République Algérienne Démocratique et Populaire

Ministère de l 'Enseignement Supérieur et de la Recherche Scientifique

Faculté des Sciences et de l'Ingénierie

Département du génie informatique

PROJET DE FIN D'ETUDES

Pour l'obtention du diplôme

d'ingénieur d'état en informatique

Option : Systèmes Parallèles et Distribués

Thème

Réalisation d'une bibliothèque des algorithmes de manipulation des graphes

Réalisé par :

Damani Rebeiha

Encadré par :

Mlle.Belabbaci.A

N° d'ordre :……/2009-PFE/DGI

Résumé

Vu l'importance de la théorie des graphes dans la modélisation de nombreuses de situations complexes, beaucoup de bibliothèque regroupant les algorithmes de manipulation des graphes on été développées.
L'objectif de notre travail est de développer une bibliothèque statique comportant le maximum de fonctions de manipulation des graphes pour qu'elle puisse être utilisée dans divers applications.
Mots clés : graphe, bibliothèque statique, bibliothèque.

TABLES DES MATIÈRES

TABLES DES FIGURES

Introduction

La théorie des graphes débute avec les travaux d'Euler et trouve son origine dans l'étude de certains problèmes, comme celui des ponts de Königsberg[Lab.81], les habitants de Königsberg se demandaient s'il était possible, en partant d'un quartier quelconque de la ville, de traverser tous les ponts sans passer deux fois par le même pont et de revenir à leur point de départ.

Aujourd'hui, la théorie des graphes est un vaste domaine de recherche elle est développée et utilisée dans diverses disciplines tel que les réseaux de communication, les applications industrielles (problème du voyageur de commerce), l'étude de circuits électriques, la chimie(modélisation de structures), les sciences sociales (modélisation des relations), la biologie, l'économie, Elle constitue l'un des instruments les plus courants et les plus efficaces pour résoudre des problèmes discrets posés en Recherche Opérationnelle.

De manière générale, un graphe permet de représenter simplement la structure et les connexions, c'est une structure de données puissante pour l'informatique.

Plusieurs bibliothèques regroupent des fonctions de manipulation des graphes ont été réalisées par exemple Boost Graph Library(BGL), Ruby Graph Library(RGL), Template Graph Library(TGL), Adaptive Graph Library (AGL), Ocamlgraph, SeqAn, Goto, Graph Magics,...

Dans ce travail, on a développé notre bibliothèque « graph », elle contient le maximum des fonctions de manipulation des graphes.

Les fonctions ont été implémentées en langage c++ pour qu'elle puisse être utilisée par divers applications développés dans n'importe quel langage de programmation.

Enfin, cette bibliothèque est statique.

Elle fonctionne pour le moment uniquement sous la plate forme Windows.

On a développé également une application de démonstration pour tester le fonctionnement des différentes fonctions de la bibliothèque.

Ce mémoire est composé de trois chapitres :

Le premier chapitre contient les notions de base de la théorie des graphes.

Le deuxième chapitre contient les fonctions de manipulation des graphes implémentées dans la bibliothèque. Pour chaque fonction, on a donne sa description, son algorithme et un exemple d'application.

Le troisième chapitre contient la description de la bibliothèque, de la structure de données utilisée pour implémenter les graphes et l'application de démonstration.

On termine par une conclusion où on cite les perspectives de notre travail.

On a préparé également un manuel de références qui détaille les fonctions accessibles de la bibliothèque ainsi que leurs prototype.

Chapitre 1

Notions de base sur les graphes

Dans ce chapitre nous présentons quelques notions de base qui sont nécessaires dans la théorie des graphes.

Nous commençons par des définitions sur les graphes, les relations entre les sommets (successeur, prédécesseur), les opérations sur les graphes (graphe complémentaire, graphe inverse,…), Chaîne, chemin,…

1. Définition d'un graphe :

1.1. Graphe non orienté :

On appelle graphe G = (X, U) la donnée d'un ensemble X dont les éléments sont appelés sommets et d'une partie de A symétrique ((x, y) ∈ U, (y, x) ∈ U) dont les éléments sont appelés arêtes. [Did.03]

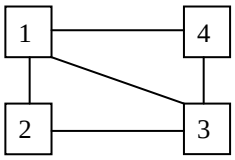


Fig.1.1 : Graphe non orienté.

1.2. Graphe orienté :

On appelle graphe orienté ou digraphe G = (X,U) la donnée d'un ensemble X dont les éléments sont appelés sommets et d'une partie de U de X × X dont les éléments sont appelés arcs ou arêtes. [Did.03]

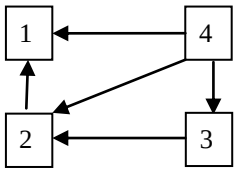


Fig.1.2 : Graphe orienté.

2. Relation entre les sommets :

2.1. Successeurs d'un sommet :

L'ensemble des successeurs d'un sommet x est désigné par $\Gamma^{+}(x)$ ou (Γ (x)) tel que :

Γ (x) = {y / y ∈ X; (x,y) ∈ U}. [S1]

Exemple : dans le graphe (1) de la Fig.1.2, Γ est définie ainsi :

$\Gamma^{+}(1) = \emptyset$; $\Gamma^{+}(2) = \{1\}$; $\Gamma^{+}(3) = \{2\}$; $\Gamma^{+}(4) = \{1,2,3\}$;

2.2. Prédécesseurs d'un sommet :

L'ensemble des prédécesseurs d'un sommet x est désigné par $\Gamma^{-}(x)$ ou ($\Gamma^{-1}(x)$) tel que :

$\Gamma^{-}(x) = \{y / y \in X; (y, x) \in U\}$. [S1]

Exemple : dans le graphe (1) de la Fig.1.2, Γ^{-} est définie ainsi :

$\Gamma^{-}(1) = \{2,4\}$; $\Gamma^{-}(2) = \{3,4\}$; $\Gamma^{-}(3) = \{4\}$; $\Gamma^{-}(4) = \emptyset$;

3. Degré d'un sommet :

3.1. Demi-degré extérieur :

Le demi-degré extérieur d'un nœud x est désigné par $d^+(x)$ tel que : $d^+(x) = |\Gamma(x)|$. [S1]

Exemple : dans le graphe (1) de la Fig.1.2, les demi-degrés extérieurs sont définis ainsi :

$d^+(1) = 0$; $d^+(2) = 1$; $d^+(3) = 1$; $d^+(4) = 3$;

3.2. Demi-degré intérieur :

Le demi-degré intérieur d'un nœud x est désigné par $d^-(x)$ tel que : $d^-(x) = |\Gamma^-(x)|$. [S1]

Exemple : dans le graphe (1) de la Fig.1.2, les demi-degrés intérieurs sont définis ainsi :

$d^-(1) = 2$; $d^-(2) = 2$; $d^-(3) = 1$; $d^-(4) = 0$;

3.3. Degré :

Le degré du nœud x est désigné par d (x) tel que : $d(x) = d^+(x) + d^-(x)$. [S1]

Exemple : dans le graphe (1) de la Fig.1.2, les demi-degrés intérieurs sont définis ainsi :

$d(1) = 2$; $d(2) = 3$; $d(3) = 2$; $d(4) = 3$;

4. Le nombre d'arcs :

Nombre d'arcs du graphe G est désigné par | U | tel que $|U| = \sum_{x \in X} d(x)/2$.

Exemple : dans le graphe (1) de la Fig.2, le nombre d'arcs est définie ainsi :

$|u| = (2 + 3 + 2 + 3) / 2 = 5$.

5. chaîne:

Une chaîne est une suite finie de sommets reliés entre eux par des arêtes. [Sig.02]

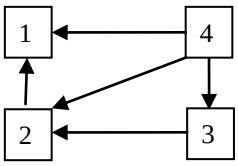


Fig.1.3 : Exemple de graphe pour la chaîne.

la suite (1,4,3,2,4,3) est une chaîne de 1 à 3 et (3,2,1,3) n'est pas une chaîne.

- Il existe différents types de chaîne :

5.1. Chaîne élémentaire

une chaîne est élémentaire si chaque sommet x apparaît au plus une fois. [Sig.02].

Exemple : dans le graphe de la Fig.1.3 la suite (1,4,3,2) est une chaîne élémentaire de 1 à 2 mais (1,4,3,2,4) n'est pas une chaîne élémentaire car le sommet 4 apparait deux fois.

5.2. Chaîne simple :

Une chaîne est simple si chaque arête apparaît au plus une fois. [Sig.02].

Exemple : dans le graphe de la Fig.1.3 la suite (1,4,3,2) est une chaîne simple de 1 à 2

mais (4,3,2,1,4,3,2) n'est pas une chaîne simple.

5.3. Chaîne hamiltonienne :

Une chaîne hamiltonienne est une chaîne élémentaire passant par tous les sommets. [Sig.02]

Exemple : dans le graphe de la Fig.1.3 la suite (1,4,3,2) est une chaîne hamiltonienne de 1 à 2 mais (3,4,1,2,4) n'est pas une chaîne hamiltonienne.

5.4. Chaîne eulérienne :

Une chaîne eulérienne est une chaîne simple passant par toutes les arêtes. [Sig.02]

Exemple : dans le graphe de la Fig.1.3 la suite (4,1,2,4,3,2) est une chaîne eulérienne de 4 à 2 mais (1,2,3,1,4,3,2) n'est pas une chaîne eulérienne.

6. Chemin :

Un chemin est une suite de sommets reliés par des arcs dans un graphe orienté. [Sig.02].

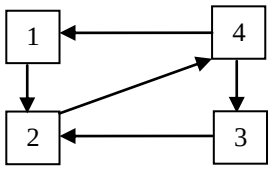


Fig.1.4 : Exemple de graphe pour le chemin.

Exemple : dans le graphe de la Fig.1.4 la suite (2,4,3,2) est un chemin de 2 à 1 mais (3,4,2,1) n'est pas chemin.

- Il existe des différents types de chemin :

6.1.Chemin élémentaire :

Une chemin est élémentaire si chaque sommet x apparaît au plus une fois. [Sig.02]

Exemple : dans le graphe de la Fig.1.4 la suite (4,3,2) est un chemin élémentaire de 4 à 2. (4,1,2,4,3) n'est pas chemin élémentaire .

6.2. Chemin simple :

Un chemin est simple si chaque arc apparaît au plus une fois. [Sig.02]

Exemple : dans le graphe de la Fig.1.4 la suite (1,2,4,3) est chemin simple mais (4,3,2,4,3) n'est pas chemin simple.

6.3. Chemin hamiltonien :

Un chemin hamiltonien est un chemin élémentaire passant par tous les sommets. [Sig.02]

Exemple : dans le graphe de la Fig.1.4 la suite (3,2,4,1) est un chemin hamiltonien mais (4,1,2,4,3) n'est pas chemin hamiltonien.

6.4. Chemin eulérien :

Un chemin eulérien est un chemin simple passant par tous les arcs. [Sig.02].

Exemple : dans le graphe de la Fig.1.4 la suite (4,1,2,4,3,2) est un chemin eulérien mais (2,4,3,2,4,1) n'est pas chemin eulérien.

7. Cycle :

Un cycle est une chaîne qui revient à son point de départ. [Sig.02].

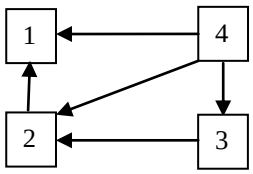


Fig.1.5 : Exemple de graphe pour le cycle.

La suite (1,4,3,2,1) est un cycle mais (1,4,3,2) n'est pas un cycle.

- Il existe différents types de cycle :

7.1. Cycle élémentaire :

Un cycle élémentaire est une chaîne élémentaire qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.5 la suite (1,4,3,2,1) est un cycle élémentaire mais (3,2,1,4,2,3) n'est pas un cycle élémentaire .

7.2. Cycle simple :

Un cycle simple est une chaîne simple qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.5 la suite (1,4,3,2,1) est un cycle simple mais (3,2,1,4,2,3) n'est pas un cycle simple.

7.3. Cycle hamiltonien :

Un cycle hamiltonien est une chaîne hamiltonien qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.5 la suite (1,4,3,2,1) est un cycle hamiltonien mais (3,2,1,4,2,3) n'est pas un cycle hamiltonien.

7.4. Cycle eulérien:

Un cycle eulérien est une chaîne eulérienne qui revient à son point de départ.

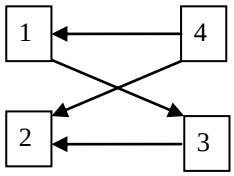


Fig.1.6 : Exemple de graphe pour cycle

La suite (1,4,2,3,1) est un cycle eulérien mais (2,3,1,3,2) n'est pas un cycle eulérien.

8. Circuit :

Un circuit est un chemin qui revient à son point de départ. [Sig.02]

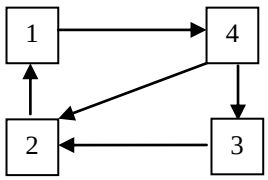


Fig.1.7 : Exemple de graphe pour circuit

La suite (4,2,1,4) est un circuit mais (4,2,3,4) n'est pas un circuit.

- Il existe différents types de circuit :

8.1. Circuit élémentaire :

Un circuit élémentaire est un chemin élémentaire qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.7 la suite (1,4,2,1) est un circuit élémentaire mais (2,1,4,2,1,4,2) n'est pas un circuit élémentaire .

8.2. Circuit simple :

Un circuit simple est un chemin simple qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.7 la suite (4,3,2,1,4) est un circuit simple mais (1,4,2,1,4) n'est pas un circuit simple.

8.3. Circuit hamiltonien :

Un circuit hamiltonien est un chemin hamiltonien qui revient à son point de départ. [Sig.02]

Exemple : dans le graphe de la Fig.1.7 la suite (4,3,2,1,4) est un circuit hamiltonien mais (1,2,4,1) n'est pas un circuit hamiltonien.

8.4. Circuit eulérien :

Un circuit eulérien est un chemin eulérien qui revient à son point de départ. [Sig.02]

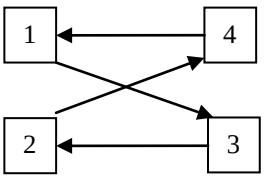
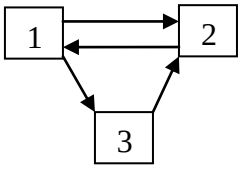


Fig.1.8 : Exemple de graphe pour circuit eulérien

La suite (4,1,3,2,4) est un circuit eulérien.(3,2,1,3) n'est pas un circuit eulérien.

9. Distance :

la distance entre x et y est la longueur du plus court chemin entre ces sommets. [S2]



D =

	1	2	3
1	0	1	1
2	1	0	2
3	2	1	0

Fig.1.9 : Exemple de la distance.

10. Parcours d'un graphe :

Parcourir un graphe consiste à choisir un sommet et à énumérer à partir de celui-ci ses sommets en suivant ses arcs autant que possible. Le choix de l'arc et du nœud à visiter en priorité est défini au préalable par le type du parcours en profondeur, ou en largeur.

10.1. Parcours en profondeur (DFS : Depth First Search) :

Pour parcourir un graphe en profondeur on suit les étapes suivantes :

- ✓ Initialement tous les sommets sont marqués " non visités".
- ✓ Choisir un sommet s de départ et le marquer " visité".
- ✓ Chaque sommet adjacent à s, non visité, est à son tour visité en utilisant DFS récursivement.
- ✓ Une fois tous les sommets accessibles à partir de s ont été visités, le parcours en profondeur est terminé [S3].

Remarque : il n'y a pas un parcours en profondeur unique.

Exemple : voyons le fonctionnement du parcours en profondeur sur le graphe suivant :

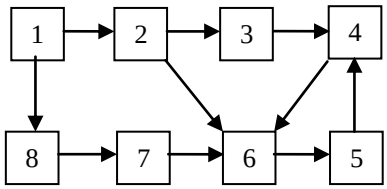


Fig.1.10 : Graphe pondéré.

- ✓ On Choisit le sommet 1.
- ✓ On explore dans l'ordre les sommets 1 , 2 , 3 , 4 , 6 , 5 , 8 ,7.

10.2. Parcours en largeur (BFS : Breadth First Search):

pour parcours un graphe en largeur on suit les étapes suivantes :

- ✓ Initialement tous les sommets sont marqués " non visités".
- ✓ Choisir un sommet s de départ et le marquer " visité".
- ✓ Visiter tous les voisins d'un sommet avant de visiter le sommet suivant qui sera le premier voisin à avoir été visité auparavant. Pour conserver les voisins d'un sommet en cours de visite que l'on visitera ultérieurement, on utilisera une file. [S3]

Exemple : on applique la méthode du parcours en largeur sur le graphe de la Fig.1.10 :

- ✓ On Choisit le sommet 1;
- ✓ On explore dans l'ordre les sommets 1 , 2 , 8 , 3 , 6 , 7 , 4 , 5.

Remarque : il n'y a pas un parcours en largeur unique.

11. Connexité :

11.1. Graphe connexe :

Un graphe est connexe s'il existe une chaîne entre tout couple de sommets. [S4]

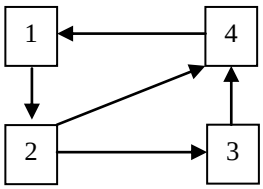


Fig.1.11: Graphe connexe

11.2. Graphe fortement connexe :

Un graphe est fortement connexe s'il existe un chemin entre tout couple de sommets. [S4]

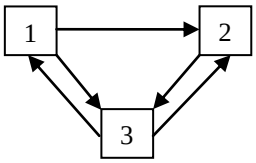


Fig.1.12 : Graphe fortement connexe.

11.3. Composantes fortements connexes :

On appelle composante fortement connexe (cfc) un sous graphe fortement connexe maximal, c'est à dire un sous graphe fortement connexe qui n'est pas inclus dans un autre sous graphe fortement connexe. [S4]

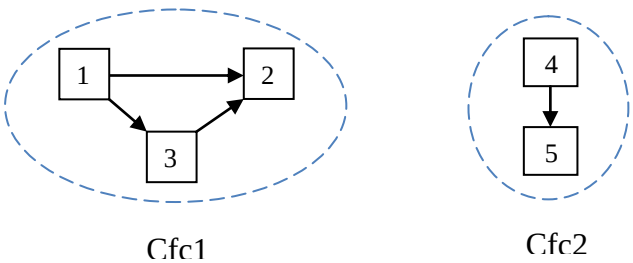


Fig.1.13 : Graphe non fortement connexe

12. Graphes particulier :

12.1. Graphe r- régulier :

On dit qu'un graphe est r-régulier si : ∀ x ∈ X , d(x) = r. [S1]

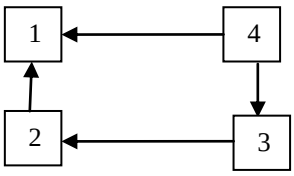


Fig.1.14 : Graphe 2-régulier.

12.2. Graphe complet:

On dit qu'un graphe est complet si : ∀ x,y ∈ U, (x,y) ∉U ⇒ (y,x) ∈ U. [S1]

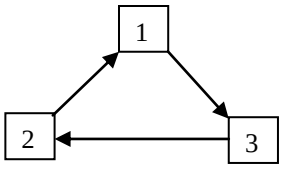


Fig.1.15 : Graphe complet

12.3. Graphe réflexif :

On dit qu'un graphe est réflexif si : ∀ x ∈ X, (x , x) ∈ U. [S1]

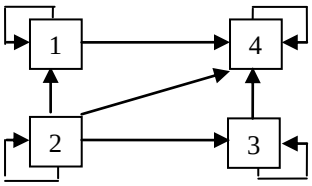


Fig.1.16 : Graphe réflexif.

12.4. Graphe irréflexif :

On dit que un graphe est réflexif si : ∀ x ∈ X, (x , x) ∉ U. [S1]

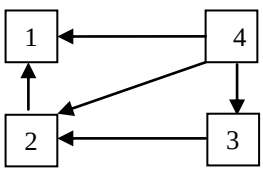


Fig.1.17 : Graphe irréflexif.

12.5. Graphe symétrique :

On dit que un graphe est symétrique si : $\forall x,y \in X, (x,y) \in U \Rightarrow (y,x) \in U$. [S1]

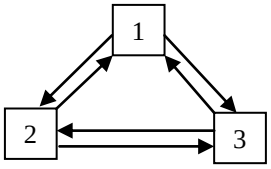


Fig.1.18 : Graphe symétrique.

12.6. Graphe asymétrique :

On dit que un graphe est asymétrique si : $\forall x,y \in X, (x,y) \in U \Rightarrow (y,x) \notin U$. [S1]

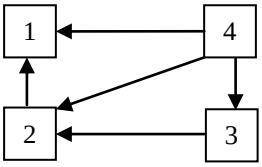


Fig.1.19 : Graphe asymétrique.

12.7. Graphe eulérien :

Un graphe est eulérien si et seulement s'il est connexe et $\forall x \in X$: d(x) est un entier pair. [Sig.02]

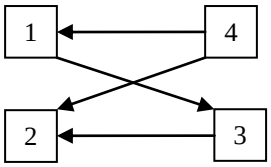


Fig.1.20 : Graphe eulérien.

12.8. Graphe semi eulérien :

Un graphe est semi eulérien si et seulement s'il est connexe et il admet zéro sommets impairs ou deux sommets exactement impairs. [Sig.02]

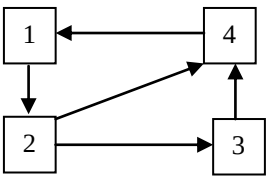


Fig.1.21 : Graphe semi eulérien.

12.9. Graphe hamiltonien :

Un graphe est hamiltonien si et seulement si : ∀ x ∈ X : d(x) >= n. (n : le nombre de sommet) [Sig.02]

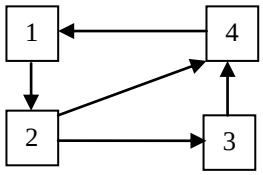


Fig.1.22 : Graphe hamiltonien.

12.10. Arbre :

Un arbre est un graphe connexe tel que le nombre d'arcs est égale à n-1. [Mar.03]

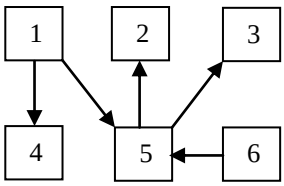


Fig.1.23: Arbre.

12.11. Arborescence :

Un Arborescence est un graphe connexe tel que ∃ s ∈ X, $d^-(s) = 0$ et ∀x ≠ s, $d^-(x) = 1$. [Mar.03]

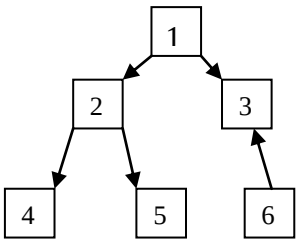


Fig.1.24: Arborescence.

13.Réseau de transport :

Un réseau de transport est un graphe orienté sans boucle qui possède deux sommets particuliers : une source s et un puits p.les arcs portent des capacités (poids positifs). [Mar.03]

14. Flot :

Un flot f dans un réseau de transport est une fonction qui associe à chaque arc u une quantité f(u) qui représente la quantité de flot qui passe par cet arc, en provenance de la source s et en destination du puits p. [Mar.03]

Un flot doit respecter les trois règles suivantes :

1. $\forall\ (i,j) \in U,\ 0 \leq f(i,j) \leq \gamma(i,j)$,le flot sur chaque arc est compatible avec la capacité γ.
2. $\forall\ i \neq s,p,\ \sum_{(i,j) \in U} f(i,j) = \sum_{(j,i) \in U} f(j,i)$, le flot est conservé à chaque nœud.
3. $F = \sum_{(s,i) \in U} f(s,i) = \sum_{(i,p) \in U} f(i,p)$, le flot complet est égal au flot entrant par s et sortant par p.

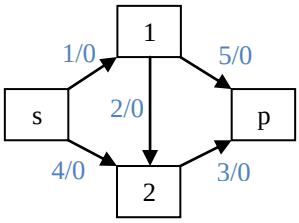


Fig.1.25 : Réseau transport.

Conclusion

Après la présentation des notions fondamentales dans la théorie des graphes. On va étudier dans le chapitre suivant les fonctions de manipulation des graphes : les fonctions classiques (tester l'existence d'une chaîne, parcours en profondeur, parcours en largeur,…), les fonctions de recherche de plus court chemin, fonctions de recherche d'un arbre recouvrant minimum,…

Chapitre 2

Fonctions de manipulation des graphes

Dans ce chapitre on présente les fonctions de manipulation des graphes on a différencié :
les fonctions classiques tel que le parcours d'un graph, la tester l'existence d'un chemin, ajouter d'un sommet,…
et les fonctions de recherche du plus court chemin, les fonctions de recherche d'un arbre recouvrant minimum,...
Pour chaque fonction on a donné son principe de fonctionnement, son algorithme et un exemple explicatif.

1. Fonctions élémentaires :

Soit le graphe G=(X,U) ; X = {1,2,…….,n} , U = {u_1,u_2,…..,u_m};

1.1. Chaîne:

La fonction ***chn*** ci- dessous permet de tester si une suite de sommets est une chaîne.

Fonction chn(Graphe G = (X,U) , c: suite de sommets) : booléen

Variable x : sommet;

1. Posons chn = vrai;
2. x := premier(c);
3. Si (x,suivant(x)) ∉ U ou (suivant(x),x) ∉ U alors chn := faux et aller à 6;
4. x := suivant(x) ;
5. Si suivant(x) ≠ nul alors retourner à 3;
6. Fin.

➢ Les types de Chaîne:

Chaîne élémentaire :

La fonction ***chnel*** ci- dessous permet de tester si les sommets sont distincts dans la suite (c),

Fonction chnel(Graphe G = (X,U), c : suite de sommes) : booléen

Variable x: sommet;

1. Posons S = Ø; chnel = vrai;
2. x := premier(c);
3. S := S ∪ {x}; x := suivant(x);
4. Si x ∈ S alors chnel := faux et aller à 7;
5. Si S ≠ c alors retourner à 3;
6. Si chnel = vrai alors chnel:= chn(G,c);
7. Fin.

Chaîne simple :

La fonction ***chns*** ci- dessous permet de tester si les arêtes sont distinctes dans la suite (c).

Fonction chns(Graphe G = (X,U) , c : suite de sommets) : booléen

Variable x,y,z : sommet;

1. Posons chns = vrai;
2. x := premier(c);
3. Si suivant(x) ≠ nul alors y := suivant(x) sinon aller à 8;
4. Si suivant(y) ≠ nul alors z:= suivant(y) sinon aller à 8;
5. Si z ≠ x alors z := suivant(z) sinon chns := faux et aller à 9;

6. Si précédent(z) = y ou suivant(z) = y alors chns = faux et aller à 9;
7. x := y et retourner à 3;Si chns = vrai alors chns:= chn(G,c);
8. Si chns = vrai alors chns:= chn(G,c);
9. Fin.

Chaîne hamiltonienne :

La Procédure ***chain_hamil*** ci- dessous, teste si une suite de sommets est une chaîne hamiltonienne.

Procédure chain_hamil (Graphe G = (X,U))

Variable c : suite de sommets, n : nombre de sommet du graphe;

1. Si taille(c) ≠ n alors la suite c n'est pas une chaîne hamiltonienne et aller à 3.
2. Si chnel(G,c) = vrai alors la suite c est une chaîne hamiltonienne ;
3. Fin.

Chaîne eulérienne :

La Procédure ***chain_euler*** ci- dessous, teste si une suite de sommets est une chaîne eulérienne.

Procédure chain_simp (Graphe G = (X,U))

Variable c : suite de sommets, n : nombre de sommet du graphe;

1. Si taille(c) - 1 ≠ nbr_arc(G) alors la suite c n'est pas chaîne eulérienne et aller à 3;
2. Si chns(G,c) = vrai alors la suite c est une chaîne eulérienne, sinon elle n'est pas une chaîne eulérienne;
3. Fin.

1.2. Chemin :

La fonction ***chm*** a le même principe que la ***chn*** mais dans ce cas le teste doit être dans un seul sens pour cela en remplace la ligne 2 de la fonction chn par :

2. Si (x,suivant(x)) ∉ U alors chm := faux et aller à 6;

➢ Les types de chemin :

Chemin élémentaire :

Pour tester si une suite de sommets est un chemin élémentaire, on utilise une autre fonction appelée ***chmel*** qui a le même principe que la ***chnel*** mais cette fonction appelle la fonction ***chm*** au lieu de ***chn***.

Chemin simple :

Pour tester si une suite de sommets est un chemin simple, on utilise une autre fonction appelée ***chms*** qui a le même principe que la ***chns***, cette fonction appelle la fonction ***chm*** au lieu de ***chn***.

Chemin hamiltonien :

Pour tester si une suite de sommets est un chemin hamiltonien, la procédure ***chemin- hamil*** a le même principe que ***chain_hamil***, cette procédure utilise la fonction ***chmel*** au lieu de ***chnel***.

Chemin eulérien :

Pour tester si une suite de sommets est un chemin eulérien, la procédure ***chemin_euler*** a le même principe que ***chain_ euler***, cette procédure utilise la fonction ***chms*** au lieu de ***chns***.

2.1. Cycle :

La Procédure ***cycl*** ci- dessous, teste si une suite de sommets est un cycle.

Procédure cycle(Graphe G = (X,U));

Variable c : suite de sommets;

1. Si premier(c) ≠ dernier(c) alors la suite c n'est pas un cycle et aller à 3;
2. Si chn(G,c) = vrai alors la suite c est un cycle sinon la suite n'est pas un cycle;
3. Fin.

➢ Les types de cycle :

Cycle élémentaire :

La Procédure ***cycl_elem*** ci- dessous, teste si une suite de sommets est un cycle élémentaire.

Procédure cycl_elem(Graphe G = (X,U));

Variable c : suite de sommets;

1. Si premier(c) ≠ dernier(c) alors la suite c n'est pas un cycle élémentaire et aller à 5 ;
2. Si (précédent(dernier(c)),dernier(c)) ∉ U ou (dernier(c),précédent(dernier(c))∉ U alors la suite c n'est pas un cycle élémentaire et aller à 5;
3. c := c –{dernier(c)};
4. Si chnel(G,c) = vrai alors la suite c est un cycle élémentaire sinon la suite n'est pas un cycle élémentaire;
5. Fin.

Cycle simple :

La Procédure ***cycl_simp*** ci- dessous, teste si une suite de sommets est un cycle simple.

Procédure cycl_simp(Graphe G = (X,U));

Variable c : suite de sommets;

1. Si premier(c) ≠ dernier(c) alors la suite c n'est pas un cycle simple et aller à 3 ;
2. Si chns(G,c) = vrai alors la suite c est un cycle simple sinon la suite n'est pas un cycle simple;
3. Fin.

Cycle hamiltonien :

La Procédure ***cycl_hamil*** ci- dessous, teste si une suite de sommets est un cycle hamiltonien.

Procédure cycl_hamil(Graphe G = (X,U));

Variable c : suite de sommets;

1. Si taille(c) ≠ n+1 alors la suite c n'est pas un cycle hamiltonien et aller à 5;
2. Si (précédent(dernier(c)),dernier(c)) ∉ U ou (dernier(c),précédent(dernier(c))) ∉ U alors la suite c n'est pas un cycle hamiltonien et aller à 5;
3. c := c –{dernier(c)} ;
4. Si chnel (G,c) = vrai alors la suite c est un cycle hamiltonien sinon n'est pas un cycle hamiltonien;
5. Fin.

Cycle eulérien:

La Procédure ***cycl_euler*** ci- dessous, teste si une suite de sommets est un cycle eulérien.

Procédure cycl_euler(Graphe G = (X,U));

Variable c : suite de sommets;

1. Si taille(c) ≠ nbr_arc (G) alors la suite c n'est pas un cycle eulérien et aller à 4;
2. Si premier(c) ≠ dernier(c) alors la suite c n'est un pas cycle eulérien et aller à 4 ;
3. Si chns(G,c)=vrai alors la suite c est un cycle eulérien sinon elle n'est pas un cycle eulérien;
4. Fin.

2.2. Circuit :

Pour tester si une suite de sommets est un circuit, la procédure ***circuit*** a le même principe que la procédure cycl, mais elle utilise la fonction ***chm*** au lieu de ***chn***.

- Les types de circuits :

Circuit élémentaire :

Pour tester si une suite de sommets est un circuit élémentaire, la procédure ***circuit_elem*** a le même principe que ***cycl_elem*** mais elle utilise la fonction ***chmel*** au lieu de ***chnel***.

Circuit simple :

Pour tester si la suite de sommets est un circuit simple, la procédure ***circuit_simp*** a le même principe que ***cycl_simp*** mais elle utilise la fonction ***chms*** au lieu de ***chns***.

Circuit hamiltonien :

Pour tester si une suite de sommets est un circuit hamiltonien, la procédure ***circuit_hamil*** a le même principe que ***cycl_hamil*** mais elle utilise la fonction ***chmel*** au lieu de ***chnel***.

Circuit eulérien :

Pour tester si une suite de sommets est un circuit eulérien, la procédure ***circuit_euler*** a le même principe que ***cycl_euler*** mais elle utilise la fonction ***chms*** au lieu de ***chn***s.

2.3. Distance :

La Fonction ***distance*** ci- dessous, permet de trouver la matrice des distances.

Fonction distance (Graphe G = (X,U)) : tableau d'entiers

Variable Dis[1..n],M[1..n] : tableau d'entiers; i,j,k,n : entier;

1. Posons i = 1; j = 1; k =2;
2. Si i = j alors M[i,j] := 0 et Dis[i,j] := 0 et aller à 4;
3. Si (i,j) ∈ U alors M[i,j] := 1 Dis[i,j] := 1; sinon M[i,j] := ∞; Dis[i,j] := ∞;
4. j := j+1 Si j <= n alors retourner à 2;
5. i := i+1; j := 1; Si i <= n alors retourner à 2;
6. $M^k[i,j] := \max(M^{k-1}[i,j] * M[i,j])$;
7. Si $M^k[i,j] \neq \infty$ et $Dis[i,j] \neq \infty$ alors Dis[i,j] := k;
8. j := j+1 Si j <= n alors retourner à 7;
9. i := i+1; j:=1 Si i <= n alors retourner à 7;
10. k := k+1;
11. si k<=n alors retourner à 6;
12. Fin.

2.4. Parcours d'un graphe :

2.4.1. Parcours en profondeur (DFS) :

La Procédure ***DFS*** ci- dessous, est utilisée par la procédure ***parcour_profond***.

DFS(Graphe G = (X,U) , sommet x)

Variable y : sommet; S : ensemble de sommets;

1. Marquer x;
2. Afficher x ;
3. Posons S = Ø;
4. Pour y ∉ S S := S ∪ {y};
5. Si (x,y) ∉ U et y non marqué alors DFS(G,y);
6. Si S ≠ X alors retourner à 4;
7. Fin.

La Procédure ***parcour_profond*** ci- dessous, permet de parcourir un graphe en profondeur;

Procédure parcour_profond(Graphe G = (X,U) , x : sommet);

1. Initialiser tous les sommets à non marqué ;

2. DFS(G,x);
3. Fin.

2.4.2. Parcours en largeur (BFS):

La Procédure ***parcour_largeur*** ci- dessous, permet de parcourir un graphe en largeur.

Procédure parcour_largeur(Graphe G = (X,U) , x : sommet);

Variable Mark : tableau de booléens; i,n : entier; s,y,z,w: sommet,

S : ensemble de sommets, F : File;

1. Initialiser tous les sommets à non marqué ;
2. Marquer x et Enfiler(F,x);
3. y := premier(F);
4. posons S = Ø;
5. Pour z ∉ S S := S ∪ {z};
6. Si (y,z) ∈ U et z non marqué alors marquer y et enfiler(F,z)
7. Si S ≠ X alors retourner à 5;
8. défiler (F);
9. Si F ≠ Ø alors retourner à 3;
10. Fin.

1.7. Connexité :

1.7.1. Graphe connexe :

La Procédure ***connex*** ci- dessous, teste si un graphe est connexe ou non.

La Procédure connex utilise la procédure ***parcour_largeur1*** qui a le même principe de ***parcour_largeur*** on remplace la ligne 7 par : Si (y,z) ∉ U ou (z,y) ∉ U alors aller à 9;

Procédure connex(Graphe G = (X,U))

Variable x,x1 : sommet ;

1. choisir x1 ;
2. parcour_largeur1(G,x1);
3. Si ∃ x ∈ X non marqué alors le graphe G n'est pas connexe sinon G est connexe;
4. Fin.

1.7.2. Graphe fortement connexe :

La Procédure ***Fconn*** ci- dessous, teste si un graphe G est connexe ou non.

Procédure Fconn(Graphe G = (X,U))

Variable x,y,x1 : sommet ;

1. choisir x1 ;
2. parcour_largeur (G,x1);

3. Si ∃ x ∈ X non marqué alors le graphe G n'est pas fortement connexe sinon G est fortement connexe;
4. Fin.

1.7.2. Composantes fortement connexes :

La fonction ***compos_fconnex*** ci- dessous, recherche la composante fortement connexe d'un graphe contenant un sommet a. L'idée de cette fonction est de parcourir le graphe à partir du point a dans le sens direct en suivant les flèches des arcs et de créer un ensemble des sommets parcourus. La même chose est effectuée dans le sens indirect en suivant les flèches des arcs en sens inverse et de créer un deuxième ensemble des sommets parcourus. L'intersection de ces deux ensembles donne les sommets qui à la fois peuvent atteindre a et sont accessibles à partir de a. Cette intersection est donc la composante fortement connexe qui contient a.

Fonction compos_fort_connex(Graphe G = (X,U), a : sommet) : ensemble de sommets;

Variable x,y : sommet ; examin :tableau de booléens; S,S1,S2 : ensemble de sommets;

1. Posons S = Ø; S1 = Ø;
2. S1:=S1 ∪ {a};
3. Pour x ∉ S S:= S ∪ {x}; examin[x] := faux;
4. Si S ≠ X alors retourner à 3;
5. Si ∃ x ∈ S1 / examin[x] = faux alors examin[x] := vrai; sinon aller à 10;
6. Posons S2 = Ø;
7. Pour y ∉ S2 S2 := S2∪ {y};
8. Si (x,y) ∈ U et y ∉ S1 alors S1:= S1∪ {y};
9. Si S2 ≠ X alors retourner à 7 sinon retourner à 5;
10. Posons S = Ø; S3 = Ø;
11. S3:=S3 ∪ {a};
12. Pour x ∉ S S:=S ∪ {x}; examin[x] := faux;
13. Si S ≠ X alors retourner à 12;
14. Si ∃ x ∈ S3 / examin[x] = faux alors examin[x] := vrai; sinon aller à 19;
15. Posons S2 = Ø;
16. Pour y ∉ S2 S2 := S2∪ {y};
17. Si (y,x) ∉ U et y ∉ S3 S1 := S1∪ {y};
18. Si S2 ≠ X alors retourner en 16 sinon retourner à 14;
19. compos_fort_connex := S1 ∩ S2 ;
20. Fin.

La Procédure ***compos_fort_connex*** ci- dessous, trouve toutes les composantes fortement connexes.

Procédure compos_fort_connex(Graphe G = (X,U))

Variable x : sommet; S,C : ensemble de sommet; i :entier;

1. Posons S = Ø ; i=1;
2. Pour x ∉ S S:=S ∪ {x};
3. Ci := compos_fort_connex(G,x); // Ci est la composante fortement connexe i;
4. S := S ∪ Ci; i:= i+1;
5. Si S ≠ X alors retourner à 2;
6. Fin.

2. Autres fonctions de manipulation des graphes :

2.1. Tri topologique :

Un tri topologique d'un graphe orienté acyclique G = (X,U) est un ordre linéaire des sommets de G tel que : si (x,y) ∈ U → x est visité avant y. [S5]

Principe du fonctionnement de l'algorithme :

Chercher une racine, puis enlever, et répéter l'opération autant de fois que nécessaire.

Remarque : il n'y a pas un tri topologique unique.

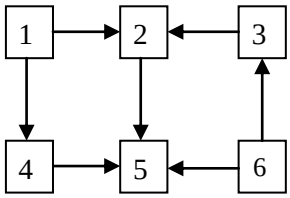


Fig.2.1: Graphe orienté

Un ordre topologique sur ce graphe peut donner par exemple la succession des sommets 1,4,6,3,2,5. En effet, chaque sommet apparait bien avant ses successeurs. Il n'y a pas unicité de l'ordre.

L'algorithme :

Variable Graphe G = (X,U); F : File;

1. Posons S = Ø;
2. Pour x ∉ S S:=S ∪ {x};
3. d (x) := dem_deg_int(G,x);
4. Si d(x) = 0 alors enfiler(F,x);
5. Si S ≠ X alors retourner à 2;

6. x := premier(F);
7. Defiler (F);
8. Afficher(x);
9. Posons S1 = Ø;
10. Pour y ∉ S1 S1 := S1 ∪ {y};
11. Si (x,y) ∈ U alors d(y) := d(y) -1;
12. Si d(y) = 0 alors enfiler(F,y);
13. Si S1 ≠ X alors retourner à 10;
14. Si F ≠ Ø alors retourner à 6;
15. Fin.

2.2. Algorithmes de recherche du plus court chemin :

Les problèmes de cheminement dans les graphes (en particulier la recherche d'un plus court chemin) comptent parmi les problèmes les plus anciens de la théorie des graphes et les plus importants par leurs applications.

Soit G(X,U,γ) un graphe orienté valué, γ (u) est le coût de l'arc u , X = (1….n) ; U = (u_1,..,u_m).

La matrice des coûts définie par : C = $(c_{i,j})_{i,j=1\ldots n}$

$$c_{i,j} = \begin{cases} 0 & \text{si } (i = j). \\ \infty & \text{si } (i , j) \notin U. \\ \gamma\ (i,j) & \text{si } (i , j) \in U. \end{cases}$$

2.2.1. Algorithme de Maria Hass (1961) :

Principe de l'algorithme : [Lab.81]

Cet algorithme ne permet pas de considérer les arcs négatifs.

- on utilise la matrice des coûts C.
- on calcule des puissances successives de C à laide des opérations suivantes :
  - ✓ ⊕ : a ⊕ b = min {a,b} et ⊗ : a ⊗ b = a + b.
  - ✓ A partir des opérations ⊕ et ⊗ $c^k[i,j] = \min\{c^{k-1}[i,h] + c[h,j]\}$.
  - ✓ Si $C^k = C^{k+1}$ alors C^k est la matrice des coûts minimums.

L'algorithme :

Variable : Graphe G = (X,U); i,j,k,l : entier; C[1..n] : tableau d'entiers;

1. Posons k = 2.
2. Posons i = 1; ;j = 1; l = 1;
3. $C^k[i,j] := \min \{C^{k-1}[i,l] + C[l,j])\}$;
4. l := l+1 si l<= n alors retourner en 3 sinon l := 1;
5. j := j+1 si j<=n alors retourner en 3 sinon j := 1

6. i := i+1 si l<=n alors retourner en 3.
7. Si $C^k[i,j] \neq C^{k-1}[i,l]$ alors k:=k+1 et retourner en 2.
8. Fin.

Exemple : soit le graphe suivant :

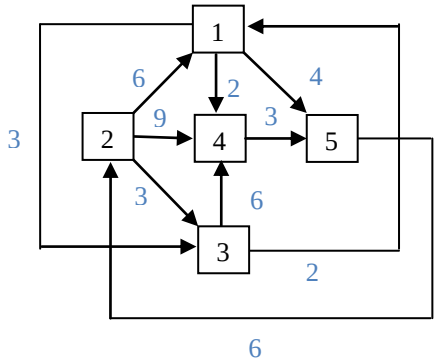


Fig 2.2 : Graphe orienté

L'application de l'algorithme sur ce graphe donne les résultats suivants :

C^1 =

	1	2	3	4	5
1	0	∞	3	2	4
2	6	0	3	9	∞
3	2	∞	0	6	∞
4	∞	∞	∞	0	3
5	∞	6	∞	∞	0

C^2 =

	1	2	3	4	5
1	0	10	3	2	4
2	5	0	3	8	10
3	2	∞	0	4	6
4	∞	9	∞	0	3
5	12	6	9	15	0

C^3 =

	1	2	3	4	5
1	0	10	3	2	4
2	5	0	3	7	9
3	2	12	0	4	6
4	15	9	12	0	3
5	11	6	9	14	0

C^4 =

	1	2	3	4	5
1	0	10	3	2	4
2	5	0	3	7	9
3	3	2	12	0	4
4	14	9	12	0	3
5	11	6	9	13	0

C5=

	1	2	3	4	5
1	0	10	3	2	4
2	5	0	3	7	9
3	3	2	12	0	4
4	14	9	12	0	3
5	11	6	9	13	0

C^5 = C^4.

On a $C^4[5,4]$ = 13 est le cout minimum pour aller de 5 à 4.

Pour obtenir le plus court chemin entre deux sommets on va déterminer les prédécesseurs successifs du deuxième sommet jusqu'à obtention k-2 prédécesseur (chemin de longueur k).

Exemple :

le prédécesseur de sommet 4 est le sommet qui prend le min{ $C^3[5,h] + C[h,4]$ }:

$$C^3[5,h] + C[h,4] = (11,6,9,14,0) + \begin{pmatrix}2\\9\\6\\0\\\infty\end{pmatrix} = (13,15,15,14,\infty)$$

le sommet 1est le prédécesseur de 4

le prédécesseur de sommet 1est le sommet qui prend le min{ $C^2[5,h] + C[h,1]$ } tel que l <= 5 :

$$C^2[5,h] + C[h,1] = (12,6,9,15,0) + \begin{pmatrix}0\\6\\2\\\infty\\\infty\end{pmatrix} = (12,12,11,\infty,\infty)$$

le sommet 3 est le prédécesseur de 1

le prédécesseur de sommet 3est le sommet qui prend le min{ $C^1[5,h] + C[h,3]$ }:

$$C^1[5,h] + C[h,3] = (\infty,6,\infty,\infty,0) + \begin{pmatrix}3\\3\\0\\\infty\\\infty\end{pmatrix} = (\infty,9,\infty,\infty,\infty)$$

le sommet 2 est le prédécesseur de 3

Alors le plus court chemin entre 5 et 4 passe par les sommets : 5, 2, 3, 1,4.

2.2.2. Algorithme de Dijkstra – Moore (1959) :

Principe de fonctionnent de l'algorithme : [Lab.81][Lop.05]

Cet algorithme Permet de trouver les coûts minimums de chemins de i à j , $j \in X - \{i\}$.

Il ne permet pas de considérer les arcs négatifs.

Les sommets reçoivent des étiquettes λ_j qui diminueront de ∞ jusqu'au coût minimum d'un chemin de i à j.

- Au début, $\lambda_i = 0$ et $\lambda_j = \infty$, $j \in X - \{i\}$.
- On compare systématiquement $\lambda j + \gamma$ (i, j) et λ_j, et λ_j devient le plus petit de ces deux

nombres.

L'algorithme :

Variable Graphe G = (X,U); i,j,p : sommet; S : ensemble de sommets; λ : entier;

1. Choisir i;
2. Posons $\lambda_i = 0$ et $\lambda_j = \infty$, $j \in X - \{i\}$; p = i ; S:= {i } .
3. Pour $j \notin S$, on remplace λ_j par min (λj, $\lambda_p + \gamma$ (p, j)), $\forall$ j tel que (p,j) $\in$ U.
4. p := le sommet de X - S pour lequel λ est minimum. S := S $\cup$ {p}.
5. Si S =X, Fin; sinon retourner à 3.
6. Fin : λ_j = coût minimum d'un chemin de i à j.

La complexité de l'algorithme est : O(m log n).

Exemple : on applique l'algorithme ci-dessus sur le graphe de la Fig.2.2.

Après choisir le sommet 4 on obtient le tableau suivant :

$\lambda 1$	$\lambda 2$	$\lambda 3$	$\lambda 4$	$\lambda 5$	p	S
∞	∞	∞	0	∞	4	{4}
∞	∞			3_4	5	{4,5}
∞	9_5	∞			2	{4,5,2}
15_2		12_2			3	{4,5,2,3}
14_3					1	{4,5,2,3,1}

Pour obtenir le plus court chemin entre le sommet choisit (4) et les autres sommets, on va déterminer les prédécesseurs successifs du deuxième sommet jusqu'à atteindre le sommet choisit.

Les plus courts chemins partant du sommet 4 vers les autres sommets :

(4,1) : 4 → 5 → 2→ 3→ 1 = 14.

(4,2) : 4 → 5 → 2 = 9.

(4,3) : 4 → 5 → 2 → 3 = 12.

(4,5) : 4 → 5 = 3.

2.2.3. Algorithme de Floyd (1962) :

Principe de fonctionnement de l'algorithme : [Lop.05] [Mar.03]

Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de circuit de poids strictement négatif. Il y a n itérations dans cet algorithme (n étant le nombre de sommets). A l'initialisation, on a une matrice de coûts C et un autre matrice $P = (p_{i,j})_{i,j=1..n}$

$$P_{i,} = \begin{cases} 0 & \text{si } c_{i,j}=\infty \\ i & \text{sinon.} \end{cases}$$

A chaque itération k, on va faire pour tous les i et j : $c_{i,j} = \min(c_{i,j}, c_{i,k} + c_{k,j})$ $(i \neq k)$c'est-à-dire qu'on va comparer la valeur actuelle de c_{ij}, avec la valeur qu'on aurait en passant par k. En fin d'algorithme on obtient :

$c_{i,j}$ = longueur du plus court chemin entre i et j

$p_{i,j}$ = prédécesseur de j sur le plus court chemin de i `a j

Cet algorithme détecte l'existence d'un circuit absorbant (si C[i,k] + C[k,i] < 0).

L'algorithme :

Variable Graphe G = (X,U); i,j,n : entier; C,P : tableau d'entiers

1. Posons i =1; j =1;
2. Si C[i,j] = ∞ alors P[i,j] := 0 sinon P[i,j] := i;
3. j := j+1 si j <= n alors retourner en 2;
4. i := i+1 si i <= n alors j :=1; retourner à 2;
5. Posons i =1; j =1; k = 1;
6. Si i = k aller à 10 ;
7. Si C[i,k] + C[k,i] < 0 aller à 12. // detecter un circuit absorbant.
8. Si C[i,k] + C[k,j] < C[i,j] alors C[i,j] := C[i,k] + C[k,j]; P[i,j] := P[k,j];
9. j := j+1 si j <= n alors retourner à 8;
10. i := i+1 si i < =n alors j :=1; retourner à 6;
11. k := k+1 si k < = n alors i :=1; j :=1; retourner à 6;
12. Fin.

La complexité de l'algorithme est : $O(n^3)$.

Exemple : soit le graphe suivant :

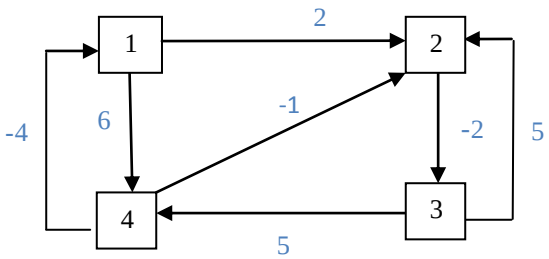


Fig.2.3: Graphe pondéré avec des arcs négatifs.

L'application de l'algorithme sur ce graphe donne les résultats suivants :

Initialisation : les matrices C, P.

C =

	1	2	3	4
1	0	2	∞	6
2	∞	0	-2	∞
3	∞	5	0	5
4	-4	-1	∞	0

P =

	1	2	3	4
1	1	1	0	1
2	0	2	2	0
3	0	3	3	3
4	4	4	0	4

K=1

C=

	1	2	3	4
1	0	2	∞	6
2	∞	0	-2	∞
3	∞	5	0	5
4	-4	-2	∞	0

P =

	1	2	3	4
1	1	1	0	1
2	0	2	2	0
3	0	3	3	3
4	4	1	0	4

K=2

C=

	1	2	3	4
1	0	2	0	6
2	∞	0	-2	∞
3	∞	5	0	5
4	-4	-2	-4	0

P =

	1	2	3	4
1	1	1	2	1
2	0	2	2	0
3	0	3	3	3
4	4	1	2	4

K=3

C =

	1	2	3	4
1	0	2	0	5
2	∞	0	-2	3
3	∞	5	0	5
4	-4	-2	-4	0

P =

	1	2	3	4
1	1	1	2	3
2	0	2	2	3
3	0	3	3	3
4	4	1	2	4

K=4

C =

	1	2	3	4
1	0	2	0	5
2	-1	0	-2	3
3	1	3	0	5
4	-4	-2	-4	0

P =

	1	2	3	4
1	1	1	2	3
2	0	2	2	3
3	1	4	3	3
4	4	1	2	4

Pour obtenir le plus court chemin de sommet i vers le sommet j, il suffit d'utiliser la ligne numéro i de la matrice P.

Par exemple, si on veut obtenir le plus court chemin de 4 vers 3, on consulte la matrice P ainsi : P[4,3] = 2 : 2 est donc le prédécesseur de 3, P[4,2] = 1 : 1 est donc le prédécesseur de 2, P[4,1] = 4 : 4 est donc le prédécesseur de 1, Finalement on a le chemin: 4 → 1 → 2 → 3.

2.2.4. Algorithme de Bellman – Ford (1958-1962) :

2.2.5. Principe de fonctionnement de l'algorithme : [Lop.05]

Cet algorihme est valable pour des graphes sans circuit négatif. Il permet de trouver les coûts minimums de chemins de x à y , $y \in X - \{x\}$. Il y a n-1 itérations dans cet algorithme (n étant le nombre de sommets). A l'initialisation, on a deux tableaux 'd' et 'pred' tel que :

$d_0(x) = 0$, pred(x) = 0.

$d_0(y) = \infty$, pred(y) = 0 / $x \neq y$.

A chaque itération (k) on va appliquer la formule suivante :

Si $d_k(i) > d_{k-1}(j) + \gamma(j,i)$ alors $d_k(i) := d_{k-1}(j) + \gamma(j,i)$; pred(i) = j; / $(j,i) \in U$

S'il n'y a pas de circuit de poids négatif, l'algorithme de Bellman-Ford retourne vrai, sinon il retourne faux : en effet, dans ce cas, il n'existe pas de plus court chemin à partir de l'origine (permet de détecter l'existence d'un circuit absorbant).

L'algorithme :

Variable Graphe G = (X,U); i , j , x , y: sommet ; k , n : entier; existe : booléen;
d : tableau d'entiers ; pred : tableau de sommets;

1. Choisir i;
2. Posons d[i] = 0; d[j] = ∞ ; pred[j] = 0 / $j \in X - \{i\}$;
3. Posons k =1; x =1; y = 1;
4. Si $(y,x) \notin U$ alors aller à 6;
5. Si d[x] > d[y] + γ(y,x) alors d[x] := d[y] + γ(y,x); pred[x] := y;
6. y := y+1 si y <= n alors retourner en 4;
7. x := x+1 si x <= n alors y := 1; retourner à 4;
8. k := k+1 si k <= n-1 alors x:=1; y := 1; retourner à 4;
9. Posons x =1; y = 1; existe = faux; //détecter un circuit absorbant.
10. Si $(y,x) \notin U$ alors aller à 12;
11. Si d[x] > d[y] + γ(y,x) alors existe := vrai;
12. y := y+1 si y <= n et existe =faux alors retourner à 10;
13. x := x+1 si x <= n et existe =faux alors y := 1; retourner en 10;
14. Si existe = vrai alors il existe un circuit de négative sinon n'existe pas;
15. Fin.

La complexité de l'algorithme est : O(nm).

Exemple : on applique l'algorithme ci-dessus sur le graphe de la Fig.2.4.

on choisir le sommet 1 comme sommet de départ, on obtient le tableau suivant :

k / d	1	2	3	4
1	0	2_1	0_2	5_3
2	0	2_1	0_2	5_3
3	0	2_1	0_2	5_3
4	0	2_1	0_2	5_3

Pour obtenir le plus court chemin entre le sommet choisit 1 et les autres sommets on va déterminer les prédécesseurs successifs du deuxième sommet jusqu'à atteindre le sommet choisi.

Les plus courts chemins partant du sommet 1 vers les autres sommets :

(1,2) : $1 \rightarrow 2 = 2$.

(1,3) : $1 \rightarrow 2 \rightarrow 3 = 0$.

(1,4) : $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 = 5$.

Observation :

On a pu voir ici comment déterminer le plus court chemin dans un graphe, grâce à des algorithmes connus (Maria Hass, Djikstra, …).

La question la plus essentielle est quel algorithme choisir pour la résolution de son graphe ? Cela peut dépendre de beaucoup de choses, la présence ou non de circuit absorbant, de longueur d'arc négatif, mais aussi de la complexité de l'algorithme.

2.3. Algorithmes de recherche du plus long chemin :

On peut également modifier l'algorithme de Bellman pour trouver le chemin de coût maximum.

2.3.1. Algorithme de Bellman – Ford modifié :

Principe de fonctionnement de l'algorithme :

L'algorithme Bellman – Ford modifié a le même principe de l'algorithme Bellman – Ford mais il suffit modifier la ligne 5 par :

Si $d[x] > d[y] + \gamma(y,x)$ alors $d[x] := d[y] + \gamma(y,x)$; pred[x] := y;

2.4. Algorithmes de recherche d'un arbre recouvrant minimum :

La recherche de l'arbre de poids minimum d'un graphe peut être résolu par des méthodes heuristiques. Nous allons donc expliciter le problème de recherche d'un tel arbre et appliquer deux algorithmes afin de le constituer.

2.4.1. Algorithme de Kruskal (1956) :

Principe de fonctionnement de l'algorithme : [Mar.03][S6]

- Trier les arcs par ordre croissant de leur poids.
- Ajouter les arcs un par un dans un graphe G1 pour construire progressivement l'arbre.
- Un arc est ajouté si et seulement si son ajout dans G1 n'introduit pas de cycle sinon on passe à l'arc suivant dans l'ordre du tri.

L'algorithme :

Variable Graphe G = (X,U); F : ensemble de sommets; i : entier;

1. Trier les arcs de G dans l'ordre croissant des poids; [On les notera $u_1 ... u_m$]
2. Posons i =1; Posons F = Ø;
3. Si F ∪ $\{u_i\}$ est acyclique alors F := F ∪ { u_i } ;
4. i := i+1 ;
5. Si |F| ≠ n-1 alors retourner à 3;
6. Fin.

La complexité de l'algorithme est : O(m log m).

Exemple : on applique l'algorithme ci-dessus sur le graphe suivant : on choisir le sommet 1

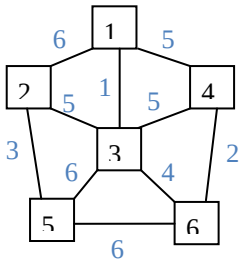
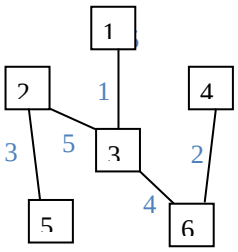


Fig.2.4: graphe non orienté.

On obtient le tableau suivant :

ordre	Arête	poids
1	(1,3)	1
2	(4,6)	2
3	(2,5)	3
4	(3,6)	4
5	(2,3)	5
6	(1,4)	5
7	(3,4)	5
8	(1,2)	6
9	(3,5)	6
10	(6,5)	6

L'arbre recouvrant minimum



Terminer

Fig.2.5: Exemple d'application de l'algorithme de Kruskal.

2.4.2. Algorithme de Prim (1957):

Principe de fonctionnement de l'algorithme : [Mar.03][S6]

- choisir arbitrairement un sommet.
- A chaque itération choisir le sommet non relie j le plus prés d'un des sommets déjà relies i et ajouter{i,j}.
- Arrêter lorsque tous les sommets ont été relies.

L'algorithme :

Variable Graphe G = (X,U); F,S : ensemble de sommets; i,x,y : sommet;

1. Posons F = Ø; S = Ø;
2. Choisir arbitrairement un sommet i;
3. S := {i};
4. Choisir une arête (x,y) de coût minimal tel que x ∈ S et y ∈ X- S;
5. S := S ∪ {y} ; F := F ∪ {(x,y)};
6. Si S = X alors retourner à 4;
7. Fin.

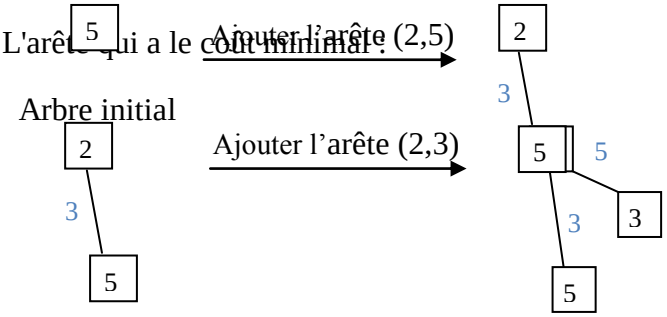
La complexité de l'algorithme est : O(m log n).

Exemple : on appliquons l'algorithme ci-dessus sur le graphe de la Fig.2.4 :

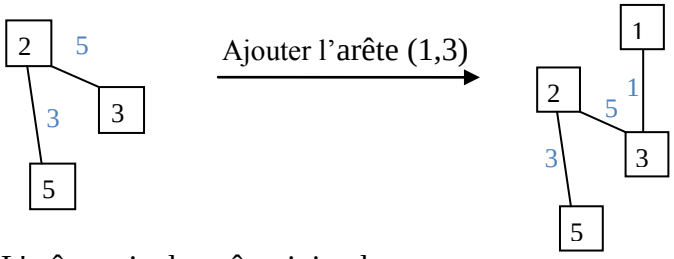
nous choisissions le sommet 5 comme sommet de départ.

L'arête (2,5) : c'est L'arête qui a le coût minimal :

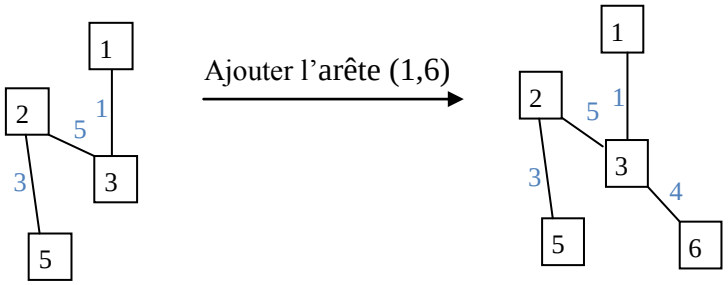
L'arête (2,3) : c'est L'arête qui a le coût minimal :



L'arête (1,3) : c'est L'arête qui a le coût minimal :



L'arête (3,6) : c'est L'arête qui a le coût minimal :



L'arête (4,6) : c'est L'arête qui a le coût minimal :

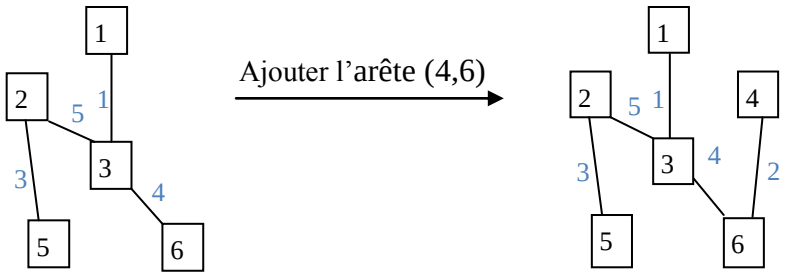


Fig.2.6 : Exemple d'application de l'algorithme de Prim.

Observation :

Si m est petit, alors l'algorithme de Kruskal est préférable sinon on applique l'algorithme de Prim.

2.5. Algorithmes de recherche du Flot maximum :

Le problème du flot maximum consiste à trouver la quantité maximale de flot qui peut circuler de la source au puits. L'algorithme le plus connu pour résoudre ce problème est celui de Ford et Fulkerson.

Nous verrons deux approches pour cette méthode :

- La première est basée sur la cherche d'une chaîne dans le réseau
- La seconde construit un graphe "d'écart" dans lequel on cherche un chemin.

2.5.1. Algorithme de Ford-Fulkerson1(chaine améliorante):

Principe de fonctionnement de l'algorithme : [S7]

On commence par un flot nul.

Chaque itération consiste à chercher une Chaîne améliorante C de s à p dans G

La chaîne C est améliorante si :

- ✓ pour tout arc u direct, f(u) < c(u).
- ✓ pour tout arc u indirect, f(u) > 0.

Le flot sur cette chaîne C peut être augmenté de:

min({c(u) - f(u) | u ∈ C et u sens direct} ∪ {f(u) | u ∈ C et u sens indirect}) .

L' algorithme Ford-Fulkerson1 :

Variable Réseau(X,U, γ), s,p:sommet, C,S,S1,S2 : ensemble d'arcs, u : arc, m,a : réel ;

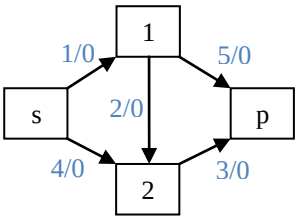
c() : capacité courante, f() : le flot courant, F : entier.

1. Posons S = Ø; S1 = Ø; S2 = Ø; F = 0;
2. Pour u ∉ S S := S ∪ {u}; f(u) .= 0;
3. Si S ≠ U alors retourner à 2;
4. Si il n'existe pas une chaîne améliorante C entre s et p alors aller à 15;
5. Posons S1 = Ø; S2 = Ø;
6. m := ∞;
7. Pour u ∈ C S1 := S1∪ {u};
8. Si u en sens directe alors a := c(u) – f(u) sinon a:= f(u);
9. Si a < m alors m := a;
10. Si S1 ≠ C alors retourner à 7;
11. Pour u ∈ C S2 := S2 ∪ {u};
12. Si u en sens directe alors f(u) := f(u) + m sinon f(u) := f(u) – m; F := F+m ;
13. Si S2 ≠ C alors retourner à 10;
14. Si il existe u une chaîne améliorante C entre s et p alors retourner à 5;

15. Fin.

La complexité de l'algorithme est : O(mF).

Exemple : On applique l'algorithme de Ford-Fulkerson1 sur le graphe de Fig.2.7 :



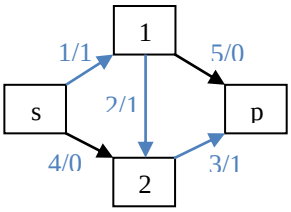
F =0

Fig.2.7 : Réseau transport

On choisit la chaîne améliorante (s , 1 , 2 , p) .

m = min({c(u) - f(u) | u C et u sens direct} {f(u) | u C et u sens indirect})

= min((1-0) , (2-0) , (3-0)) = 1. F = 0+1=1 ;



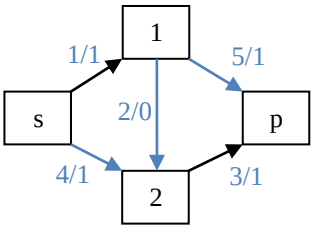
F =1

Fig.2.8 : Flot après une amélioration

On choisit la chaîne améliorante (s , 2 , 1, p) .

m = min({c(u) - f(u) | u C et u sens direct} {f(u) | u C et u sens indirect})

= min((4-0) , (2-1) ,(5-0)) = 1. F = 1+1 = 2 ;



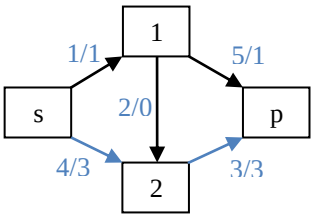
F =2

Fig.2.9 : Flot après deux améliorations.

On choisit la chaîne améliorante (s , 2 , p) .

m = min({c(u) - f(u) | u C et u sens direct} {f(u) | u C et u sens indirect})

= min((4-1) , (3-1)) = 2. F = 2+2 = 4 ;



F =4

Fig.2.10 : Flot après trois améliorations.

Il n'existe pas une chaîne améliorante dans le flot du Fig.2.9 alors F = 4.

Le flot maximum = 4.

2.5.2. Algorithme de Ford-Fulkerson2(Graphe d'écart) : [S7].

Principe de l'algorithme :

1. L'algorithme part d'un flot nul.
2. Construit un graphe d'écart à partir du réseau de transport :
   - ✓ un arc de x à y de capacité c'((x;y)) = c(u) - f(u) si c(u) > f(u),
   - ✓ un arc de y à x de capacité c'((y;x)) = f(u) si f(u) > 0.
3. Déterminer un chemin C dans ce graphe d'écart de la source vers le puits. Si un tel chemin n'existe pas alors aller à 6;
4. Modifier le flot sur tout arc de C (& = min{c'(u) | u ∈ C}, le flot est augmenté de & si u est un arc dans le réseau de transport sinon le flot est diminué);
5. Retourner à 2;
6. Fin.

La complexité de cet algorithme est : $O(nm^2)$.

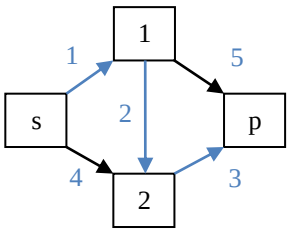
L'algorithme

Variable Réseau(X,U, γ), s,p : sommet, C,S,S1,S2 : ensemble d'arcs, u : arc, m : réel, G = (X;U1) un graphe, f() une fonction indiquant le flot circulant à travers chaque arc, c'() une fonction indiquant la capacité d'un arc de G, a() une fonction indiquant à quel arc de R correspond un arc de G; F :entier ;

1. Posons S = Ø; S1 = Ø; S2 = Ø; F =0;
2. Pour u ∉ S S := S ∪ {u}; f(u) := 0;
3. Si S ≠ U alors retourner à 2;
4. Construire le graphe d'écart G;
5. Si il n'existe pas un chemin C entre s et p alors aller à 15;
6. Posons S1 = Ø; S2 = Ø;
7. m := ∞;
8. Pour u ∈ C S1 := S1∪ {u};

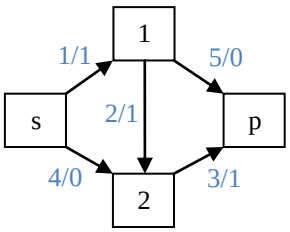
9. Si c'(u) < m alors m := c'(u);
10. Si S1 ≠ C alors retourner à en 8;
11. Pour u ∈ C S2 := S2∪ {u};
12. Si u en sens directe alors f(a(u)) := f(a(u)) + m sinon f(a(u)) := f(a(u)) – m;
    F := F+m ;
13. Si S2 ≠ C alors retourner à 11;
14. Construire le graphe d'écart G et retourner à 5;
15. Fin.

Exemple : On applique l'algorithme de Ford-Fulkerson2 sur le graphe de Fig.2.7 :



(s , 1 , 2 , p)

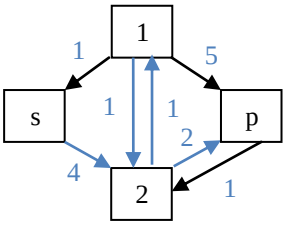
m = 1



F=1

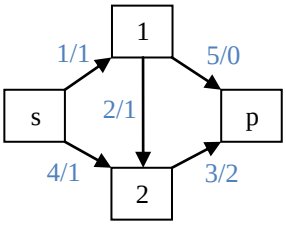
Graphe d'écart avant la 1ére itération

Graphe de flot après la 1ére itération



(s , 2 , 1 , 2 , p)

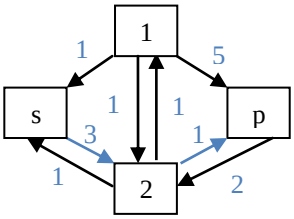
m = 1



F=2

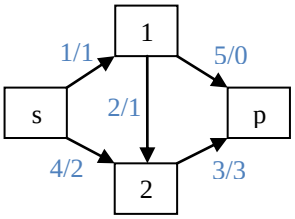
Graphe d'écart après la 1ére itération

Graphe de flot après la 2éme itération



(s , 2 , p)

m = 1



F=3

Graphe d'écart après la 2éme itération

Graphe de flot après la 3éme itération

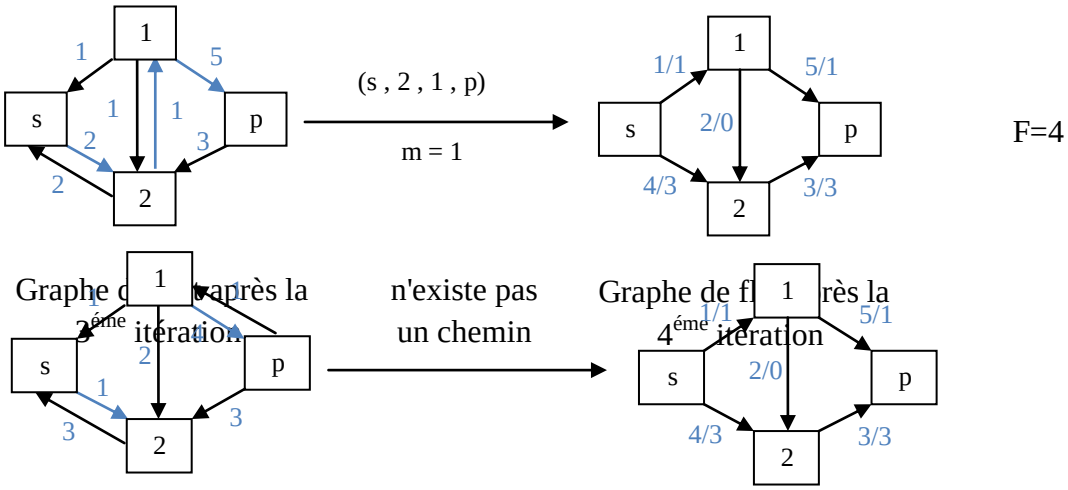


Fig.2.11 : Exemple d'application de l'algorithme Ford-Fulkerson2

Le flot maximum = 4.

2.6. Algorithme de recherche d'un flot maximal de coût minimal :

En pratique, on associe à la plupart des réseaux de transport, une notion du coût unitaire du transport qui se traduit par l'association, à chaque arc (u) du réseau, d'un nombre réel d(u) représentant ce coût unitaire.

La recherche de flot maximale est associe par le coût minimale.

Algorithme de Roy-Busackar –Gown (1961) :

Cet algorithme permet de trouver un flot maximal de coût minimal.

Principe de l'algorithme : [S8]

1. On commence par un flot nul.
2. On construit un graphe d'écart à partir du réseau de transport;
   - ✓ si c(u) > f(u) alors :
     on ajout un arc de x à y de capacité c'((x,y)) = c(u) - f(u) , d'(x,y) := d(x,y).
   - ✓ si f(u) > 0 alors :
     on ajout un arc de y à x de capacité c'((y,x)) = f(u), d'(y,x) := -d(x,y).
3. On détermine le plus court chemin par l'un des algorithmes de recherche du plus court chemin (Bellman, Floyd) dans ce graphe d'écart de la source au puits.
   Si un tel chemin n'existe pas alors aller à 6;
4. On modifie le flot sur tout arc de C (& = min{c'(u) | u ∈ C, le flot est augmenté de & si u est un arc dans le réseau de transport sinon le flot est diminué);

5. Retourner à 2;
6. Fin.

L'algorithme :

Variable Réseau(X,U, γ), s,p : sommet, C,S,S1,S2 : ensemble d'arcs, u : arc,
m, m1 : réel, G = (X;U1) un graphe, f() une fonction indiquant le flot circulant à travers chaque arc, c'() une fonction indiquant la capacité d'un arc de G,
a() une fonction indiquant à quel arc de R correspond un arc de G; F,D : entier ;

1. Posons S = Ø; S1 = Ø; S2 = Ø; F :=0; D :=0;
2. Pour u $\notin$ S S := S $\cup$ {u}; f(u) := 0;
3. Si S $\neq$ U alors retourne en 2;
4. Construire le graphe d'écart G;
5. Recherche dans G le plus court chemin ;
6. Si il n'existe pas un chemin C entre s et p alors aller à 16;
7. Posons S1 = Ø; S2 = Ø;
8. m := ∞;
9. Pour u $\in$ C S1 := S1$\cup$ {u};
10. Si c'(u) < m alors m := c'(u); m1 := d'(u) ;
11. Si S1 $\neq$ C alors retourner à 9;
12. Pour u $\in$ C S2 := S2$\cup$ {u};
13. Si u en sens directe alors f(a(u)) := f(a(u)) + m ; sinon f(a(u)) := f(a(u)) – m;
    F := F +m ; D := D +m1 ;
14. Si S2 $\neq$ C alors retourner à 12;
15. Construire le graphe d'écart G et retourner à 5;
16. Fin.

Exemple : On applique l'algorithme de Roy-Busackar –Gown sur le graphe de la Fig.2.12

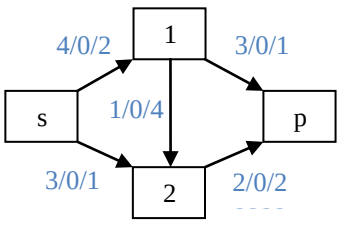
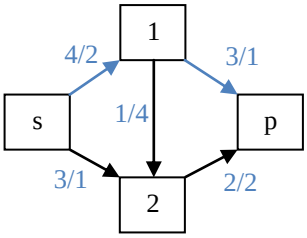


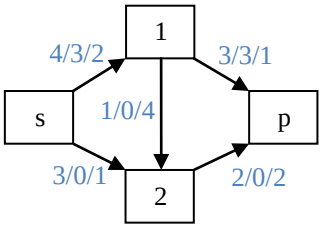
Fig.2.12 : Réseau de transport.



Graphe d'écart avant la 1ére itération

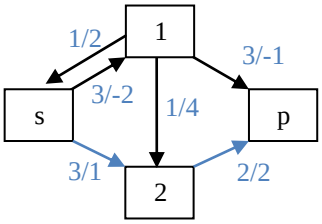
C = s , 1 , p

m =3; d = 1



Graphe de flot après la 1ére itération

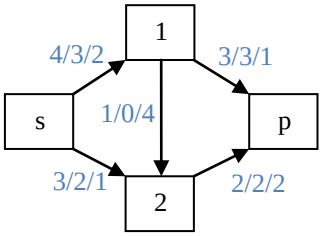
F = 0 + 3 = 3 ; D = 0 +1 = 1;



Graphe d'écart après la 1ére itération

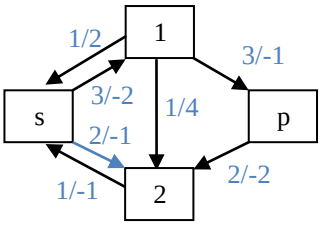
C = s , 2 , p

m =2; d = 2



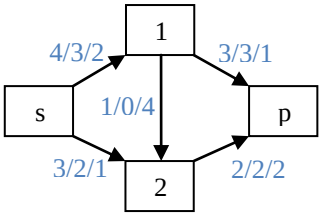
Graphe de flot après la 2éme itération

F = 3+2 = 5 ; D = 1+2 =3;



Graphe d'écart après la 2éme itération

n'existe pas un chemin



le flot maximal

Fig.2.13 : Exemple d'application de l'algorithme Ford-Fulkerson2

Conclusion

Dans ce chapitre nous avons défini les fonctions de manipulation des graphes et on a présenter le principe de chaque fonction et leur pseudo code avec des exemples explicatifs. Dans le chapitre suivant on va présenter notre bibliothèque ainsi qu'une application de démonstration.

Chapitre 3

Réalisation de la bibliothèque

Dans ce chapitre nous présentons notre bibliothèque qui rassembler les différentes fonctions de manipulation des graphes, dans le langage de programmation c++.
Le but de ce chapitre est de présenter les étapes de la réalisation d'une bibliothèque, la structure de données utilisée et on termine par une application de démonstration. .

1. Définition d'une bibliothèque :

Une bibliothèque ou librairie logicielle (ou encore, bibliothèque de programmes) est un ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

Les fonctions sont regroupées par leur appartenance à un même domaine conceptuel (mathématique, graphique,…).

Les bibliothèques logicielles se distinguent des exécutables dans la mesure où elles ne représentent pas une application. Elles ne sont pas complètes, elles ne possèdent pas l'essentiel d'un programme comme une fonction principale et par conséquent ne peuvent pas être exécutées directement. Les bibliothèques peuvent regrouper des fonctions simples comme des fonctions complexes avec de nombreuses fonctions internes non accessibles directement.

Il existe deux types de bibliothèques : les bibliothèques statiques et les bibliothèques dynamiques.

1.1. Bibliothèque statique :

Dans le cas des bibliothèques statiques, les fonctions contenues dans la bibliothèque sont chargées lors de l'édition des liens. C'est pour cela que le fichier .h est nécessaire, il permet d'indiquer quelles fonctions sont utiles. Les fonctions sont inclues dans le fichier exécutable du programme.

Les bibliothèques statiques ont l'extension .lib ou .a.

Les avantages des bibliothèques statiques :

- L'exécutable n'a plus besoin de la bibliothèque.
- Facile à porter vers une autre plateforme.
- Facile à gérer.

Les inconvénients des bibliothèques statiques :

- Un programme obtenu par compilation avec une librairie statique a un fichier exécutable beaucoup plus volumineux.
- Si une librairie statique est mise à jour alors, tout programme qui utilise cette bibliothèque doit être recompilé pour qu'il puisse prendre en compte les modifications.

Exemple : la Fig.3.1 illustre l'édition de lien statique.

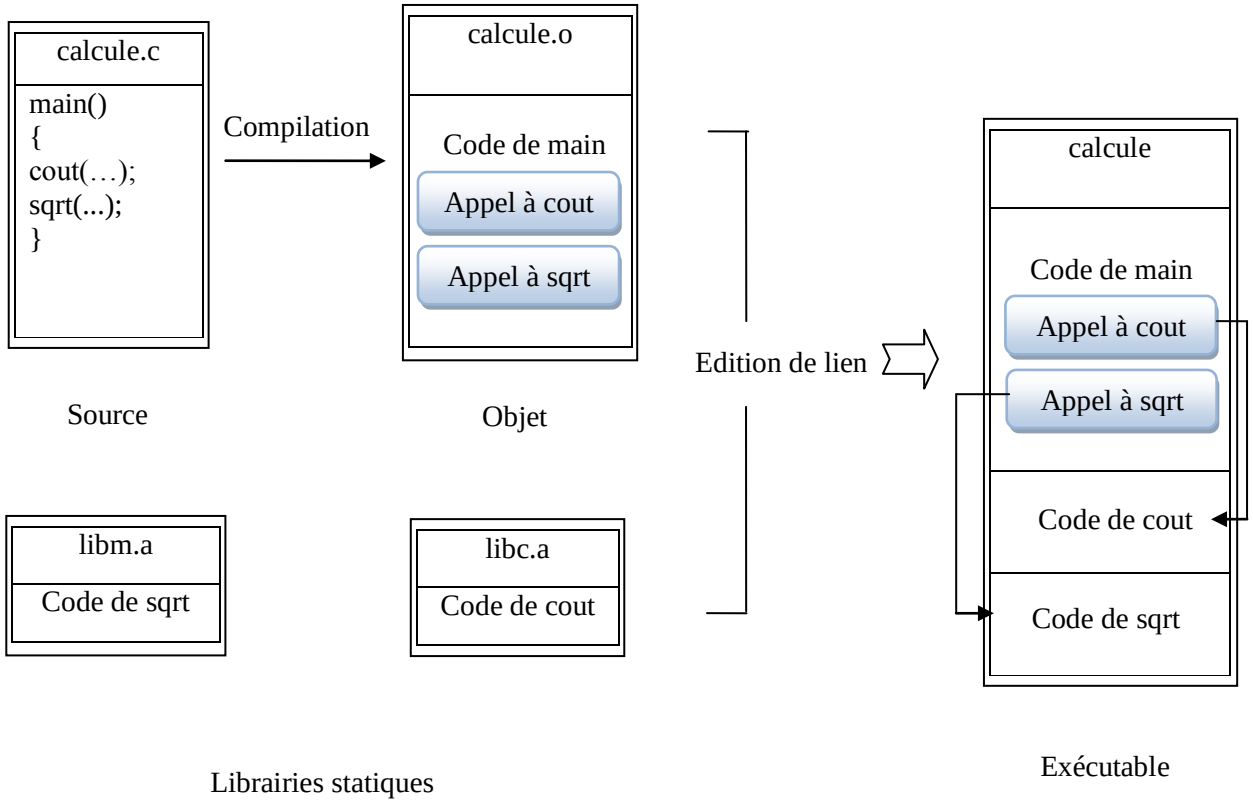


Fig.3.1. édition de lien statique

1.2. Bibliothèque dynamique :

Pour les bibliothèques dynamiques, les fonctions ne sont pas inclues dans l'exécutable.

Les fonctions sont appelées pendant l'exécution du programme tel que la bibliothèque est chargée en mémoire vive lors de l'exécution du programme, voire pendant l'exécution.

Les avantages des bibliothèques dynamiques :

- Si la bibliothèque est utilisée par plusieurs programmes, elle n'est chargée qu'une fois en mémoire.
- L'exécutable est plus léger.
- Modification de la librairie sans recompiler les exécutables qui utilisent cette dernière.

Les inconvénients des bibliothèques dynamiques:

- Il existe une dépendance entre la bibliothèque et l'environnement.
- Si une librairie dynamique a disparu, , un programme exécutable qui s'exécutait parfaitement en utilisant cette librairie peut devenir totalement inopérant.
- Un peu plus compliquée à concevoir qu'une librairie statique.

Exemple : la Fig.3.2 illustre l'édition de lien dynamique.

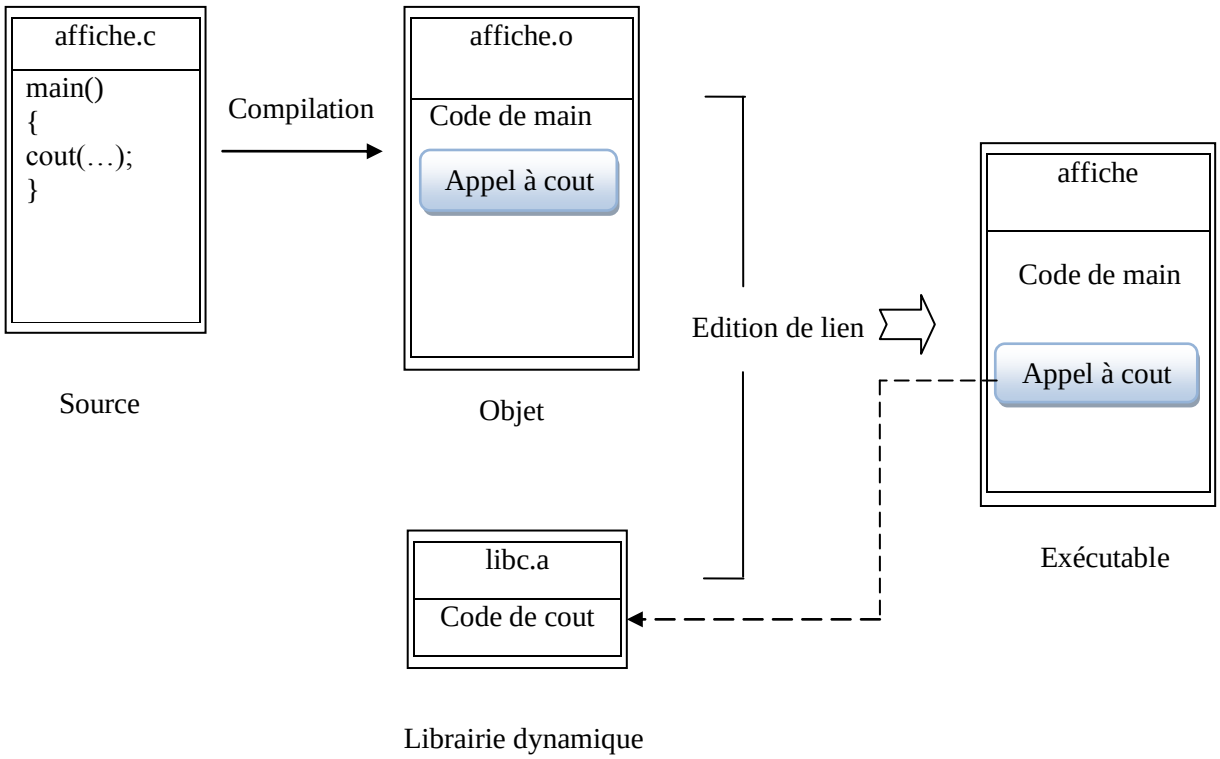


Fig.3.2. édition de lien dynamique

2. La bibliothèque graph :

La bibliothèque graph est une bibliothèque statique à cause :

- Si la bibliothèque statique est modifiée, l'exécutable reste inchangé.
- La portabilité de la bibliothèque statique.
- L'écrasement d'une bibliothèque dynamique par un fichier portant le même nom.
- Facilité de la manipulation.

3. L'utilisation de la bibliothèque « graph » :

Pour utiliser la bibliothèque graph dans un projet quelconque on suit les étapes suivantes :

1. Dans le dossier qui contient la bibliothèque « graph » copier les fichiers graphe.lib et le fichier graph.h.
2. Dans le dossier qui contient le projet coller les deux fichiers graphe.lib et graph.h.

3. Ajoutez au la bibliothèque projet.

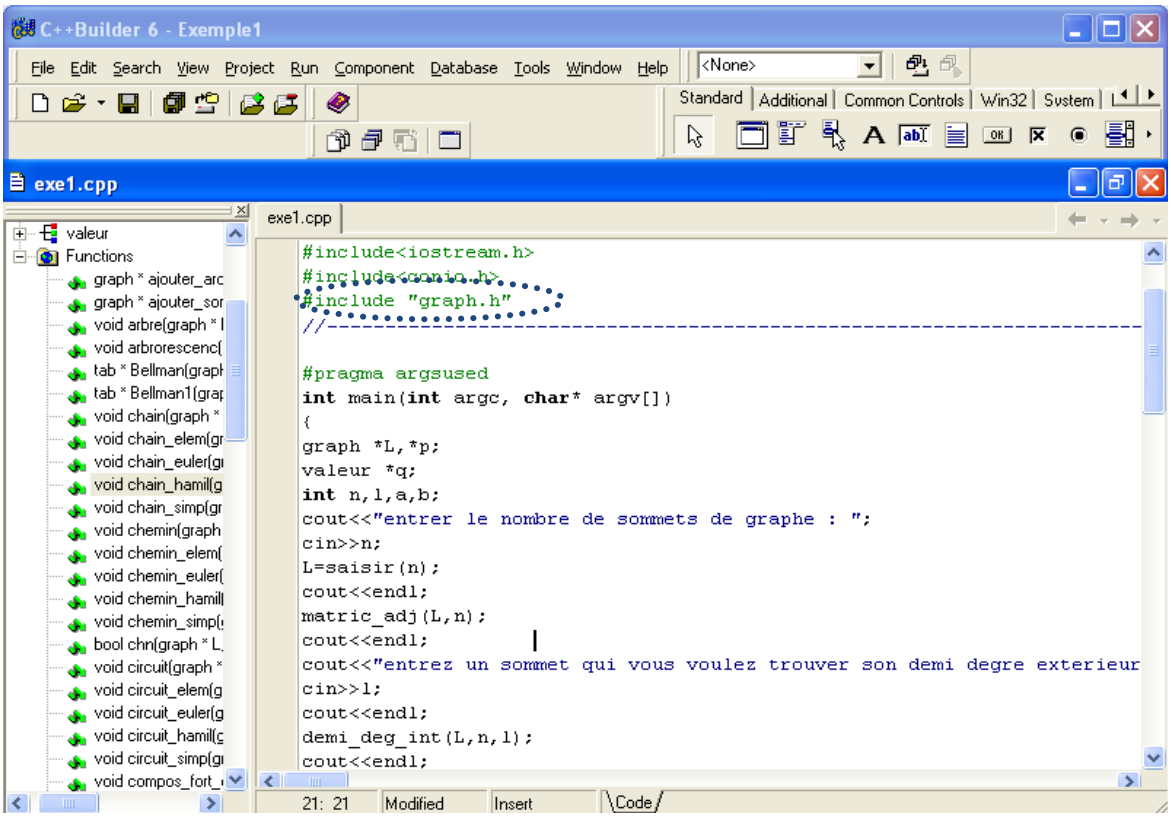


Fig.3.3 intégration la bibliothèque en application.

4. Structure de données :

Le graphe est représenté par une matrice d'adjacences M = $(m_{i,j})_{i,j=1,...,n}$. dans la matrice d'adjacences, les lignes et les colonnes représentent les sommets du graphe.

$m_{i,j}$: est le nombre d'arcs entre les sommets i et j.

Exemple 1 :

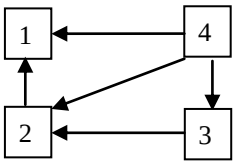


Fig.3.4. digraphe orienté simple.

La matrice d'adjacences du graphe de la Fig.3.4 est la suivante :

M =

	1	2	3	4
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	1	1	1	0

Exemple 2 :

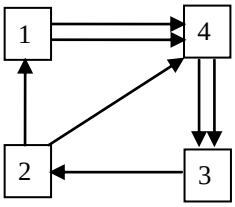


Fig.3.5. multigraphe.

La matrice d'adjacences du graphe de la Fig.3.5 est la suivante :

M =

	1	2	3	4
1	0	0	0	2
2	1	0	0	1
3	0	1	0	0
4	0	0	2	0

Problèmes : les tableaux ne constituent pas la meilleure solution pour stocker et manipuler les données à cause des points suivantes :

- La taille de la matrice étant fixée (structure statique) qui implique le gaspillage d'espace mémoire.
- Pour supprimer (insérer) un élément au milieu du tableau, il faut recopier les éléments temporairement, réallouer de la mémoire pour le tableau, puis le remplir à partir de l'élément supprimé (insérer).

Solution : Pour résoudre ces deux problèmes, on se tourne vers des structures dynamiques : les listes chaînées :

- Une liste chaînée est une structure de données qui a l'avantage de s'adapter facilement aux besoins en espace mémoire.

- Une liste chaînée est différente dans le sens où les éléments de la liste sont répartis dans la mémoire et reliés entre eux par des pointeurs. Alors on peut ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière.

Implémentation avec des listes chaînées :

On implémente la matrice d'adjacences par une liste chaînée comme illustré dans la Fig.3.6 : tel que :

som : représente la ligne (1<=som<=n).

d : représente la colonne (1<=d<=n).

val : représente la valeur entre la ligne som et la colonne d (nombre d'arcs).

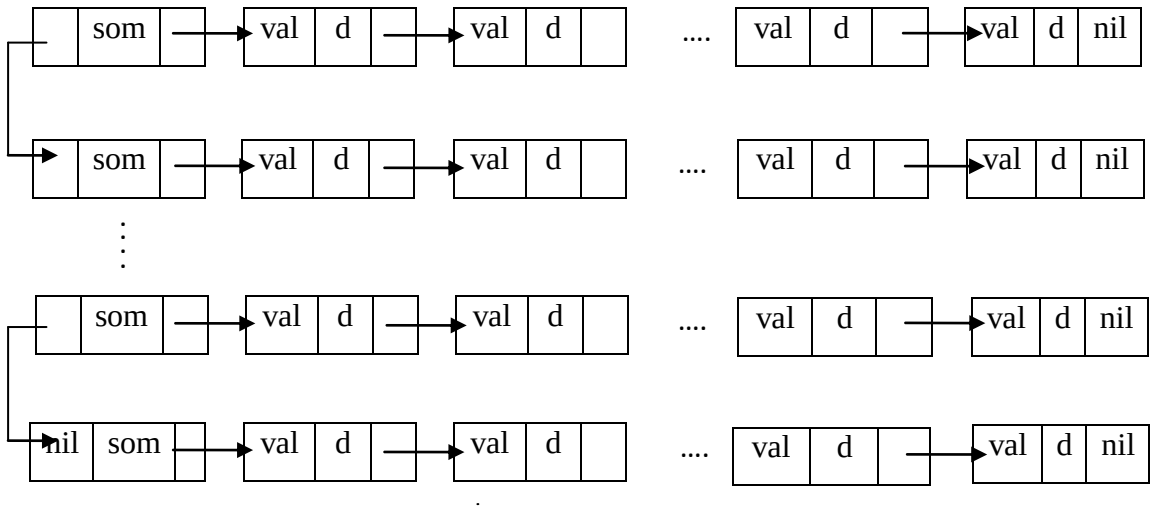


Fig.3.6. La structure de donnée d'un graphe .

5. Environnement de travail :

La bibliothèque **graph** a été développé sur une machine de type Intel Pentium IV, d'une horloge de 3 GHz, et de 1 G de DDRAM, sous Windows Xp.

6. L'outil de développement :

La bibliothèque **graph** est développé sous C++ Builder basé sur C++ , proposé par Borland. C++ Builder est un environnement de programmation visuel orienté objet permettant de développer des applications 32 bits en vue de leur déploiement sous Windows et sous Linux. C++ Builder est un outil RAD, c'est à dire tourné vers le développement rapide d'applications (Rapid Application Development) sous Windows. En un mot, C++ Builder permet de réaliser de façon très simple l'interface des applications et de relier aisément le code utilisateur aux événements Windows, quelle que soit leur origine (souris, clavier, événement système, etc.). [S9].

7. L'application de démonstration :

l'application de démonstration est une simple application qui permet de tester le bon fonctionnement des fonctions de la bibliothèque.

La bibliothèque peut être utilisée dans une application win32 ou application console.

7.1. Application win32 :

7.1.1. Description générale de l'application :

Notre application est caractérisée par une interface simple elle permet de :

- ✓ Tester les fonctions de la bibliothèque :
  - Opérations sur les graphes : graphe inverse, graphe complémentaire,…
  - Recherche plus court chemin : Dijkstra, Floyd,…
  - Recherche arbre recouvrant minimum : Kruskal, Prim.
  - Recherche flot maximum : Ford Fulkerson1, Ford Fulkerson2.
  - Recherche flot maximum à coût minimal : Ford Fulkerson1, Ford Fulkerson2.
- ✓ Consulter le manuel de références

7.1.2. Description détaillée de l'application :

La fenêtre suivante (voir la Fig.3.7) montre l'interface de l'application

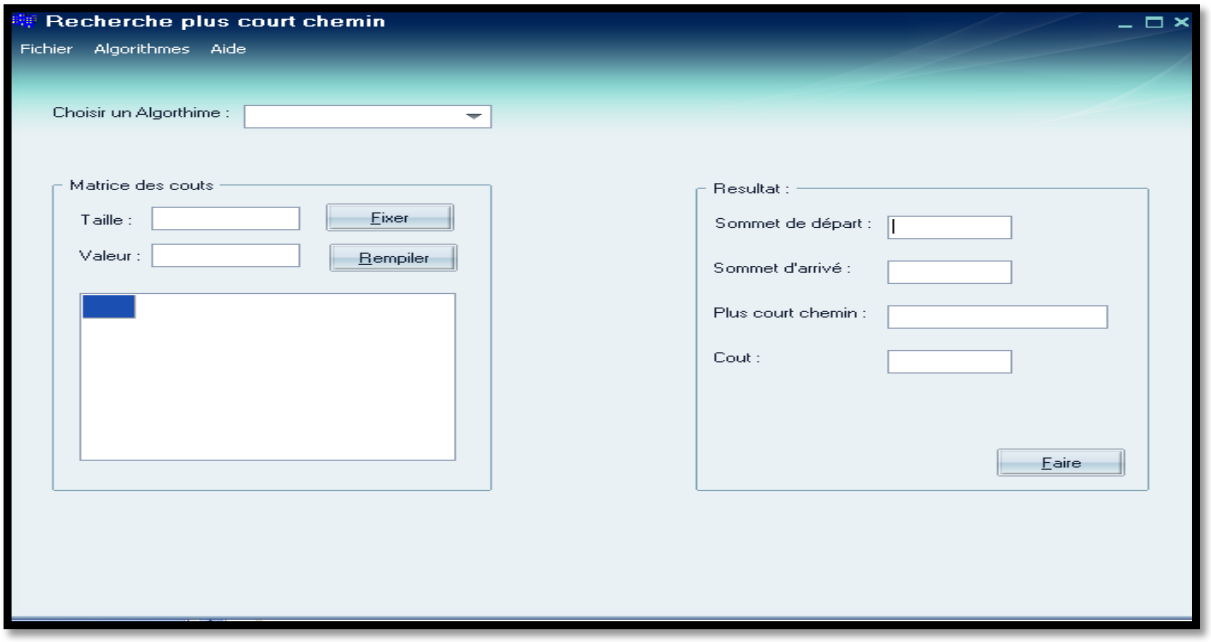


Fig.3.7. Les différentes fonctions de l'application.

Si on choisir l'algorithme de recherche plus court chemin par exemple « Dijkstra » (voir la Fig.3.8), la fenêtre suivante est affiché :

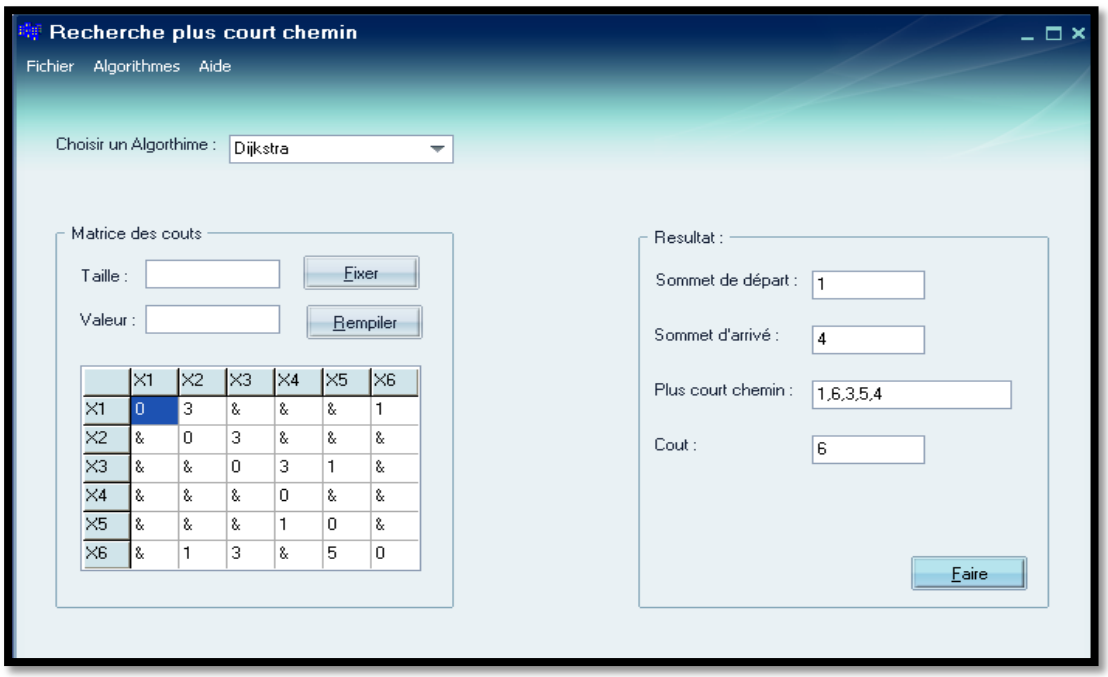


Fig.3.8. Exemple d'exécution d'algorithme Dijkstra.

7.2. Application console :

La figure suivante Fig.3.9 représente l'exécution de la bibliothèque « graph » dans une application console.

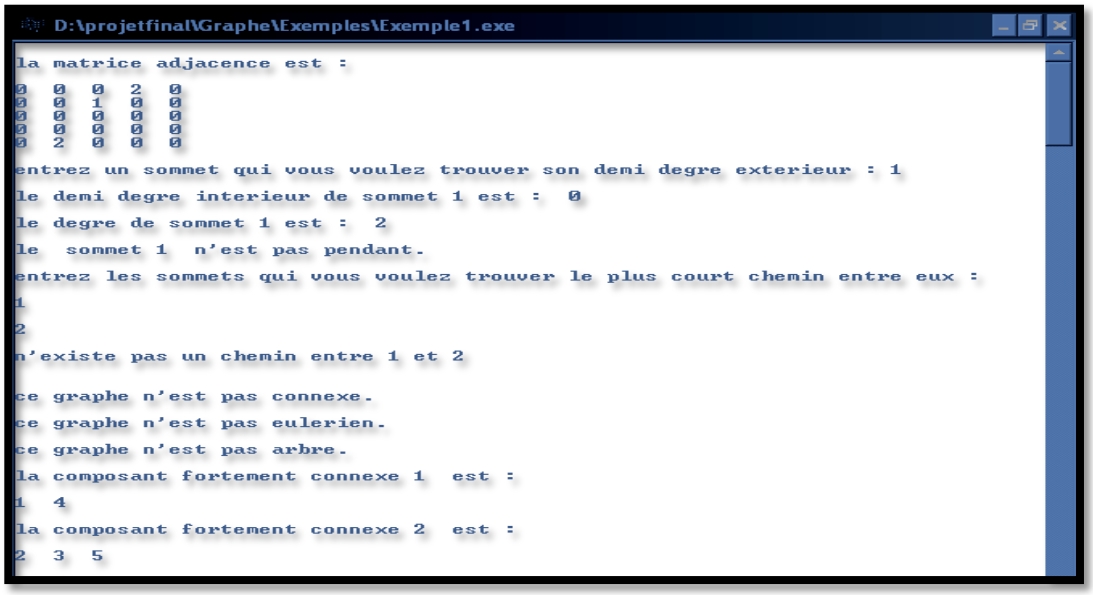


Fig.3.9. Application console.

8. Comparaison avec les autres bibliothèques :

Il existe plusieurs bibliothèques réalisées pour manipuler les graphes on comparons ces bibliothèques avec notre bibliothèque de terme l'implémentation et les fonctions réalisées.

8.1. L'implémentation

Bibliothèque	Plate form	Langage de programmation	Structure de donnée
BGL	Windows + linux	C++	Liste chainée
Ocamlgraph	Windows	Ocaml	Liste chainée
Ruby	Windows + linux	Ruby	Liste chainée
SeqAn	Windows + linux	C++	Liste chainée
Goto	Windows	java	Tableaux
TGL	Windows + linux	C++	Tableaux
graph	Windows	C++	Liste chainée

8.2. Les fonctions :

		graph	BGL	Ocamlgraph	Ruby	SeqAn	Goto	TGL
Suc et préd d'un sommet		✓		✓				
degré		✓		✓				
Le nombre d'arcs		✓						
Sommets particuliers		✓						
Opérations sur les graphes	Ajouter sommet	✓		✓			✓	
	Supprimer sommet	✓		✓			✓	
	Ajouter arc	✓		✓			✓	
	Supprimer arc	✓		✓			✓	
	Fermeture T	✓		✓	✓			
	G .complémentaire	✓		✓				
	G .inverse	✓		✓				
	Union	✓		✓				
	Intersection	✓		✓				
Test chaine, chemin, cycle, circuit		✓						
Test types de chaine chemin, …		✓						
Parcours d'un graphe		✓	✓	✓	✓			✓
connexité		✓	✓		✓			✓
Graphe particuliers		✓						
Tri topologique		✓	✓	✓	✓			
Recherche	Djikstra	✓		✓		✓		
	Maria Hass	✓						

plus court chemin	Floyd	✓	✓					
	Bellman	✓	✓					
	Jonson		✓					
	Dag		✓					
Recherche plus long chemin	Bellman	✓						
Recherche d'un arbre recouvrant minimum	Kruksal	✓	✓	✓				
	Prim	✓	✓			✓		
recherche du Flot maximum	Ford-Fulkerson1	✓	✓	✓		✓		
	Ford-Fulkerson2	✓						
	Goldberg			✓				
	Edmons -Karp		✓					
recherche du Flot maximum à cout minimum	Roy-Gown-Busker	✓		✓				
Couplage minimum		✓				✓		
Coloriage d'un graphe						✓		
Planaire			✓					

Observation :

Les trois bibliothèques graph, BGL, Ocamlgraph implémentent l'essentiels des fonctions de manipulation des graphes, mais notre bibliothèque contient le maximum des fonctions. Certain algorithmes importants ne sont implémentés que dans graph comme l'algorithme de Bellman pour chercher le plus longe chemin, Ford-Fulkerson2, Tester l'existence d'une chaîne, un chemin, un cycle et tester leurs types, ainsi que l'implémentation de l'algorithme Maria Hass. La structure de données utilisée est la liste chaînée, comme dans la majorité des autres bibliothèques.

Graph ne fonctionne pour le moment que sous Windows.

Conclusion

Dans ce travail, on a réalisé une bibliothèque statique de manipulation des graphes.

Cette **bibliothèque** comporte un nombre important de fonctions classiques tel que le parcours en profondeur ou en largeur, calcule la distance,...

Elle comporte également des fonctions plus spécialisées tel que la recherche du chemin plus court, le tri topologique, calcul du flot maximal,…

Dans un premier temps cette **bibliothèque fonctionne sous le système d'exploitation Windows.**

On a développé une application de démonstration pour tester cette bibliothèque et montrer comment l'utiliser.

Enfin, on souhaite l'améliorer surtout on se qui concerne :

1. L'ajout d'autres fonctions et des algorithmes (coloriage, planarité, ...).
2. L'adaptation de la bibliothèque pour d'autres plate formes tel que : Linux, Mac Os, …
3. Restructurer le code de cette bibliothèque afin qu'il respect la structure d'un code open source.
4. La disponibilité sur le web, pour que d'autres peuvent l'utiliser ou contribuer à son amélioration.

Annexe 1

Opérations sur les graphes

Dans cette annexe nous présentons d'autres opérations non sitées dans le chapitre 1.

Soit le graphe suivant :

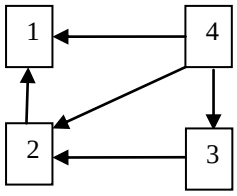


Fig.1 : Graphe orienté

1. Ajouter un sommet :

On ajoute le sommet v au graphe G lorsque v ∉ X.

Exemple : on ajoute le sommet 5 au le graphe Fig.3.1, On obtient le graphe suivant :

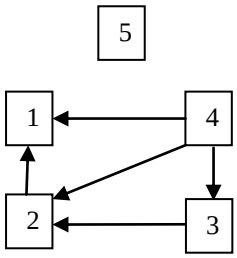


Fig.2: Graphe orienté

2. Supprimer un sommet :

On supprime le sommet v au graphe G lorsque v ∈ X.

Exemple : on supprime le sommet 3 dans le graphe Fig.3.1, On obtient le graphe suivant :

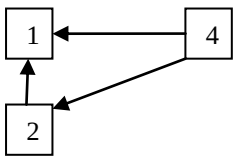


Fig.3 : Graphe orienté

3. Ajouter un arc :

On ajout d'un arc (x,y) au graphe G entre deux sommet x et y.

Exemple : on ajoute l'arc (5,1) et (4,5) au le graphe de la Fig.3.2, On obtient le graphe suivant :

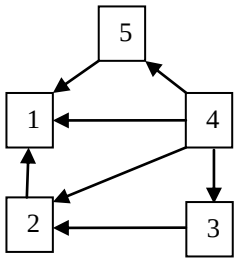


Fig.4 : Graphe orienté

4. Supprimer un arc :

On supprime l'arc (x,y) dans le graphe G tel que (x,y) ∈ U.

Exemple : on supprime l'arc (4,2) dans le graphe de la Fig.3.4, On obtient le graphe suivant :

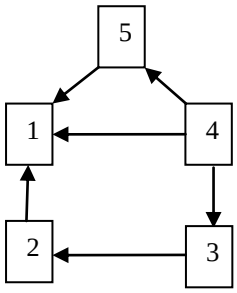


Fig.5 : Graphe orienté

5. Union :

Soient les graphes G1 = (X , U1) et G2 = (X , U2) :

G1 ∪ G2 = (X , U1) ∪ (X , U2) = (X , U1 ∪ U2).

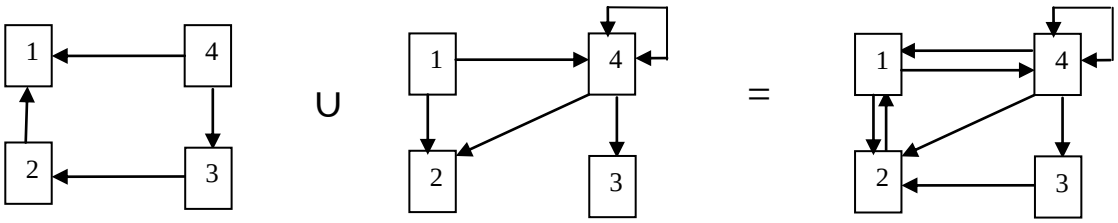


Fig.6 : Exemple d'union deux graphes.

6. Intersection :

Soient les graphes G1 = (X , U1) et G2 = (X , U2) :

G1 ∩ G2 = (X , U1) ∩ (X , U2) = (X , U1∩ U2).

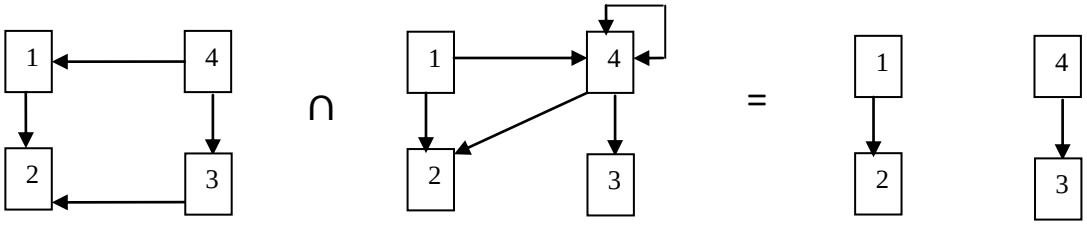


Fig.7 : Exemple d'intersection deux graphes.

7. Fermeture transitive :

Soit G =(X , U) un graphe orienté, la fermeture transitive G_f = (X , U_f) de G est définie par :

(x,y) ∈ U_f si et seulement si il existe un chemin (x_1,……x_k) de G avec x = x_1 et y = x_k.

Fig.8 : Fermeture transitive.

8. Graphe complémentaire :

On dit que le graphe G1=(X,U1) est le graphe complémentaire de graphe G si :

(x,y) ∉ U ⇔ (x,y) ∈ U1.[7]

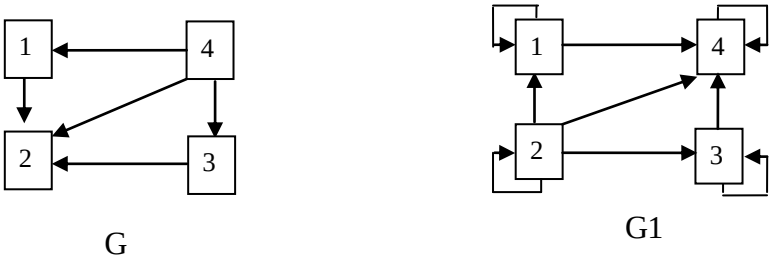


Fig.9 : Exemple du graphe complémentaire.

9. Graphe inverse :

On dit que le graphe G1=(X,U1) est le graphe inverse de graphe G si :

(x,y) ∈ U ⇔ (y,x) ∈U1.

Fig.10 : Exemple du graphe inverse.

Annexe 2

Manuel de références

ajout_arc :

graph ajout_arc(graph *L, int n, int i, int j).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet source.

j : sommet destinataire.

Description : cette fonction permet d'ajouter un arc entre deux sommets dans un graphe.

ajout_som :

graph ajout_ som(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : qu'on veut l'ajouter.

Description : cette fonction permet d'ajouter un sommet dans un graphe tel que cette sommet n'existe pas dans ce graphe.

arborescenc :

void arborescenc(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est un arborescence ou non.

arbre :

void arbre(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est un arbre ou non.

Bellman:

tab *Bellman(graph *L, int n, int i, int j, int& exit, int& vale).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

exit : si exit =1 alors il existe un chemin entre i et j sinon il n'existe pas.

vale : déterminer le coût de chemin entre i et j si il existe.

Description : cette fonction permet de trouver le plus court chemin et sa longueur entre deux sommets dans un graphe valué.

Bellman2:

tab *Bellman2(graph *L,int n, int i,int j,int& exit,int& vale).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

exit : si exit =1 alors il existe un chemin entre i et j sinon il n'existe pas.

vale : déterminer le coût de chemin entre i et j si il existe.

Description : cette fonction permet de trouver le plus long chemin et sa longueur entre deux sommets dans un graphe valué.

chain (chemin, cycle, circuit) :

void chain(chemin, cycl, circuit)(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si une suite de sommets est une chaîne (chemin, cycle, circuit) ou non.

chain_elem(chemin_elem, cycl_elem, circuit_elem) :

void chain_elem(chemin_elem, cycl_elem, circuit_elem)(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si une suite de sommets est une chaîne (chemin, cycle, circuit) élémentaire ou non.

chain_simp(chemin_simp, cycl_simp, circuit_simp) :

void chain_simp(chemin_simp, cycl_simp, circuit_simp)(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si une suite de sommets est une chaîne (chemin, cycle, circuit) simple ou non.

chain_hamil(chemin_hamil, cycl_hamil, circuit_hamil) :

void chain_hamil(chemin_hamil, cycl_hamil, circuit_hamil)(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si une suite de sommets est une chaîne (chemin, cycle, circuit) hamiltonienne ou non.

chain_euler, chemin_euler, cycl_euler, circuit_euler) :

void chain_euler, chemin_euler, cycl_euler, circuit_euler)(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si une suite de sommets est une chaîne (chemin, cycle, circuit) eulérienne ou non.

compos_fort_connex :

void graph_fortem_connex (graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de trouver tous les composants fortement connexes.

degr :

void degr(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut trouver son degré.

Description : permet de calculer la somme de demi-degré extérieur et le demi-degré intérieur.

demi_deg_ext :

void demi_deg_ext(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut trouver son demi degré extérieur.

Description : cette fonction permet de calculer et d'afficher le demi degré extérieur d'un sommet.

demi_deg_int :

void demi_deg_int(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : est sommet qu'on veut trouver son demi degré intérieur.

Description : cette fonction permet de calculer et d'afficher le demi degré intérieur d'un sommet.

Dijkstra :

tab * Dijkstra (graph *L,int n, int i, int j, int& exit, int& vale).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

exit : si exit =1 alors il existe un chemin entre i et j sinon il n'existe pas.

vale : déterminer le coût de chemin entre i et j si il existe.

Description : cette fonction permet de trouver le plus court chemin et sa longueur entre deux sommets dans un graphe valué

distanc :

void distanc(graph *L, int n, int i, int j).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

Description : cette fonction permet de calculer la distance entre deux sommet dans un graphe non value.

ferm_trans :

graph *ferm_trans(graph *L, int n);

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de trouver le fermeture transitive d'un graphe..

Floyd :

tab * Floyd(graph *L,int n, int i, int j, int& exit, int& vale).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

exit : si exit =1 alors il existe un chemin entre i et j sinon il n'existe pas.

vale : déterminer le coût de chemin entre i et j si il existe.

Description : cette fonction permet de trouver le plus court chemin et sa longueur entre deux sommets dans un graphe valué.

Ford_Fulk1 :

sommet *Ford_Fulk1(graph *L, int n, int& res, int& flot).

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

res : si res =1 alors le graphe est un réseau.

flot : flot maximum.

Description : cette fonction permet de rechercher un flot maximum dans un réseau de transport.

Ford_Fulk2 :

sommet *Ford_Fulk2(graph *L, int n, int& res, int& flot).

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

res : si res =1 alors le graphe est réseau .

flot : lflot maximum.

Description : cette fonction permet de rechercher un flot maximum dans un réseau de transport.

graph_asym :

void graph_ asym(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est asymétrique ou non.

graph_comp :

graph *graph_ comp(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

Description : cette fonction permet de trouver le complémentaire d'un graphe.

graph_complet :

void graph_complet(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est complet ou non.

graph_connex :

void graph_connex(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est connexe ou non.

graph_eulerien :

void graph_eulerien(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est eulérien ou non.

graph_fortem_connex :

void graph_fortem_connex(graph *L, int n).

Paramètres :

L : pointeur pour parcours le graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est fortement connexe ou non.

graph_hamilton :

point d'entrer du graphe.

Paramètres :

L : pointeur pour parcours le graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est hamiltonien ou non.

graph_invers :

graph *graph_ invers(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de trouver l'inverse d'un graphe.

graph_irreflex :

void graph_ irreflex(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est irréflexif ou non.

graph_nul :

void graph_nul(int n).

Paramètres :

n : nombre de sommets le graphe.

Description : cette fonction permet de tester si un graphe est nul ou non.

graph_reflex :

void graph_ reflex(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est réflexif ou non.

graph_reg :

void graph_reg(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est régulier.

graph_sem_eulerien :

void graph_sem_eulerien(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est semi eulérien ou non.

graph_sym:

void graph_ sym(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est symétrique ou non.

graph_triv :

void graph_triv(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est trivial.

graph_vide :

void graph_vide(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de tester si un graphe est vide.

inters_gra

graph *inters_gra (graph *L, graph *L1, int n)

Paramètres :

L : point d'entrer du graphe 1.

L1 : point d'entrer du graphe 2.

n : nombre de sommets de graphe.

Description : cette fonction permet de trouver l'intersection entre deux graphes.

Kruskal :

graph *Kruskal(graph *L, int n, int& z);

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

z : coût de l' arbre recouvrant minimum.

Description : cette fonction cherche un arbre recouvrant minimum d'un graphe.

Marihass:

tab * Marihass(graph *L,int n, int i, int j, int& exit, int& vale).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet de départ.

j : sommet d'arrivé.

exit : si exit =1 alors il existe un chemin entre i et j sinon il n'existe pas.

vale : déterminer le coût de chemin entre i et j si il existe.

j : le sommet d'arrivé (entier).

Description : cette fonction permet de trouver le plus court chemin et sa longueur entre deux sommets dans un graphe valué.

matric_adj :

void matric_adj(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet d'afficher un graphe.

parcour_largeur :

void parcour_largeur(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet de parcourir un graphe en largeur.

parcour_profond :

void parcour_profond (graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

Description : cette fonction permet de parcourir un graphe en profondeur.

pred_som :

void pred_som(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut trouver son propre prédécesseurs.

Description : cette fonction permet de trouver et d'afficher les prédécesseurs d'un sommet.

Prim :

graph *Prim(graph *L, int n, int& z);

Paramètres :

L : point d'entrer du graphe.

n : le nombre de sommets de graphe.

z : coût du arbre recouvrant minimum.

Description : cette fonction cherche un arbre recouvrant minimum d'un graphe.

Roy_Bus :

graph *Roy_Bus(graph *L1,sommet *L2,int n, int& res, int& flot, int& dis);

Paramètres :

L1 : point d'entrer du graphe (contient les capacités).

L2 : point d'entrer du graphe (contient les coûts).

n : le nombre de sommets de graphe.

res : si res =1 alors le graphe est réseau

flot : flot maximum.

dis : coût maximum.

Description : cette fonction permet de chercher un flot maximal de cout minimal.

saisir :

graph saisir(int n).

Paramètres :

n : nombre de sommets le graphe.

Description : cette fonction permet de créer un graphe.

som_isol :

void som_isol(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le tester.

Description : cette fonction teste si un sommet est isolé ou non.

som_pair :

void som_pair(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le tester.

Description : cette fonction teste si un sommet est pair ou non.

som_pend :

void som_pend (graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le tester.

Description : cette fonction teste si un sommet est pend ou non.

som_puit :

void som_puit(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le tester.

Description : cette fonction teste si un sommet puits ou non.

som_sourc :

void som_sourc(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le tester.

Description : cette fonction tester si un sommet source ou non.

suc_som :

void suc_som(graph *L, int n, int i).

Paramètres :

L : pointeur pour parcours le graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le trouver son propre successeurs.

Description : cette fonction permet de trouver et d'afficher les successeurs d'un sommet.

supprim_arc :

graph supprim_arc(graph *L, int n, int i, int j)

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet source.

j : sommet destinataire.

Description : cette fonction permet de supprimer un arc entre deux sommets dans un graphe.

supprim_som :

graph supprim_som(graph *L, int n, int i).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

i : sommet qu'on veut le supprimer.

Description : cette fonction permet de supprimer un sommet dans un graphe tel que cette sommet il est existe dans ce graphe.

tri_topologique :

void tri_topologique(graph *L, int n).

Paramètres :

L : point d'entrer du graphe.

n : nombre de sommets de graphe.

Description : cette fonction permet d'ordonner un graphe dans un ordre tri topologique**.**

union_gra :

graph *union_gra (graph *L, graph *L1, int n).

Paramètres :

L : point d'entrer du graphe 1.

L2 : point d'entrer du graphe 2.

n : nombre de sommets de graphe.

Description : cette fonction permet de trouver l'union entre deux graphes.

Glossaire

Adjacent : deux sommets sont adjacents s'ils sont reliés par une arête.

Arbre couvrant minimum : est un arbre couvrant dont la somme des poids des arêtes le constitue, est minimale.

Arbre couvrant : graphe partiel d'arbre.

Arc : est un lien entre deux sommets. L'arc (i, j) est caractérisé par un sommet initial i et un sommet terminal j.

Arête multiple : lorsque plusieurs arêtes relient deux sommets, on les appelle arêtes multiples.

Boucle : arrête qui rejoint un sommet avec lui-même.

Circuit absorbant : circuit a poids négative.

Degré entrant : synonyme de demi-degré intérieur.

Degré sortant : synonyme de demi-degré extérieur.

Digraphe : synonyme de graphe orienté.

Édition des liens : permet de regrouper les fichiers .o et les bibliothèques pour former un exécutable ou une nouvelle bibliothèque.

Graphe acyclique : graphe ne contenant pas de cycle.

Graphe nul : graphe sans sommet et sans arête.

Graphe partiel : graphe obtenu en enlevant des arêtes du graphe G.

Graphe pondéré : voir graphe valué.

Graphe simple : graphe ne contient ni boucle ni arêtes multiples.

Graphe trivial : graphe composé d'un seul sommet et d'aucune arête.

Graphe valué : graphe, orienté ou non, dont les arêtes (ou les arcs) possèdent un poids.

Graphe vide : graphe sans arête.

Heuristique : méthode de résolution de problèmes, non fondée sur un modèle formel et qui n'aboutit pas nécessairement à une solution.

Incident : un sommet est incident à une arête s'il est situé à une des deux extrémités de cette arête. Inversement, une arête est incidente à un sommet si elle "touche" ce sommet.

Longueur du chemin: nombre des arrêtes composant le chemin.

Multigraphe : un graphe doté d'une ou plusieurs arêtes multiples.

Nœud : voir sommet.

Ordre : nombre de sommets du graphe

Planaire : un graphe est planaire si on peut le dessiner dans un plan sans croiser deux arêtes.

Racine : Une racine, dans un graphe orienté, est un sommet r à partir duquel on peut atteindre tous les autres sommets du graphe orienté. On dit aussi que tout autre sommet du

Sommet isolé : est un sommet de degré zéro.

Sommet pair : sommet de degré pair.

Sommet pendant : sommet de degré un.

Sommet puits : sommet de degré extérieur égal à zéro.

Sommet source : sommet de degré intérieur égal à zéro.

Sommet: Point dans un graphe ayant un nom unique.

Sous graphe : un sous-graphe d'un graphe est obtenu en y enlevant des sommets et toutes les arêtes incidentes à ces sommets.

Voisin : voir adjacent.

Webographies

[S1]http://www.ufrsciencestech.ubourgogne.fr/ .

[S2] http://*www.futura-sciences.com/*

[S3] http:// *www.universalis.fr/.*

[S4] http://www.dictionnaire.sensagent.com/

[S5] http://www.graal.ens-lyon.fr/~fvivien/Enseignement/Algo-2001-2002/

[S6] http:// www.techniques-ingenieur.fr/

[S7] http://www.nawouak.net/.

[S8] http://www.apprendre-en-ligne.net/graphes/

[S9] http://eric.univ-lyon2.fr/.

Bibliographies

[Did.03] Didier Maquin,Eléments de Théorie des Graphes, Institut National Polytechnique de Lorraine école Nationale Supérieure d'Electricité et de Mécanique 2003.

[Sig.02] Eric Sigward, introduction à la théorie des graphes 2002.

[Mar.03] Philppe Laccomme – christian Prins –Marc Sevaux, Algorithmes de graphes 2003.

[Lab.81]Jacques Labelle, théorie des graphes 1981.

[Lop.05] Pierre Lopez LAAS–CNRS, Cours de GRAPHES 2005.