe est dit valué et est noté  $G(V,E,w)$ [3]. comme l'exemple suivant dans la figure : Figure 1.6 [3] :

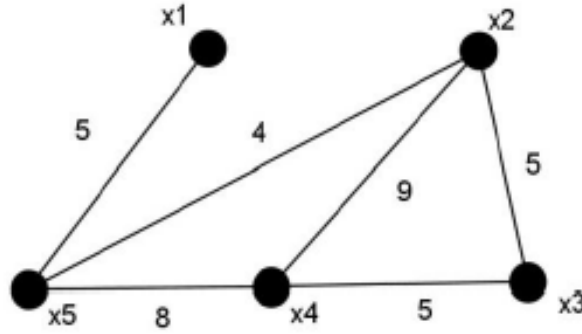


FIGURE 1.6 – graphe valué

$G(V=\{x_1, x_2, x_3, x_4, x_5\}; E=\{(x_1, x_5); (x_2, x_3); (x_2, x_4); (x_2, x_5); (x_3, x_4); (x_4, x_5)\})$ ,  
 $w=\{w(x_1, x_5) = 4, w(x_2, x_3) = 5, w(x_2, x_4) = 9, w(x_2, x_5) = 4, w(x_3, x_4) = 5, w(x_4, x_5) = 8\}$

### 1.2.5 Chaîne – Cycle

Une chaîne est une séquence d'arcs telle que chaque arc ait une extrémité commune avec le suivant. Un cycle est une chaîne qui contient au moins une arête, telle que toutes les arêtes de la séquence sont différentes et dont les extrémités coïncident [4]. Exemple dans la figure : Figure 1.7 [3] :

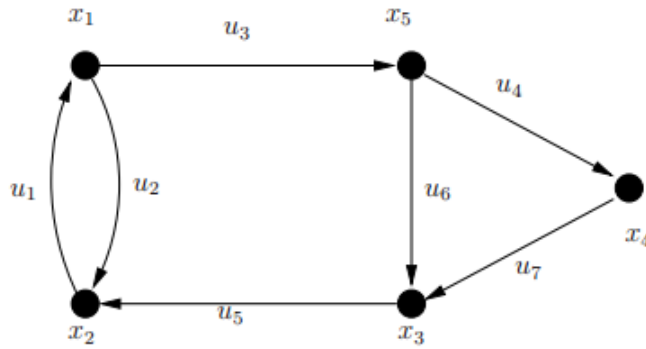


FIGURE 1.7 – Chaîne – Cycle

$\langle u_2, u_5, u_6, u_4 \rangle$  est une chaîne de  $x_1$  à  $x_4$   $\langle u_4, u_7, u_6 \rangle$  est un cycle

### 1.2.6 Connexité

Un graphe  $G(V, E)$  est connexe s'il existe pour chaque paire de sommet une chaîne (une suite d'arcs) reliant chacun des deux sommets. Un graphe non connexe se décompose en composantes connexes. Exemple dans la figure : Figure 1.8 [3] :



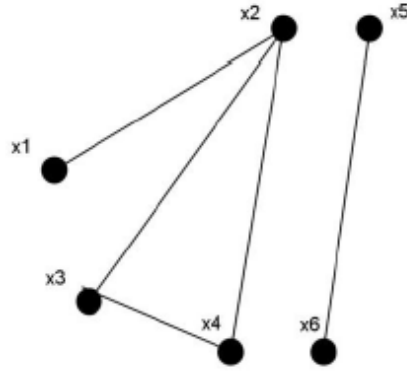


FIGURE 1.8 – graphe non connexe

$G(V=\{x_1, x_2, x_3, x_4, x_5, x_6\}, E=\{(x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_2), (x_5, x_6)\})$  est non connexe,  $\{x_1, x_2, x_3, x_4\}$  et  $\{x_5, x_6\}$  sont les deux composantes connexes de  $G$ .

### 1.3 Caractéristiques des graphes

Les caractéristiques des graphes sont définies par Pierre Lopez comme suit [4] :

Les deux données de départ qui permettent d'analyser un graphe sont :

Le nombre de noeuds  $|V(G)|$  que l'on notera  $n$ .

Le nombre de liens  $|E(G)|$  que l'on notera  $m$ .

Graphe réflexif :  $\forall x_i \in X, (x_i, x_i) \in U$ .

Graphe irréflexif :  $\forall x_i \in X, (x_i, x_i) \notin U$ .

Graphe symétrique :  $\forall x_i, x_j \in X, (x_i, x_j) \in U \Rightarrow (x_j, x_i) \in U$ .

Graphe asymétrique :  $\forall x_i, x_j \in X, (x_i, x_j) \in U \Rightarrow (x_j, x_i) \notin U$  ( si  $G$  est asymétrique,  $G$  est irréflexif ).

Graphe antisymétrique :  $\forall x_i, x_j \in X, (x_i, x_j) \in U$  et  $(x_j, x_i) \in U \Rightarrow x_i = x_j$  ( si  $G$  est asymétrique,  $G$  est aussi antisymétrique ).

Graphe transitif :  $\forall x_i, x_j \in X, (x_i, x_j) \in U, (x_j, x_k) \in U \Rightarrow (x_i, x_k) \in U$ .

Graphe complet :  $\forall x_i, x_j \in X, (x_i, x_j) \notin U \Rightarrow (x_j, x_i) \in U$ .

Exemple dans la figure Figure 1.9 [4] :

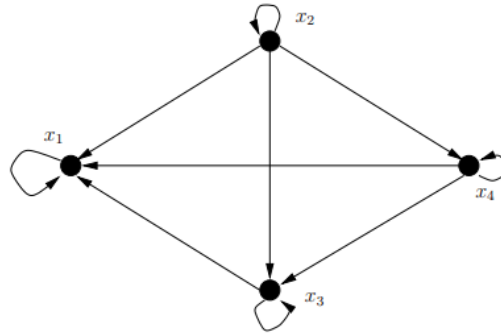


FIGURE 1.9 – Graphe réflexif, antisymétrique, transitif et complet

### 1.3.1 Adjacence

- Deux sommets sont adjacents (ou voisins) s'ils sont joints par un arc [7].
- Deux arcs sont adjacents s'ils ont au moins une extrémité commune [7].

### 1.3.2 Voisinage

Le voisinage d'un noeud  $x$ ,  $N(x)$  est l'ensemble des noeuds auxquels il est relié :

$$N(x) = \{y \in V, (x, y) \in E\} [1].$$

Dans le cas d'un graphe orienté, les arêtes entrantes et les arêtes sortantes d'un noeud sont différenciées.

Un noeud  $y$  est un successeur du noeud  $x$  s'il existe une arête ayant son extrémité initiale en  $x$  et son extrémité terminale en  $y$ .

L'ensemble des successeurs de  $x$  dans  $G$  se note :

$$R_+G(x) [3].$$

De même, on dit que  $y$  est un prédécesseur de  $x$  s'il existe une arête de la forme  $(y, x)$ .

L'ensemble des prédécesseurs de  $x$  dans  $G$  se note :

$$R_-G(x) [3].$$

L'union des successeurs et des prédécesseurs d'un noeud  $x$  forme le voisinage de  $x$  :

$$N(x) = R_-G(x) \cup R_+G(x) \text{ [7].}$$

### 1.3.3 Degré

Dans le cas d'un graphe orienté, les arêtes entrantes et les arêtes sortantes d'un noeud sont différenciées.

Le degré d'un noeud est alors la somme du nombre d'arêtes entrantes en  $x$  et du nombre d'arêtes sortantes de  $x$ .

Ces deux nombres sont respectivement appelés le demi-degré entrant de  $x$  et le demi-degré sortant de  $x$ .

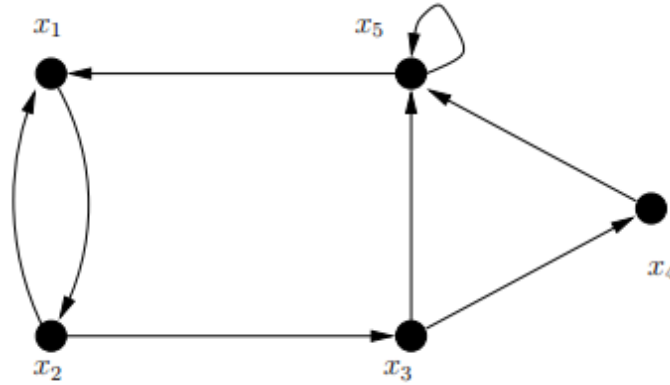
Le demi-degré extérieur de  $x_i$ ,  $d^+(x_i)$ , est le nombre d'arcs ayant  $x_i$  comme extrémité initiale :  $d^+(x_i) = |R_+(x_i)|$  [4].

Le demi-degré intérieur de  $x_i$ ,  $d^-(x_i)$ , est le nombre d'arcs ayant  $x_i$  comme extrémité finale :  $d^-(x_i) = |R_-(x_i)|$  [4].

Le degré de  $x_i$  est :  $d(x_i) = d^+(x_i) + d^-(x_i)$  [4].

Le degré d'un sommet d'un graphe non orienté est le nombre d'arêtes qui lui sont incidentes.

Exemple dans la figure Figure 1.10 [4] :



$$X = \{x_1, x_2, x_3, x_4, x_5\}$$

FIGURE 1.10 – Graphe orienté  
 $d^+(x_2) = 2; d^-(x_2) = 1; d(x_2) = 3.$   
 $d^+(x_5) = 2; d^-(x_5) = 3; d(x_5) = 5.$

## 1.4 Représentations et stockage en mémoire

### 1.4.1 Généralité

Un certain nombre de représentations existent pour décrire un graphe. En particulier, elles ne sont pas équivalentes du point de vue de l'efficacité des algorithmes. On distingue principalement la représentation par matrice d'adjacence, par matrice d'incidence sommets-arcs (ou sommets-arêtes dans le cas non orienté) et par listes d'adjacence.

La matrice d'adjacence [8] fait correspondre les sommets origine des arcs (placés en ligne dans la matrice) aux sommets destination (placés en colonne). Dans le formalisme matrice booléenne, l'existence d'un arc  $(x_i, x_j)$  se traduit par la présence d'un 1 à l'intersection de la ligne  $x_i$  et de la colonne  $x_j$  ; l'absence d'arc par la présence d'un 0 (dans un formalisme dit matrice aux arcs les éléments représentent le nom de l'arc). Exemple dans la figure Figure 1.11 [4] :

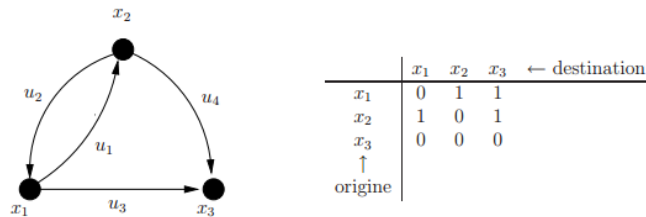


FIGURE 1.11 – Matrice adjacence

Place mémoire utilisée :  $n^2$  pour un graphe d'ordre  $|X| = n$  (i.e.,  $n$  sommets).

Remarque : pour des graphes numérisés (ou valués ou pondérés), remplacer « 1 » par la valeur numérique de l'arc.

### 1.4.2 Matrice d'incidence sommets-arcs

ligne  $\leftrightarrow$  *sommet*

colonne  $\leftrightarrow$  *arc*

Si  $u = (i, j) \in U$ , on trouve dans la colonne  $u$  :  $a_{iu} = 1$ ,  $a_{ju} = -1$  ; tous les autres termes sont nuls [4]. Exemple dans la Figure 1.12 [4] :

Place mémoire utilisée :  $n \times m$ .

Remarques : la somme de chaque colonne est égale à 0 (un arc a une origine et une destination).

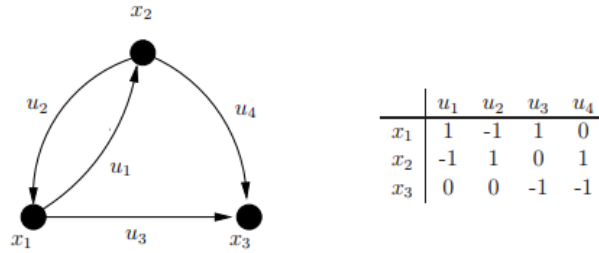


FIGURE 1.12 – Matrice d'incidence

la matrice est totalement unimodulaire, toutes les sous-matrices carrées extraites de la matrice ont pour déterminant  $+1$ ,  $-1$ .

### 1.4.3 Listes d'adjacence

L'avantage de la représentation par listes d'adjacence [4], par rapport à celle par matrice d'adjacence, est le gain obtenu en place mémoire; ce type de représentation est donc mieux adapté pour une implémentation. Le but est de représenter chaque arc par son extrémité finale, son extrémité initiale étant définie implicitement. Tous les arcs émanant d'un même sommet sont liés entre eux dans une liste. A chaque arc sont donc associés le nœud destination et le pointeur au prochain sommet dans la liste. On crée deux tableaux LP (tête de listes) de dimension  $(n + 1)$  et LS (liste de successeurs) de dimension  $m$  (cas orienté) ou  $2m$  (cas non orienté). Pour tout  $i$ , la liste des successeurs de  $i$  est dans le tableau LS à partir de la case numéro  $LP(i)$ .

1. On construit LS par  $R_+(1), R_+(2), \dots, R_+(n)$ .
2. On construit LP qui donne pour tout sommet l'indice dans LS où commencent ses successeurs.
3. Pour un sommet  $i \rightarrow$  1er suivant :  $LS(LP(i))$ ; 2ème suivant :  $LS(LP(i) + 1)$ . L'ensemble des informations relatives au sommet  $i$  est contenue entre les cases  $LP(i)$  et  $LP(i + 1) - 1$  du tableau LS.
4. Si un sommet  $i$  n'a pas de successeur, on pose  $LP(i) = LP(i + 1)$  (liste vide coincée entre les successeurs de  $i - 1$  et ceux de  $i + 1$ ). Pour éviter des tests pour le cas particulier  $i = n$  (le sommet  $i + 1$  n'existant pas), on « ferme » par convention la dernière liste en posant  $LP(n + 1) = m + 1$ .

Exemple dans la figure Figure 1.13 [4] :

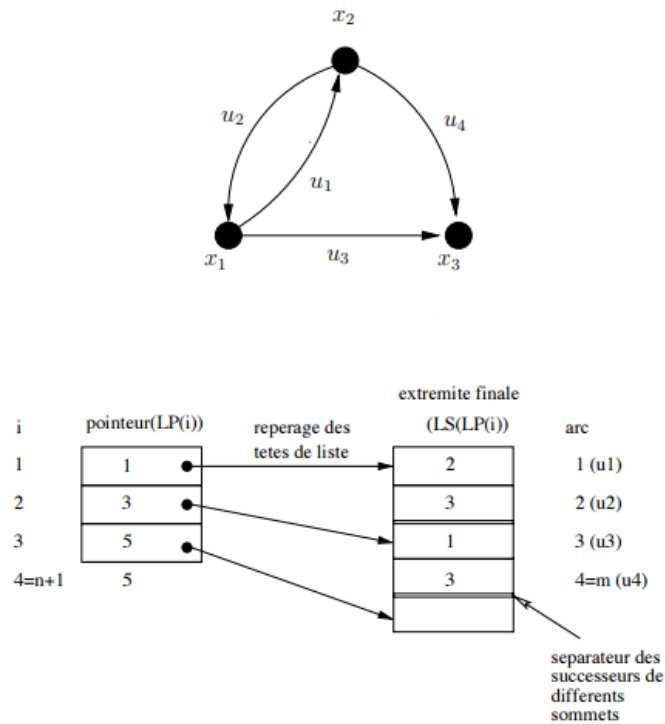


FIGURE 1.13 – Listes d’adjacence  
Place mémoire utilisée :  $(n + 1) + m$ .

## Conclusion

Un petit tableau récapitulatif fera l’affaire.

Opérations	Liste d’adjacence	Matrice d’adjacence
Retirer une arête	$O(d)$ avec $d$ le degré du nœud	$O(1)$
Ajouter une arête	$O(1)$	$O(1)$
Itérer sur les voisins d’un nœud	$O(d)$ avec $d$ le degré du nœud	$O(N)$
Tester si deux nœuds sont voisins	$(d)$ avec $d$ le degré du nœud	$O(1)$
Complexité mémoire	$O(N + A)$	$O(N^2)$

TABLE 1.1 – Liste d’adjacence vs Matrice d’adjacence

### 1.4.4 Autres représentations

Il existe bien d’autres représentations d’un graphe [8]. Citons particulièrement la matrice d’incidence. C’est une matrice de di-

mensions  $N \times A$  qui indique quels liens arrivent sur quels sommets. A l'intersection d'une ligne correspondant à noeud et d'une colonne correspondant à une arête on trouve un nombre. Il vaut 0 si cette arête n'est pas reliée à ce noeud. En revanche, si ce noeud est une extrémité de l'arête, ce coefficient diffère selon si le graphe considéré est orienté ou non. Pour un graphe non orienté, cette valeur vaudra alors 1 dans le cas général, et 2 si cette arête réalise une boucle sur ce noeud (car ce noeud est deux fois l'extrémité de cette arête). Ainsi la somme des coefficients sur une colonne vaudra toujours 2. Pour un graphe orienté ce coefficient vaut  $-1$  si l'arc sort du noeud considéré, et 1 lorsque l'arc entre dans le noeud. La raison pour laquelle je ne vous l'ai pas proposée en structure de données est simple : ses performances sont en tout point inférieures à celles de la liste ou de la matrice d'adjacence, quel que soit l'opération considérée. La consommation en mémoire, du  $O(N \times A)$ , soit un  $O(N^3)$  en pire des cas, est aberrante si le graphe est dense. Alors, pourquoi cette représentation a été inventée. Eh bien il se trouve que les graphes sont des objets mathématiques comme les autres, et leurs différentes représentations ont nécessairement des liens entre elles. Des relations relient la matrice d'incidence avec la matrice Laplacienne, la matrice d'adjacence, la matrice des degrés ou encore le line graph. Cela sert dans des calculs, et occasionnellement dans certains algorithmes.

## 1.5 Graphe orienté acyclique et Tri topologique

Dans cette section , nous introduisons le problème de tri topologique. On se référera à [7] pour plus de détails sur les notions introduites ci-dessous.

### 1.5.1 définition informelle

En théorie des graphes, un graphe orienté sans cycle  $G = (V, E)$  est défini par la donnée d'un ensemble de sommets  $V$  et d'un ensemble d'arcs  $E$ , chaque arc étant un couple de sommets (par exemple, si  $x$  et  $y$  sont des sommets, les couples  $(x,y)$  et  $(y,x)$  notés respectivement  $xy$  et  $yx$  peuvent être des arcs du graphe  $G$ ) et qui ne possède pas de circuit [2].

Sa notion formalise un outil traditionnel d'analyse, dont on trouve des exemples :

en matière d'ordonnancement de projet, d'organigrammes. . .

dans tous les modèles par couches, tel que le Modèle OSI, le modèle

des fonctions du langage, de Bühler (ou de Taber-Nida), ou en psychologie la Pyramide des besoins de Maslow.

En informatique, la notion s'applique en particulier pour la représentation des termes avec partage, pour l'organisation des démonstrations en déduction naturelle ou pour la théorie des langages de l'orientation objet, en ce qui concerne la classification des types [2]. Pour un graphe sans circuit on peut construire une relation d'ordre sur les sommets, dérivée de la relation d'adjacence : un sommet  $v$  est inférieur au sommet  $w$  s'il existe un chemin  $(v, w)$  mais pas  $(w, v)$  dans le graphe. Il s'agit d'un ordre partiel : il peut y avoir des sommets qui ne sont reliés par aucun chemin, donc incomparables au sens de l'ordre. Le tri topologique n'est possible que si le graphe ne comporte pas de circuit [5].

### 1.5.2 Algorithme du Graphe sans cycle

Dans cette courte section, on considère un graphe simple orienté et on désire tester algorithmiquement si celui-ci possède ou non un cycle, condition nécessaire pour appliquer une des plus importantes applications sur ce type de graphe *Le Tri Topologique* qui consiste à ordonner les sommets d'un dag en une suite linéaire dans laquelle l'origine de chaque arc apparaît avant son extrémité.

Voici tout d'abord une condition nécessaire pour qu'un graphe soit sans circuit.

**Lemme :** Si un graphe simple orienté  $G = (V, E)$  est sans cycle, alors il existe un sommet  $v$  tel que  $d^-(v) = 0$  (resp.  $d^+(v) = 0$ ).

**Démonstration :** Considérons un chemin simple  $(x_1, \dots, x_k)$  de  $G$  de longueur maximale déterminé par des sommets de  $G$  (autrement dit, ce chemin passe par des sommets distincts et il n'est pas possible d'avoir un chemin passant par plus de sommets). Si  $d^-(x_1) > 0$ , alors il existe  $y \in \text{pred}(x_1)$ . Si  $y$  était égal à un des  $x_j$ , on aurait alors un cycle  $(y, x_1, \dots, x_j)$ , ce qui est impossible par hypothèse. Or par maximalité du chemin  $(x_1, \dots, x_k)$ , il n'est pas possible d'avoir un sommet distinct des  $x_j$  et tel que  $(y, x_1) \in E$ .

On peut en déduire une condition nécessaire et suffisante pour qu'un graphe soit sans circuit.

**Proposition :** Soit  $G = (V, E)$  un graphe simple orienté. Ce graphe est sans cycle si et seulement si il existe un sommet  $v$  tel que  $d^-(v) = 0$  et pour tout sommet  $v$  tel que  $d^-(v) = 0$ , le graphe  $G - v$  est sans cycle.



**Démonstration :** La condition est nécessaire. Si  $G$  est sans cycle, on peut appliquer le lemme précédent. De plus, tout sous-graphe  $G - v$  d'un graphe sans cycle  $G$  est bien sûr sans cycle. La condition est suffisante. Soit  $v$  un sommet tel que  $d^-(v) = 0$ . Par hypothèse,  $G - v$  est sans cycle. Par conséquent, si le graphe  $G$  possède un cycle, ce dernier doit nécessairement passer par  $v$ . Si un cycle passe par  $v$ , on en conclut que  $d^-(v) \geq 1$ . C'est impossible, donc  $G$  est dépourvu de cycle.

Ce résultat nous fournit un premier algorithme permettant de décider si un graphe est sans cycle.

Algorithme
<b>Entrées :</b> un graphe simple orienté $G = (V, E)$ <b>Sorties :</b> vrai ou faux Tant qu'il existe $v \in V$ tel que $d^-(v) = 0$ , $G := G - v$ Si $G = \emptyset$ alors sortie : "oui, $G$ sans cycle" sinon sortie : "non, $G$ possède un cycle"

**Remarque :** Si on implémente cet algorithme à l'aide de listes d'adjacence, la détection d'un sommet  $v$  pour lequel  $d^-(v) = 0$  nécessite de parcourir l'ensemble du graphe. Un tel parcours est effectué à chaque étape de la boucle. Cela a pour conséquence de fournir un algorithme dont la complexité est quadratique en fonction de  $|E| + |V|$  et celle-ci peut être améliorée.

**Théorème :** Soit  $G = (V, E)$  un graphe simple orienté. Le graphe  $G$  est sans cycle si et seulement si il est possible d'énumérer les sommets de  $V = v_1, \dots, v_n$  de manière telle que, pour tout  $i = 1, \dots, n$ , le demi-degré entrant de  $v_i$  restreint au graphe  $G_i = G - v_1 - \dots - v_{i-1}$  soit nul, ce que l'on note  $d_{G_i}^-(v_i) = 0$ .

**Démonstration.** La condition est nécessaire. Supposons  $G$  sans cycle. Ainsi, par le lemme, il existe un sommet  $v_1$  de  $G = G_1$  tel que  $d^-(v_1) = d_{G_1}^-(v_1) = 0$ . D'après la proposition précédente,  $G_1 - v_1 = G_1$  est sans cycle. Ainsi, il existe un sommet  $v_2$  tel que  $d_{G_2}^-(v_2) = 0$ . On continue de la sorte de proche en proche et on obtient l'énumération proposée.

La condition est suffisante. Supposons disposer d'une énumération des sommets ayant les propriétés indiquées. Procédons par récurrence.

rence. Le graphe  $G_n$  est restreint à l'unique sommet  $v_n$  et est donc sans cycle. Le graphe  $G_{n-1}$  contient les sommets  $v_n$  et  $v_{n-1}$ . De plus,  $d_{G_{n-1}}^-(v_{n-1}) = 0$ . En d'autres termes,  $G_{n-1}$  possède au mieux un arc de  $v_{n-1}$  à  $v_n$ . Il est donc sans cycle. Appliquons ce raisonnement pour une étape  $i$  quelconque. Si le graphe  $G_{i+1}$  est sans cycle, alors le graphe  $G_i$  se compose du graphe  $G_{i+1}$  auquel on ajoute  $v_i$  et éventuellement des arcs de  $v_i$  vers les sommets de  $G_{i+1}$ . On en conclut que  $G_i$  est sans cycle.

On déduit de ce résultat un algorithme efficace.

Algorithme
<b>Entrées</b> : un graphe simple orienté $G = (V, E)$ <b>Sorties</b> : vrai ou faux Pour tout $v \in V$ , <i>initialiser</i> $d(v)=0$ Pour tout $v \in V$ , Pour tout $w \in \text{succ}(v)$ , $d(w)=d(w)+1$ $aTraiter := \emptyset$ $nbSommet := 0$ Pour tout $v \in V$ , Si $d(v) = 0$ , alors $aTraiter = aTraiter \cup v$ $nbSommet := nbSommet + 1$ Tant que $aTraiter \neq \emptyset$ , <i>faire</i> Soit $v$ , le premier élément de $aTraiter$ $aTraiter := aTraiter \setminus v$ Pour tout $w \in \text{Succ}(v)$ , <i>faire</i> $d(w) = d(w) - 1$ si $d(w) = 0$ , alors $aTraiter = aTraiter \cup w$ $nbSommet := nbSommet + 1$ Si $nbSommet =  V $ alors sortie : "oui, G sans cycle" sinon sortie : "non, G possède un cycle"

La variable  $d$  associée à chaque sommet du graphe permet de stocker le demi-degré entrant du sommet (par rapport au graphe envisagé au moment de la construction). À chaque fois qu'un élément  $v$  est enlevé de la liste  $aTraiter$ , celui-ci est énuméré. Ainsi, on énumère d'abord les sommets de demi-degré entrant nul. Ensuite, lorsqu'un sommet est traité, on le supprime du graphe et on modifie en conséquence les demi-degrés entrants. Si tous les sommets ont été traités, cela correspond à dire que tous les sommets ont été énumérés. Au vu du théorème précédent, on en conclut que le graphe est

sans cycle.

**Définition** : Soit  $G = (V, E)$  un graphe simple orienté. Un tri topologique de  $G$  est une énumération  $v_1, \dots, v_n$  des sommets de  $G$  de manière telle que si  $(v_i, v_j)$  est un arc de  $G$ , alors  $i < j$ .

**Théorème** : Un graphe simple orienté admet un tri topologique si et seulement si il est sans cycle.

**Démonstration** : Il est clair que si un graphe possède un cycle, alors quelle que soit l'énumération de ses sommets, il ne peut s'agir d'un tri topologique. Si un graphe est sans cycle, alors une énumération de ses sommets donnant lieu à un tri topologique est immédiatement donnée par le théorème

**Remarque** : Il n'y a pas qu'un seul tri topologique pour un graphe donné  $G = (V, E)$ . En effet, si on dénote par :

$$S(G) = \{v \in V \mid d^-(v) = 0\}$$

l'ensemble des sources de  $G$ , alors l'ensemble des tris topologiques de  $G$  est donné par la formule récursive suivante :

$$\Pi(G) = \bigcup_{v \in S(G)} \{v.\sigma \mid \sigma \in \Pi(G - v)\}$$

où  $\sigma$  est un tri topologique de  $G - v$  et où  $v.\sigma$  désigne l'énumération des sommets de  $G$  en débutant par  $v$  puis en suivant l'énumération prescrite par  $\sigma$ .

### 1.5.3 Algorithme de tri topologique

Tous ces résultats nous permettent de conclure un algorithme de tri topologique :

**Algorithme Tri Topologique****Entrées :**  $G$  graphe simple orienté sans cycle  $G = (V, E)$ **Sorties :**  $L$  liste des sommets de  $G$  ordonnés{construire l'ensemble  $E$  et la table des prédécesseurs} $E \leftarrow \emptyset$ ;**pour tout**  $x$  de  $G$  **faire** $\text{nbpred}[x] \leftarrow d^-(x)$ **si**  $\text{nbpred}[x]=0$  **alors**  $E \leftarrow E \cup \{x\}$  **fin si****fin pour**{ $E$  contient au moins un sommet}**tant que**  $E \neq \emptyset$ ; **faire**{Invariant :  $\forall x \in E, \text{nbpred}[x] = 0 \Rightarrow x \in L$ }soit  $s \in E$ ajouter( $L, s$ ) $E \leftarrow E - \{s\}$ **pour tout**  $x$  de  $G$  adjacent à  $s$  **faire**{Invariant :  $x \notin L$ } $\text{nbpred}[x] \leftarrow \text{nbpred}[x] - 1$ **si**  $\text{nbpred}[x]=0$  **alors**  $E \leftarrow E \cup \{x\}$  **fin si****fin pour****fin tant que**{ $L$  contient tous les sommets de  $G$  ordonnés}**rendre**  $L$ **1.5.4 Applications du tri topologique****Make**

make est un utilitaire mis au point en 1977 par Stuart Feldman permettant d'automatiser la génération de fichiers à partir de règles simples (et éventuellement en faisant appel à d'autres programmes ou à la ligne de commande). Il est souvent utilisé pour fabriquer des fichiers exécutables ou des bibliothèques à partir des fichiers sources et d'un compilateur. Les projets de grande envergure qui contiennent un nombre important de fichiers ont rendu indispensable l'utilisation de ce type d'outils. Le (ou les) fichiers que l'on cherche à produire (appelés fichiers cibles) dépendent de la fabrication de plusieurs autres fichiers "intermédiaires", qui eux même peuvent dépendre de la création de plusieurs autres fichiers, et ainsi de suite jusqu'aux fichiers qui étaient déjà présents au début du processus (les fichiers source). Les dépendances entre les différents fichiers, et la liste de commande permettant de créer chacun d'eux, sont spécifiées par le programmeur dans un fichier appelé Makefile

que le programme make va lire.

### **Implémentation de la base de données**

En se trouve parfois devant des bases énormes ou les clés étrangères sont très difficile à les organiser pour ne pas tomber dans l'erreur :

Error 1215 : Cannot add foreign key constraint

Explication : on ajoute une clé étrangère qui référence un tableau pas encore implémenté. La possibilité de représenter la relation des dépendances entre les différents tableaux ou un ordre partiel persiste entre une table et un autre ayant une clé étrangère qui référence sur le premier.

## **1.6 conclusion**

Dans cette première partie nous avons introduit le vocabulaire ainsi que les notions et notations nécessaires à la compréhension des graphes utilisés dans le chapitre 2 et 3.

Deuxième partie

Réalisation

## Chapitre 2

# Réalisation en c

### 2.1 Introduction

Dans ce chapitre nous implémentons l'algorithme de tri topologique en C. Le C est un langage de programmation impératif et généraliste. Inventé au début des années 1970, C est devenu un des langages les plus utilisés. Nous essayons alors d'implémenter un graphe et le trier l'aide des structures de données offertes par ce langage.

#### 2.1.1 Principe du tri topologique

C'est un processus de tri d'éléments sur lesquels n'est définie qu'une relation d'ordre partiel, c'est-à-dire que la relation d'ordre est connue pour certaines paires d'éléments, mais pas pour toutes. Plus généralement, une relation d'ordre partiel sur un ensemble  $S$  est une relation entre les éléments de  $S$ . Elle est dénotée par le symbole  $<$  énoncé « précède », et elle satisfait aux propriétés (ou axiomes) suivantes, quel que soit le triplet d'éléments distincts  $x, y, z$  de  $S$  :

- (1) si  $x < y$  et  $y < z$  alors  $x < z$  (transitivité)
- (2) si  $x < y$ , alors non  $y < x$  (asymétrie)
- (3) non  $z < z$  (irréflexivité)

**Exemple** Nous disposons de l'ensemble des tâches :

$\{T1, T2, T3, T4, T5, T6, T7, T8, T9, T10\}$

et de l'ensemble des relations  $\{T1 < T2; T2 < T4; T4 < T6; T2 < T10; T4 < T8; T6 < T3; T1 < T3; T3 < T5; T5 < T8; T7 < T5; T7 < T9; T9 < T4; T9 < T10\}$

Le tri topologique associé est :  $T1, T2, T7, T9, T4, T6, T3, T5, T8, T10$

A ces descriptions formelles nous associons un graphe des tâches qui respecte les relations d'ordre (il existe donc un graphe avant tri et

un graphe après tri), ce graphe est orienté. Voici les graphes avant et après tri de l'exemple précédent :  
 graphe initial :

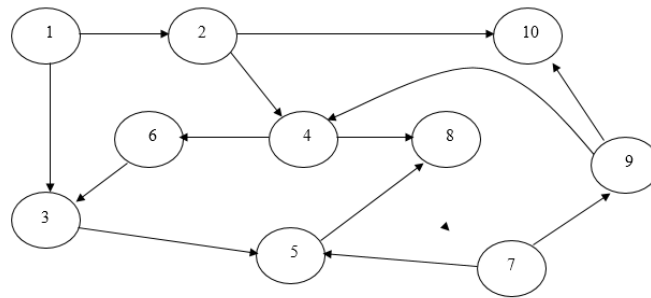


FIGURE 2.1 – Graphe représentant l'ensemble S

Puis après le tri topologique :

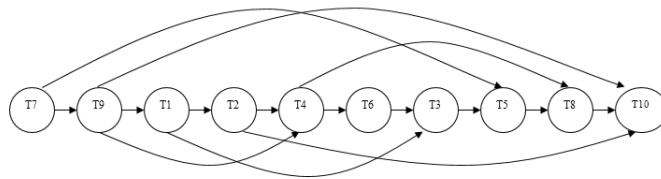


FIGURE 2.2 – Résultat 1 après tri topologique sur S

ou encore :

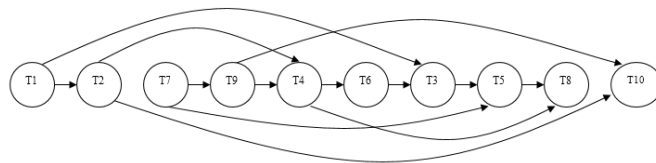


FIGURE 2.3 – Résultat 2 après tri topologique sur S



## 2.2 Algorithme proposé

Nous commençons par choisir un élément qui n'est précédé par aucun autre (il doit y en avoir au moins un, sinon il y aurait une boucle). Cet élément est placé en tête de la liste résultante et retiré de l'ensemble  $S$ . L'ensemble restant est toujours partiellement ordonné; nous pouvons donc lui appliquer le même algorithme, et ainsi de suite jusqu'à ce qu'il (ensemble  $S$ ) soit vide. Bien entendu, la suppression d'un élément  $s$  de  $S$  entraîne la suppression de toutes les relations de la forme  $s < x$ .

Remarque importante : Les propriétés (1) et (2) de l'ordre partiel garantissent que le graphe ne contient pas de cycle. C'est exactement la condition nécessaire pour que l'inclusion dans un ordre linéaire soit possible. A propos de la propriété (1) : transitivité

Cas1 : respect de la transitivité

$1 < 2 \quad 2 < 4$



FIGURE 2.4 – Respect de la transitivité

Cas2 : non respect de la transitivité



FIGURE 2.5 – Non respect de la transitivité

A propos de la propriété 2 : asymétrique

Cas1 : respect  $1 < 2$



FIGURE 2.6 – Respect  $1 < 2$

Cas2 : non respect  $1 < 2 < 1$

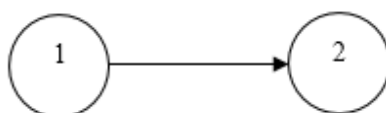


FIGURE 2.7 – Non respect  $1 < 2 < 1$

### 2.2.1 Choix d'une structure de données

Áfin de pouvoir décrire l'algorithme proposé ci-dessus de manière plus rigoureuse, nous allons choisir une structure de données pour représenter  $S$  et la relation d'ordre. Un tel choix est conditionné par les opérations que nous devons effectuer, et en particulier par les opérations de sélection d'un élément sans prédécesseur. Nous distinguons deux opérations fondamentales :

- Trouver les éléments de  $S$  sans prédécesseurs
- déterminer l'ordre linéaire des éléments de  $S$ . Un tel traitement (ou opération) consiste à choisir un élément ( $s$ ) appartenant à l'ensemble des éléments sans prédécesseurs, à supprimer l'élément choisi de  $S$  et les relations de la forme  $s < x$ .

Puisque le nombre  $n$  d'éléments de  $S$  n'est pas connu a priori, il est pratique d'organiser l'ensemble sous forme d'une liste linéaire. Chaque élément (ou nœud) de cette liste est représenté par les champs suivants :

- une clé d'identification
- l'ensemble de ses successeurs
- le nombre de ses prédécesseurs

—pointeur vers l'élément suivant dans la liste. De même l'ensemble des successeurs d'un élément peut être représenté par une liste linéaire  $v$

Illustration :

$S = \{A, B, C, D, E\}$

Les relations d'ordre définies sur  $S$  sont :

$A < D$   $A < B$   $A < C$   $D < C$   $B < E$   $C < E$

Ces relations sont définies par le graphe orienté sans circuits (appelé DAG :Directed Acyclic Graphs) suivant :

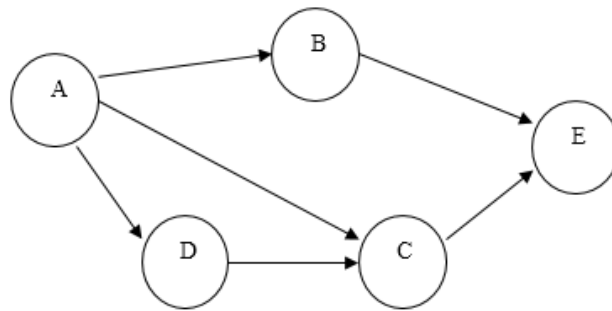


FIGURE 2.8 – DAG représentant  $S$

Un tel graphe est traduit par la représentation physique suivante conformément aux choix faits précédemment :

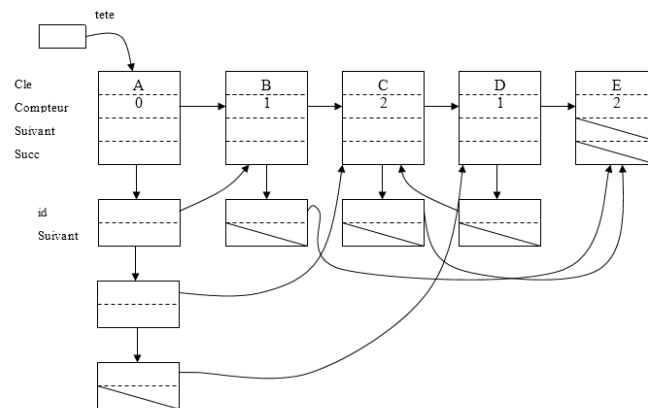


FIGURE 2.9 – Représentation physique du graphe en  $C$

Compteur représente le nombre des prédécesseurs d'un élément de  $S$ .

Succ représente la liste des successeurs d'un élément de S chaque nœud de cette liste comporte deux champs :

-Id pointe sur un élément de S : le successeur d'un élément de S est un élément de S.

-Suivant pointe sur le suivant de cette liste que nous pouvons qualifier de liste secondaire par rapport à la principale.

## 2.3 Traduction en C

```
Struct noeud_tete
{
    unsigned cle;\*Nous supposons que les clés sont des
    nombres entiers mais pas nécessairement des entiers
    consécutifs de 1 à n\*
    unsigned compteur;\*nombre de prédécesseurs qui va-
    rie dynamiquement ,il décroît\*
    struct noeud_tete *suivant;
    struct noeud_queue
    { struct noeud_tete *id;
      struct noeud_queue *suivant;
    }*succ;
};
```

Opération : sans prédécesseurs

Il semble raisonnable de regrouper tous ces éléments en une liste linéaire. Puisque la liste principale des têtes ne servira plus par la suite, nous pouvons utiliser le champ suivant pour relier toutes les têtes qui ont zéro prédécesseur.

Pour des raisons pratiques, nous construisons la nouvelle liste en ordre inverse.

```

Struct   noeud_tete   *sans_pred   (struct
noeud_tete *tete)
struct noeud_tete *p; \*suivant\*
struct noeud_tete *q; \*courant\*
p = tete;
tete = NULL;
While(p)
{ q = p;
  p = q → suivant;
  if(q == 0)
  { q → suivant = tete;
    tete = q;
  }
}
return tete;
}

```

Programmation de l'opération ordre linéaire :

```

Void   ordre_linéaire   (struct   noeud_tete
*sans_p)
\*sans_p est une liste sans prédécesseur\*
{struct noeud_queue *q;
 struct noeud_tete *t;
 while (sans_p)
 { printf( "%u ",sans_p → cle);
   q = sans_p → succ;
   t = sans_p;
   sans_p = sans_p → suivant;
   \*MAJ des successeurs\*
   while(q)
   {
     q → id → compteur = q → id → compteur - 1;
     if(q → id → compteur == 0)
     { q → id → suivant = sans_p;
       sans_p = q → id;
     }
   }
 }
}

```

```

    q = q → suivant ;
} \*fin while englobé \*
supprimer_queue(t → succ) ;
free(t) ;
} \*fin ordre linéaire \*
void supprimer_queue(struct noeud_queue *s)
{ struct noeud_queue *courant ;
  struct noeud_queue *successeur ;
  courant = s ;
  while(courant)
  { successeur = courant → suivant ;
    free(courant) ;
    courant = successeur ;
  }
}

```

Opération de création :

Une telle opération peut être décomposée comme suit :

- sous programme engendré

Il permet d'engendrer une liste(struct noeud\_tete \*tete) à partir d'un fichier texte .Ce dernier est censé stocker l'ensemble des relations définies sur S. Chaque relation a le format suivant : Cle<cle<rc> avec cle une valeur entière non signée qui caractérise un élément appartenant à S ,< symbolise "précède" et <rc> est le caractère retour chariot ou fin de ligne(entrée ou return).

Voici l'interface ou la signature du sous-programme engendré.

Struct noeud\_tete \*generation(char \*fexterne) :

Elle permet de créer une liste matérialisée par un pointeur struct noeud\_tete\* à partir d'un fichier externe ayant un format bien déterminé

- sousprogramme paire : il traite une paire d'éléments appartenant à S reliés par une relation. Voici l'interface ou la signature du sous-programme paire :

void paire(struct noeud\_tete \*s, unsigned x, unsigned y) :

s est une liste non vide x et y forment une relation( $x < y$ ).

Paire permet d'ajouter cette relation à la liste s.

Programmation de la procédure paire :

Une telle procédure peut être décomposée comme suit :

-la fonction trouve\_tete permettant de localiser un élément donné (clé) dans une liste donnée (struct noeud\_tete \*)

struct noeud\_tete \*trouve\_tete (struct noeud\_tete \*s, unsigned cle) :

si *cle* appartient à *s* alors nous rendons la référence du *cle* dans *s*  
sinon nous ajoutons *cle* à la fin de *s* et nous rendons également sa référence

-la procédure *insérer\_queue* ayant l'interface suivante :

`void insérer_queue (struct noeud_tete * s, struct noeud_tete *p) :`

Elle insère *p* en tête de la liste du successeur  $s \rightarrow succ$

En appliquant une démarche de programmation descendante, la programmation de la procédure *paire* est donnée comme suit :

```
Void paire( struct noeud_tete *s, unsigned x ,
unsigned y )
{ struct noeud_tete *p; /*reference de x */
  struct noeud_tete *q; /*reference de y */
  p=trouve_tete(s,x);
  q=trouve_tete(s,y);
  insérer_queue(p,q);
   $q \rightarrow compteur = q \rightarrow compteur + 1;$ 
}
```

Programmation *trouve\_tete* et *insérer\_queue* :

```
Structnoeud_tete*trouve_tete(structnoeud_tete*
s, unsignedcle)
{structnoeud_tete * p; \*pour parcourir s \*
  structnoeud_tete * q; \*allocation \*
   $p = s;$ 
  while( $p \rightarrow cle \neq cle$ )

  ( $p \rightarrow suivant \neq null$ )
   $p = p \rightarrow suivant;$ 
  if( $p \rightarrow cle == cle$ )
  return p; \*success\*
```

```

else
{ \*adjunction \*
q=(struct noeud_tete *) malloc(sizeof(struct
noeud_tete))
q → cle = cle;
q → compteur = 0;
q → succ = null;
\*chaînage\*
p = q;
} return q ; \*echec + création\*
}
void inserer_queue(struct noeud_tete *s,struct
noeud_tete *p)
{ \*insertion en tête\*
structnoeud_queue *q;
q=(struct noeud_queue *)malloc(sizeof(struct
noeud_queue))
q → id = p;
q → suivant = s → succ;
s → succ = q;
}

```

## Programmation génération

Nous avons besoin d'un sous\_programme permettant de traiter la première ligne d'une façon indépendante. Un tel sous\_programme permet d'initialiser le processus de génération de la liste (struct noeud\_tete \*)

```

Struct noeud_tete *initialisation(unsigned
cle1,unsigned cle2)
\*cle1<cle2\*
{ struct noeud_tete *s;\*à rendre\*
struct noeud_tete *p;
struct noeud_queue *q;

```



```

/*premier élément */
p=( struct noeud_tete *)malloc(sizeof(struct
noeud_tete ));
p-> cle = cle1;
p-> compteur = 0;
s=p;
/*deuxième élément*/
p=( struct noeud_tete *)malloc(sizeof(struct
noeud_tete ))
p-> cle = cle2;
p-> compteur = 1;
p-> succ = null;
p-> suivant = null;
/*chainage*/
s-> suivant = p;
/*suceesseurs */
q=( struct noeud_queue *)malloc(sizeof(struct
noeud_queue))
q-> id = p;
q-> suivant = null;
s-> succ = q;
return s;
}

```

## 2.4 Conclusion

Ainsi nous avons pu mettre en oeuvre dans ce programme l'algorithme du tri topologique, la complexité a été considérablement limitée en effectuant un parcours du graphe qui effectue sans parcours supplémentaire la recherche de circuits.

De plus l'utilisation de pointeurs dans la liste des successeurs a permis de limiter le nombre d'opérations, le temps de calcul, et en même temps de se rapprocher de la structure réelle d'un graphe (un pointeur représentant un arc).

## Chapitre 3

# Réalisation en java

### 3.1 Introduction

Après avoir implémenté le problème de tri topologique par un langage de programmation procédurale, on a pu constater ses limitations. Nous avons essayé de se rapprocher de la notion graphe ce n'était pas assez clair, d'autant plus que c'est à l'utilisateur d'entrer la liste des dépendances dans un fichier externe. tout ça nous mène à chercher une autre solution avec un langage plus performant et puissant pour se rapprocher du réel et optimiser la complexité : Le langage orienté objet java.

Dans ce chapitre nous allons réaliser un programme pour le tri topologique associé à une interface pour dessiner les graphes en entrés.

#### 3.1.1 Conception de l'application

Diagramme du classe

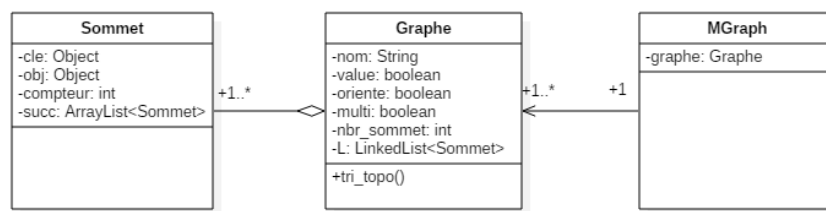


FIGURE 3.1 – Diagramme du classe

## 3.2 Réalisation de l'application

### 3.2.1 Implémentation des classes

#### La classe Graphe

La classe "Graphe" contient plusieurs attributs nécessaires au bon fonctionnement de l'application comme des booléens :

private boolean oriente;

private boolean value;

Ceux-ci permettent respectivement de déterminer si le graphe est orienté ou valué. La classe contient aussi des autres attributs pour stocker les sommets, leur nombre, stocker et afficher le résultat du tri ainsi que le nom du graphe (pas vraiment nécessaire) :

private int nbr\_sommet;

private LinkedList<Sommet> L;

private String nom;

Le constructeur construit un graphe selon les paramètres insérés : nom, value et oriente

Un autre constructeur sans paramètres est aussi codé.

```
public Graphe(String nom, boolean value,
boolean oriente)
{
    this.nom = nom;
    this.value = value;
    this.oriente = oriente;
    this.L = new LinkedList<Sommet>();
    this.nbr_sommet=0;
}
```

Les méthodes pour ajouter et supprimer un sommet sont respectivement :

```
public void ajouterSommet(Sommet s)
{
    L.add(s);
    nbr_sommet ++; }
public void supprimerSommet(Sommet s)
{
    L.remove(s);
    nbr_sommet --;
}
```

Nous pouvons obtenir un Sommet par l'appel de la méthode `get(int index)` ou `getFirst()` ou `getLast()`, 3 méthodes spéciales à la structure `LinkedList<Sommet>` pour accéder à ses données (ici des objets de type `Sommet`) , elle prend l'objet qui se trouve à l'indice "index" dans la liste des sommets et le renvoi.

```
Sommet getSommet(index i)
{
    L.get(i);
}
```

Les méthodes permettant de changer les propriétés du graphe sont :

- `public void setOriente(boolean bool)` qui permet de changer le graphe en graphe orienté si `bool` est vrai (`true`) ou non orienté si `bool` est faux (`false`).
- `public void setValue(boolean value)` qui permet de changer le graphe en graphe valué si `value` est vrai (`true`), non valué (`false`) sinon.
- `public void setNom(String nom)` qui permet de changer le nom du graphe.

**Algorithme de tri** Nous s'intéressons dans cette partie à résoudre le problème du tri topologique. En premier, nous devons tester la conformité de notre graphe pour subir le tri, il devrait être orienté et acyclique :

- Pour vérifier que le graphe est orienté, on doit tester l'attribut booléen `oriente` s'il est vrai alors le graphe est orienté.
- Pour vérifier que le graphe est acyclique, on doit prouver pour chaque sommet constituant le graphe que la chaîne commençant par lui ne soit pas fermée.

Ce test est établi par la fonction booléenne `boolean test_Boucle_unitaire(Sommet t, ArrayList<Sommet> succ)` qui prend en paramètre un sommet et une liste des successeurs :

```

boolean test__Boucle_unitaire(Sommet
t, ArrayList<Sommet> succ)
{
  if (succ.isEmpty()) {
    return false;
  } else {
    Iterator<Sommet> it = succ.iterator();
    while (it.hasNext())
      Sommet t1 = it.next();
    Iterator<Sommet>          it1          =
    t1.getSucc().iterator();
    while (it1.hasNext())
      Sommet t2 = it1.next();
    if (t2.equals(t) {
      return true;
    } else { return test__Boucle_unitaire(t,
      t2.getSucc());
    }
  }
}

```

Ensuite la fonction boolean test\_\_Boucle prend en charge de tester tous les sommets du graphe comme suit :

```

boolean test__Boucle()
{
  Iterator<Sommet>          it          =
  this.getL().iterator();
  while (it.hasNext()) {
    Sommet t = it.next();
    if (test__Boucle_unitaire(t, t.getSucc())) {
      return true; }
    }
  return false;
}

```

Après la vérification de la conformité de notre graphe, nous cherchons ainsi à implémenter l'algorithme du tri topologique qui consiste à :

1. chercher les sommets dont  $d^-(x) = 0$  ou encore les sommets qui n'avaient pas des prédécesseurs, on devrait disposer alors d'une fonction `LinkedList< Sommet > sans_pred()` qui nous rend une liste

des sommets dont ils n'avaient pas des prédécesseurs (la liste E présenté dans l'algorithme du tri chapitre Graphe)

```
LinkedList<Sommet> sans_pred() {
    LinkedList<Sommet> sans = new Linked-
    List<Sommet>();
    Iterator<Sommet> it = L.iterator();
    while (it.hasNext()) {
        Sommet t = it.next();
        if (t.getCompteur() == 0) {
            sans.add(t);
        }
    }
    return sans;
}
```

**2.** la méthode void tri\_topo() nous permettons de trier le graphe selon les procédés :

Tant que la liste L soit non vide et pour chaque sommet s dans E (la liste des sommets sans predecesseurs) on parcourrait sa liste des successeurs pour décrémente  $d^-(x)$ , ensuite supprimer s de la liste des sommets.

```
void tri_topo()
{
    int comp = 0;
    while (!this.getL().isEmpty()) {

        Sommet t = this.sans_pred().pop();
        Iterator<Sommet> it1 = t.getSucc().iterator();
        L.remove(t);
        while (it1.hasNext()) {
            Sommet tsucc = it1.next();
            comp = tsucc.getCompteur();
            tsucc.setCompteur(-comp);
        }
    }
}
```

### La classe Sommet

La classe "Sommet" est implémentée pour modéliser les noeuds d'un graphe. elle a plusieurs attributs pour s'approcher et mieux modéliser un noeud :

<code>private Object cle;</code>	Son identifiant unique
<code>private String nom;</code>	Son nom
<code>private Object obj;</code>	Son implémentation graphique
<code>private int compteur;</code>	Son nombre de prédécesseurs
<code>private ArrayList &lt; Sommet &gt; succ;</code>	Sa liste des adjascences

Le constructeur construit un Sommet dont la clé et l'objet sont insérés par l'interface MGraph et le compteur (le nombre des sommets dont il est l'arrivé) sera initialisé à zéro ainsi que l'instanciation de la liste des successeurs :

```
public Sommet(Object cle, Object obj)
{
    this.cle = cle;
    this.obj = obj;
    this.compteur=0;
    this.succ=new ArrayList<Sommet>();
}
```

Nous pouvons modifier le nom du sommet à partir du l'interface MGraph et du la méthode d'édition suivante :

```
public void setNom(String nom)
{
    this.nom = nom;
}
```

Les méthodes pour ajouter et supprimer un successeur sont respectivement :

```

public void ajouterSucc(Sommet s)
{
    int comp = s.getCompteur();
    succ.add(s);
    s.setCompteur(++comp);
}

public void supprimerSucc(Sommet s)
{
    int comp = s.getCompteur();
    succ.remove(s);
    s.setCompteur(--comp);
}

```

Nous pouvons obtenir un successeur par l'appel de la méthode `get(int index)` offerte par la structure de donnée `ArrayList<Sommet>` pour accéder à ses données (ici des objets de type `Sommet`) , elle prend l'objet qui se trouve à l'indice "index" dans la liste des successeurs et le renvoi.

```

Sommet getSuccessee(index i)
{
    succ.get(i);
}

```

### 3.2.2 Représentation de l'interface

La représentation graphique du graphe se fait grâce à la classe `Mgraph`. Nous pouvons imaginer cet objet comme un panneau sur lequel sont disposés les sommets et les connexions qui composent le graphe. La figure : Figure 3.2 présente notre interface principale.

#### Initialiser le `MGraph`

`Mgraph` dispose d'un attribut `graphe` qui est un objet de type `Graphe` :



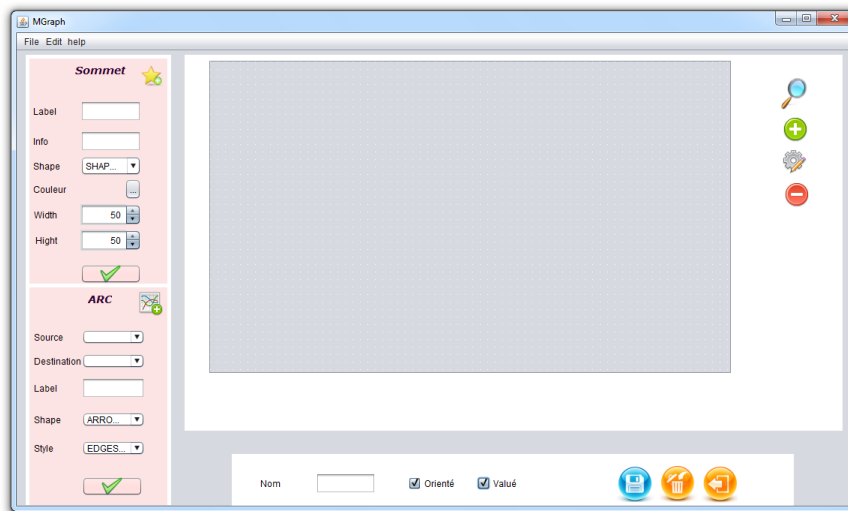


FIGURE 3.2 – Interface Mgraph

```
Mgraph()
{
  this.graphe=new Graphe();
}
```

### Dessiner un sommet

Dessiner un sommet, ou encore un MSommet pour ne pas confondre entre l'objet Sommet de la classe Sommet et sa représentation graphique de type Objet que nous avons choisi d'appeler MSommet, est tellement simple avec un panel qui s'affiche à gauche de notre panneau du dessin en appuyant sur un bouton "+" ou bien dans la barre de menu, le menu ajouter Sommet permet de changer ce panel en une fenêtre pour changer sa position, la dimensionner et la réduire afin d'augmenter la flexibilité des outils de dessin pour l'utilisateur. Cet outil nous offre une multitude d'options pour dessiner un MSommet, nous pouvons absolument faire tout : choisir le nom, la taille : hauteur et largeur, la forme, la couleur et l'information qui sera portée par lui.

—nom est un JTextField, tout ce qui est inséré subit un cast en type Objet qui sera par la suite la clé du Sommet insérée dans la liste L du Graphe graphe.

—height est un JSpinner par défaut est à 50 permet de définir la hauteur du MSommet.

—width est un JSpinner par défaut est à 50 permet de définir la

largeur du MSommet.

—shape est un Jcombobox qui offre de différentes formes pour présenter les MSommets selon le besoin du l'utilisateur (ellipse,rectangle,actor,cylinder,DoubleEllipse,hexagon,rhombus,swimlane,triangle,cloud).

Le tableau : Table 3.1 montre ces différentes formes

Exemple : actor pour présenter des personnes, cylinder pour présenter une hiérarchie de dépendances entre des bases de données...

	Forme du triangle
	Forme du cloud
	Forme du hexagon
	Forme doubleEllipse
	Forme du swimlane
	Forme du rhombus
	Forme du actor
	Forme du ellipse
	Forme du rectangle
	Forme du cylinder

TABLE 3.1 – Différents forme du MSommet

—couleur est une fenêtre de dialogue qui s'ouvre pour choisir le couleur désiré comme l'indique la figure : Figure 3.3

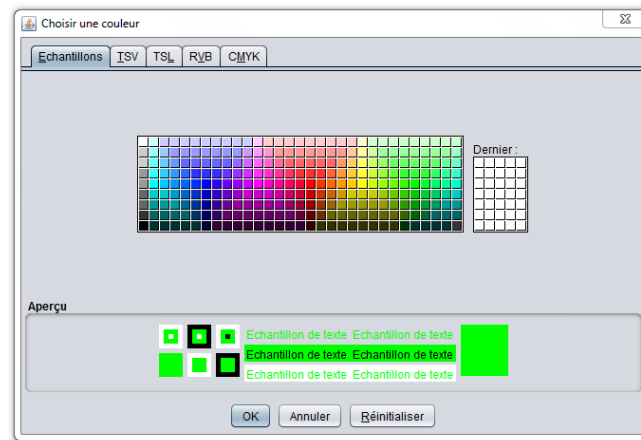


FIGURE 3.3 – Choix du couleur

Enfin le bouton dont l'icone est une flèche de validation  $\checkmark$  est responsable des actions suivantes :

1. un Objet `obj` est créé, il porte tous les informations concernant `MSommet` (référence, style, forme, information..) puis affiche le sommet dans notre panneau de dessin selon une position  $(x,y)$  choisie par le programme et selon les caractéristique choisies.
2. créer un objet de type `Sommet` dont la clé est le nom casted en type objet et l'objet `obj` est celui décrit en 1.
3. insérer le `Sommet` créé dans la liste `L` des sommet du Graphe `graphe` :

```
graphe.getL().add(new Sommet(label,obj));
```

4. incrémenter l'attribut `nbr_sommet` du Graphe `graphe`.

```
int nbr = graph.getNbr_sommet ;
graph.setNbr_sommet(++nbr)
```

## Dessiner une connexion

**Dynamiquement** Pour dessiner une connexion comme pour un sommet on a deux possibilités soit un panel qui s'affiche à gauche en appuyant au bouton "+" soit dans la fenêtre AjouterArc accédée à partir du menu ajouterArc.

Cette fenêtre nous offre aussi des propriétés et des caractéristiques afin de dessiner des connexions plus performantes, représentantes et particulières

Ce panel contient les champs suivants :

- source une liste des MSommets existants, le MSommet choisi est considéré comme l'origine d'arc
- destination une liste des MSommets existants, le MSommet choisi est considéré comme l'arrivé d'arc
- label si le graphe est valué ce champ est accessible et l'arc porte ainsi le texte entré.
- shape est un JComboBox d'une liste des styles de la fin de la flèche, ainsi nous pouvons choisir la forme de la fin de la flèche selon le besoin, le goût et le désir de l'utilisateur comme l'indique le tableau : Table 3.2





	Forme oval
	Forme diamond
	Forme open
	Forme classic

TABLE 3.2 – Différentes forme de la fin de la flèche d'une connexion

—style, l'arc peut être sous plusieurs forme , ainsi style est un JComboBox pour choisir la forme d'arc. Ces différentes formes sont indiquées dans le tableau Table :3.3

	Style du TopBottom
	Style du SideToSide
	Style du EntityRelation
	Style du Loop

TABLE 3.3 – Différentes forme d'une connexion

Enfin le bouton dont l'icone est une flèche de validation  $\checkmark$  est responsable d'enregistrer tous ces paramètres et affiche la connexion dans le panneau du dessin.

**Manuellement** Afin d'accroître la souplesse et l'adaptabilité de notre application nous offrons la possibilité de dessiner manuellement des connexions, par un simple clic sur le MSommet dans notre panneau un arc en sort et suit notre curseur vers la destination désirée tout en respectant le type de graphe si orienté alors un arc avec un flèche comme terminaison sinon une simple ligne.

**Sauvgard** Une fois nous avons terminé le dessin de toutes les connexions le bouton "sauvegarde " prend en charge de traduire ces flèches des instanciations d'objet, en fait à chaque fois que ce bouton est tapé tous les Sommets enregistrés dans la liste liée L du Graphe graphe sont mis à jour :

1. la liste des successeurs des Sommets enregistré dans L sera vidée pour garder la cohérence et éviter à chaque fois d'ajouter des successeurs existants.
2. Notre panneau est analysé et toutes les connexions sont renvoyées.
3. le Sommet du MSommet destination subit les modifications suivantes : son attribut compteur s'incrémente.
4. le Sommet du MSommet d'origine insère le Sommet du MSommet destination dans sa liste du successeurs.

Remarque :

- dans le cas d'un graphe non orienté le sommet source est le sommet dont il est l'origine de la ligne de connexion.
- il existe absolument la possibilité d'avoir une boucle en choisissant dans la source et la destination la même entrée.
- les connexions arbitraires dont la source ou la destination est perdue ne seront pas sauvegardées, le programme les ignorent.

### Supprimer un composant

la suppression d'un composant soit un MSommet soit une connexion n'est pas compliquée, il suffit de sélectionner les composants que nous voudrions supprimer et cliquer par la suite sur le bouton à droite du panneau qui porte l'icône "-".

La suppression entraîne non seulement la disparition des composants mais aussi au niveau du Graphe les modifications suivantes :

- si le composant est une connexion le Sommet du Msommet\_destination est supprimé de la liste des successeurs du Sommet du MSommet\_origine et son attribut compteur est desincrémenté.
- si le composant est un noeud, il est supprimé de la liste liée L des Sommet du Graphe et toutes les connexions dont il est l'origine sont supprimées de la même façon décrite ci-dessus, ainsi que l'attribut nbr\_sommet est desincrémenté.

### Supprimer le graphe

Grâce au bouton au bas du panneau nous avons la possibilité de supprimer le graphe entier :

- le panneau serait vidé.
- le Graphe est supprimé et initialisé de nouveau.

### Editer un composant

**MSommet** pour un sommet on a vu que nous pouvons modifier le nom à travers la méthode setCle().

A travers notre panneau on peut changer le nom d'un MSommet en y double cliquant puis on le sélectionne et on appuie sur le bouton à gauche dont l'icône est un symbole d'édition, pour enregistrer la modification et pour que le Sommet soit aussi modifié à travers la méthode setCle() décrite ci-dessus.

**Graphe** Nous pouvons modifier à tout moment les caractéristiques du graphe :

—cocher la boîte valué, le Grphe graphe sera valué.

```
graphe.setValue(true)
```

—cocher la boîte orienté, le Grphe graphe sera orienté.

```
graphe.setOriente(true)
```

—Changer le nom du graphe dans le JTextField nom au-dessous du panneau.

```
graphe.setNom(le_nom_entré)
```

### Visualiser le graphe

notre application offre la possibilité d'un "*quick view* " : une représentation simplifiée de notre MGraph les MSommets sont de simples points en gras avec leurs noms respectifs.

Quick view permet de vérifier le bon fonctionnement des opérations d'ajout, d'édition...

### Trier le graphe

Grâce à un bouton dans la barre Menu tri topologique, la méthode tri\_topo du Graphe graphe se lance

```
graphe.tri_topo()
```

Si tout marche bien (graphe orienté sans boucle) le résultat s'affiche dans une nouvelle fenêtre "tri topologique" et un graphe s'affiche montrant le nouveau MGraph trié.

Le bouton Help de la barre menu contient une définition simple du tri topologique pour aider l'utilisateur à comprendre comme l'indique la figure : Figure 3.4.

### Sauvegarde du résultat

Nous pouvons sauvegarder le résultat du tri décrit au-dessus dans un fichier texte avec les informations sur le Graphe graphe trié.

Une fenêtre s'ouvre pour choisir l'emplacement du fichier comme le montre la figure : Figure3.5.

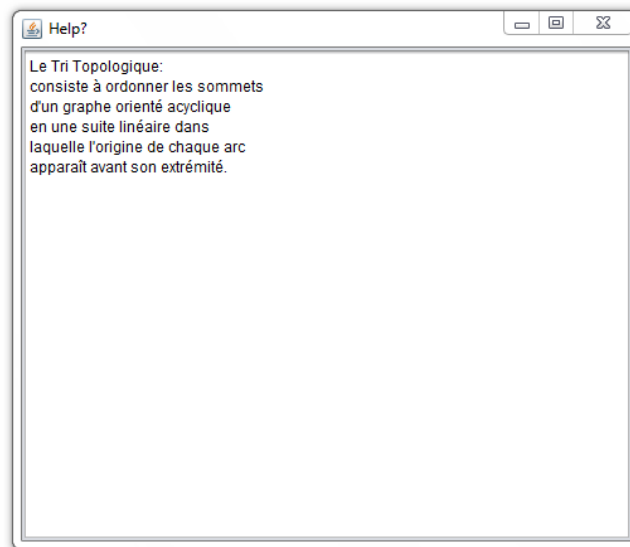


FIGURE 3.4 – Help

### Imprimer le graphe

Nous avons aussi la possibilité de sauvegarder le MGraph sous forme d'une image <.png> et l'imprimer ensuite. c'est pour permettre à l'utilisateur exploiter ces graphes dessinés dans d'autres applications (Biologiques, Technologiques, Sociaux, Educative : "e-learn"...) )

### 3.2.3 Teste de l'application

**Dessiner le graphe** Nous proposons de traiter le problème de compilateur make, nous devons enter le graphe de dépendance entre les différents fichiers d'un projet. Nous commencons ainsi à dessiner les MSommet dont les labels sont les noms des fichiers comme le montre la figure : Figure3.6, ensuite nous dessinons les connexions entre les sommets avec des relations de dépendance comme le montre la figure : Figure 3.7.

Une fois le graphe dessiné, un clic sur le bouton de sauvegarde et l'objet Graphe graph est bien instancié et notre graphe est prêt pour l'opération du tri.



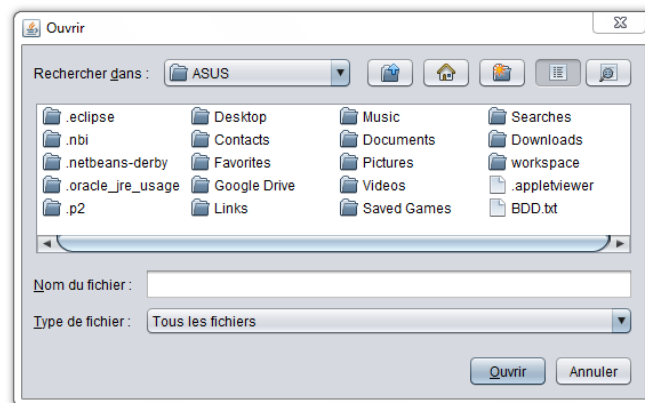


FIGURE 3.5 – Sauvgarde du résultat

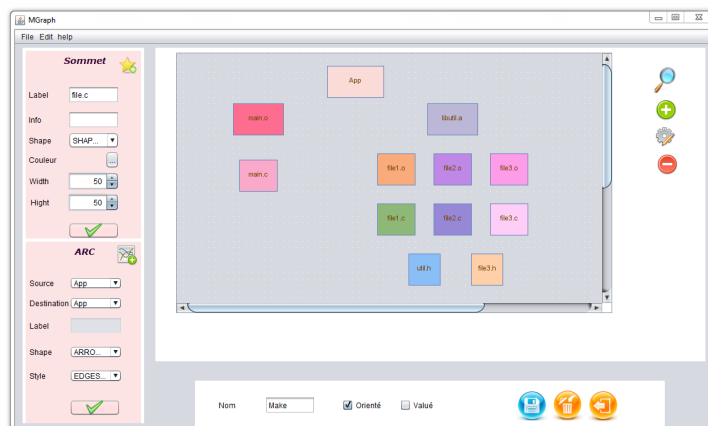


FIGURE 3.6 – MSommets dessinés

**Visualiser le graphe** À chaque fois que nous voulons tester que nous avons bien entré nos données et que toutes les connexions sont bien validées le bouton Quick view prend en charge ce test et nous rend à chaque fois notre graphe simplifié avec tous les sommets et connexions. Exemple dans la figure : Figure 3.8

**Trier le graphe** Après le dessin et la vérification du graphe, la méthode `tri_topo()` est lancée et elle nous rend soit une fenêtre d'erreur s'il s'agit d'un graphe cyclique comme l'indique l'exemple de la figure : Figure3.11 ou non orienté comme l'indique l'exemple de la figure : Figure3.12. Sinon elle nous rend le graphe trié où les sommets sont alignés dans leur ordre linéaire comme l'exemple de la

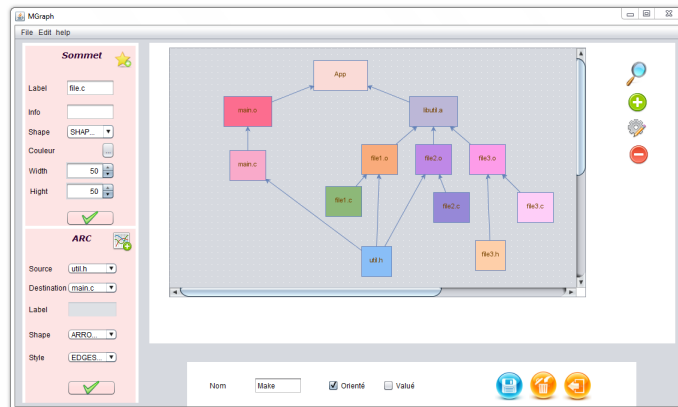


FIGURE 3.7 – MGraph du Make dessiné

figure : Figure3.9

Puisque un Dag peut avoir plusieurs résultat de tri, Notre exemple peut nous rendre des résultats différents mais biensûr corrects à chaque fois nous lancons l'opération de tri comme le montre le figure : Figure 3.10 qui est un autre tri topologique de même exemple entrée Make.

**Sauvgarder le résultat** La sauvegarde de résultat se fait dans un fichier texte et comprend les informations sur le graphe entré et le résultat du tri : une liste des sommets dans leur ordre de tri comme l'indique la figure : Figure3.13.

**Imprimer le graphe** L'impression du graphe dessiné se fait par la création d'une image .png, qui par la suite peut être imprimée ou simplement sauvegardée dans notre ordinateur. La figure : Figure3.14 nous montre le graphe de l'exemple Make enregistré dans le répertoire "D :Make.PNG".

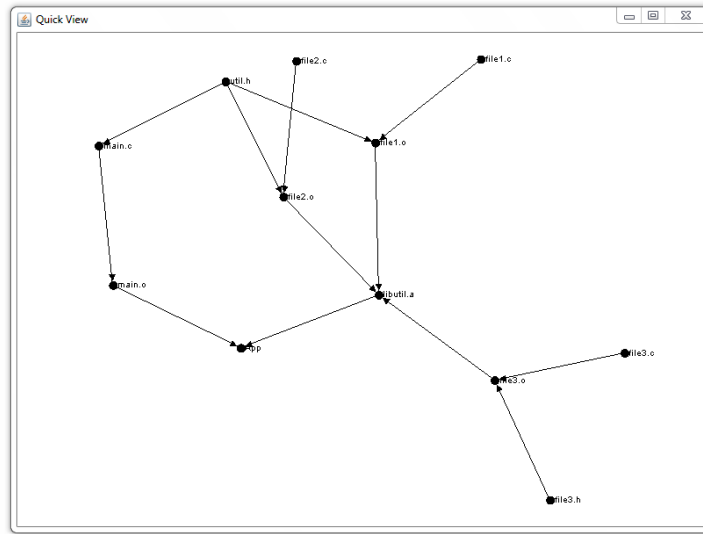


FIGURE 3.8 – Quick view Make

### 3.2.4 Conclusion

Ce logiciel a été créé pour aider à dessiner des graphes qui par la suite sont une entrée de l'algorithme du tri topologique ou même un simple teste si notre graphe est un dag ou non. Un outil puissant qui englobe la plupart des notions de la théorie du graphe et efficace pour dessiner et répondre à tous les types de graphes. L'interface graphique est plutôt réussie même si nous pourrions toujours améliorer certains points comme accentuer la flexibilité du panneau du MGraph. Un point pourrait tout de même paraître intéressant : c'est que si quelqu'un le désire, il peut modifier le code de l'application et le recompiler pour la compléter et ajouter d'autres algorithmes concernant le graphe : problème du plus court chemin, problème de coloration ... ou regarder le code simplement pour en apprendre plus sur la conception orientée objet et aider les étudiants (ou quelqu'un le désirant) à se familiariser avec les graphes et le tri des graphes. Vu l'étendu du sujet, ce logiciel pourra être, par la suite, complété par moi-même ou bien tout développeur le désirant.

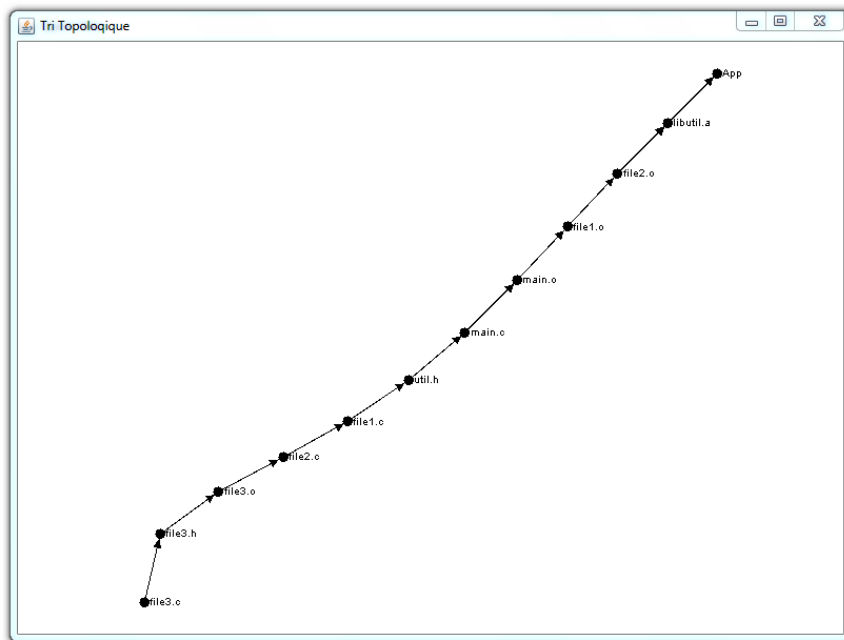


FIGURE 3.9 – Résultat du tri (1)

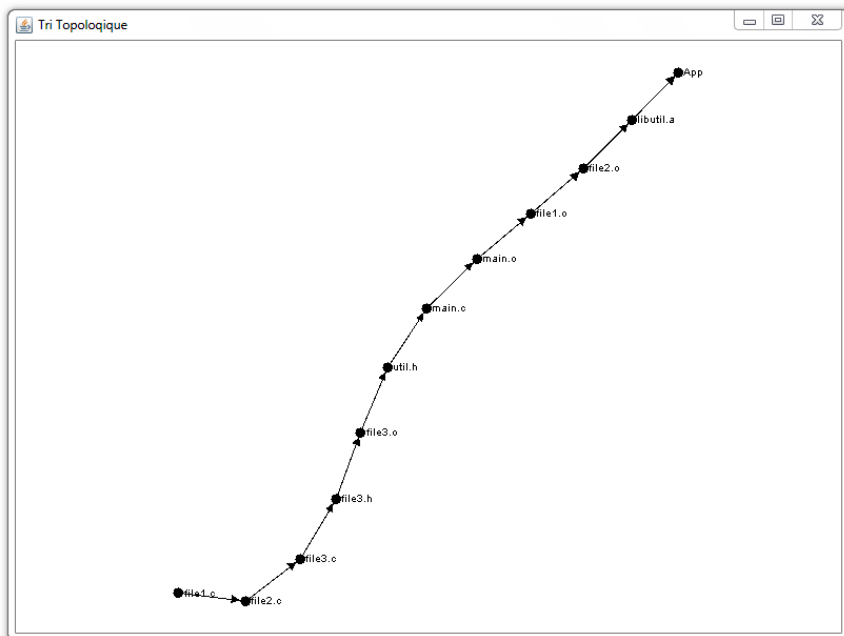


FIGURE 3.10 – Résultat du tri (2)

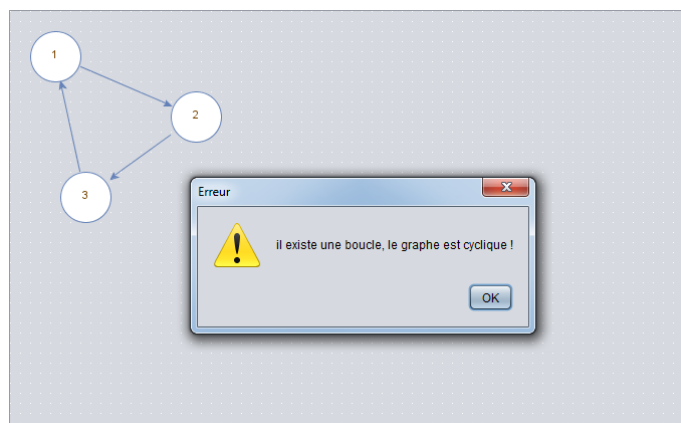


FIGURE 3.11 – Graphe cyclique

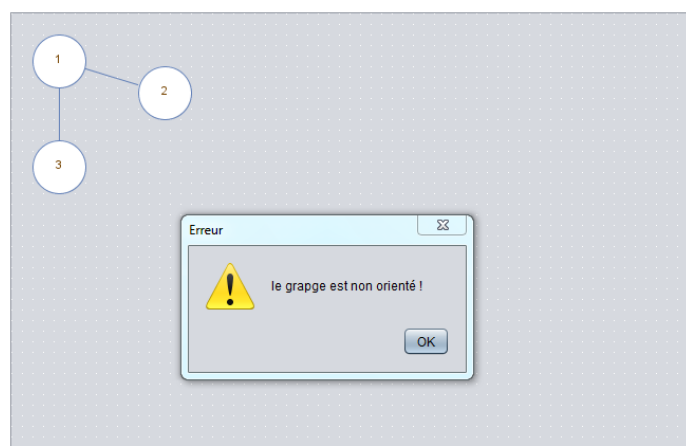


FIGURE 3.12 – Graphe non orienté

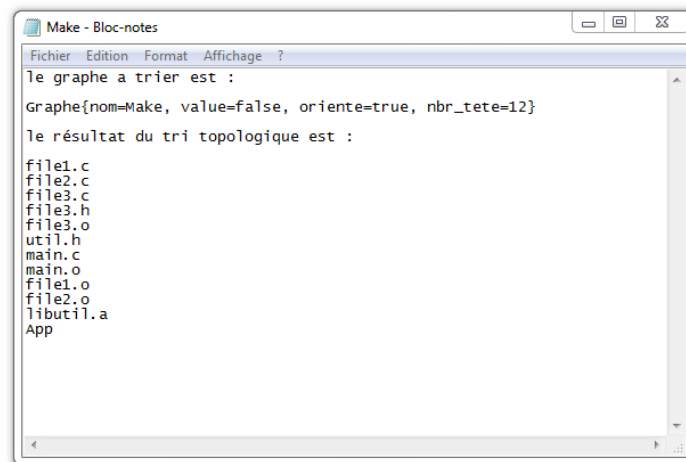


FIGURE 3.13 – Sauvgarde du tri topologique

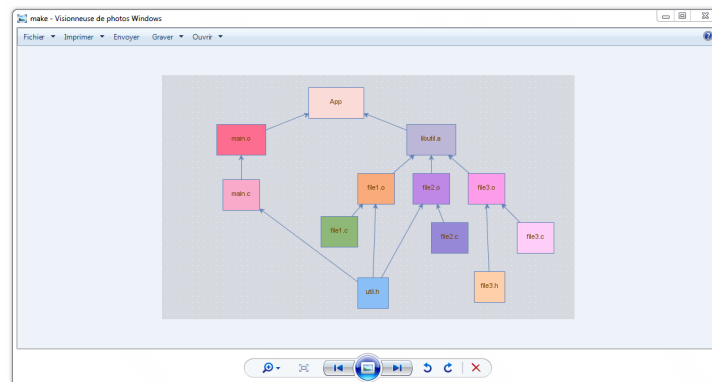


FIGURE 3.14 – Image du MGraph Make

# Conclusion

Au terme de ce rapport, nous tentons de cerner en quelques lignes les points les plus importants traités au cours de notre travail.

En premier temps, j'ai introduit les principales définitions et notations nécessaires à la compréhension des graphes et les différentes représentations pour le décrire, ensuite j'ai introduit le problème de tri topologique sa définition, ses applications et enfin l'algorithme pour tester que le graphe est acyclique et son algorithme.

En deuxième temps, J'ai présenté l'algorithme proposé pour implémenter le problème étudié en c, les structures des données choisies et sa traduction en C.

Ensuite j'ai implémenté l'algorithme en un langage orienté objet Java. J'ai décrit l'implémentation du problème en java, ensuite j'ai présenté une interface qui permet d'entrer des graphes généraliste, j'ai décrit ses différents opérations et actions et enfin j'ai introduit un exemple complet du problème Make.

Quant aux fruits de ce travail, nous pouvons citer les avantages suivants :

- Vérifier l'avantage de programmation orienté objet pour mieux représenter les objets réels et optimiser la complexité du l'algorithme de tri topologique.
- Implémenter une interface simple, flexible et puissante pour dessiner, visualiser et stocker des graphes générique.
- Le tri du graphe, nous permet autre que le trier typologiquement, vérifier s'il contient un cycle, ainsi une deuxième utilisation du même algorithme utilisée dans plusieurs domaines.

Quant aux perspectives de ce travail, nous pourrions envisager les prolongements suivants :

- Java, un langage permettant d'intégrer efficacement l'applica-

tion dans une page Web grâce aux applets.

- Ajouter d'autres algorithmes concernant le graphe : problème du plus court chemin, problème de coloration ...

Ce projet m'a permis d'utiliser en pratique les connaissances acquises tout au long de ces trois années de licence fondamentale en informatique à l'ISIMM. Sa réalisation m'a permis non seulement de compléter mes connaissances en théorie de graphe mais aussi dans le langage de programmation Java et c. Ainsi que d'approfondir mes connaissances dans les bonnes pratiques de programmation.



# Bibliographie

- [1] Ioan Todinca avec le concours de Julien Tesson. *Algorithmique des graphes et quelques notes de cours*. 29 avril 2008.
- [2] Claude Berge. *Théorie des graphes et ses applications*. 1958.
- [3] Yaël Champclaux. *Un modele de recherche d information basé sur les graphes et les similarités structurelles poue l amélioration du processus de recherche d information*. PhD thesis, l Université Toulouse III Paul Sabatier, 4 décembre 2009.
- [4] Pierre Lopez. *GRAPHES*. LAAS CNRS, 2 avril 2008.
- [5] Maciej MACOWICZ. *Approche générique des traitements de graphes*. PhD thesis, L INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON, 7 novembre 1997.
- [6] Didier Muller. *Introduction À la théorie des graphes*. CAHIERS DE LA CRM : Commission Romandede Mathématique, 2012.
- [7] Michel Rigo. *Théorie des graphes*. Université de Liege, 2010.
- [8] Algue Rythme. *DÉCOUVERTE DES ALGORITHMES DE GRAPHE*. geste de savoir, 05 mars 2016.