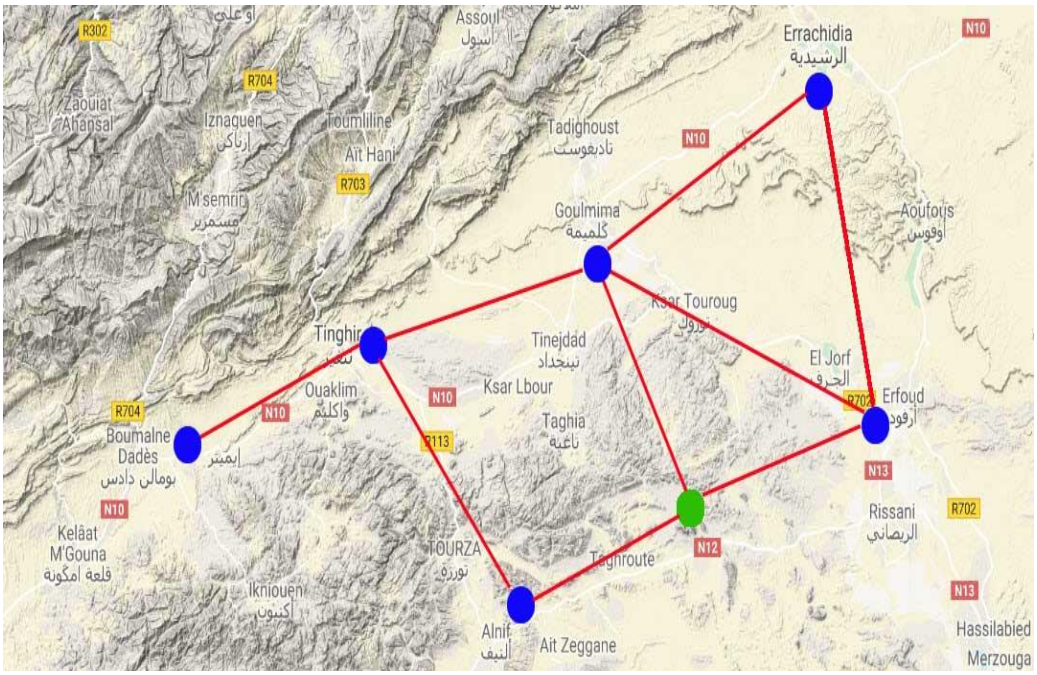
Université Moulay Ismail
Faculté des Sciences et Techniques Errachidia
Département Informatique
Master MST SIDI
2018/2019



Théorie des graphes

IMPLEMENTAION EN LANGAGE C

Hassan Zekkouri | Théorie des graphes et technique d'optimisations | 11 mars 2019
Responsable : Prof. M. Driss Aouragh

Table des matières

INTRODUCTION

La théorie des graphes et les algorithmes qui lui sont liées est un des outils privilégiés de modélisation et de résolution des problèmes dans un grand nombre de domaines allant de la science fondamentale aux applications technologiques concrètes. On peut citer les réseaux électriques et transport d'énergie, le routage du trafic dans les réseaux de télécommunications et les réseaux d'ordinateurs, le routage du trafic de véhicules et l'organisation des tournées ou rotations, les problèmes de localisation (d'entrepôts, d'antennes) et de déplacements, les problèmes d'ordonnancement de tâches et d'affectation des ressources.

Dans ce rapport, nous allons faire une implémentation d'une partie des algorithmes de théorie des graphes et ceci en utilisant le langage C comme langage de programmation.

Noter que les implémentations sont faites en langage C sous forme d'un projet Code Blocks.

Avant-propos

Dans la partie du code « Projets CodeBlocks » les entrées sont regroupées dans des fichier texte et la lecture est faite avec le fonction « freopen(nomFichier, modeLecture, stdin) », donc pour tester avec d'autre graphe il suffira de changer les données de ces fichiers textes (matrices d'adjacences, suites des liens, ...).

Certains graphique et vérification sont fait grâce au logiciel Grin 4.0 qui sera dans le CD.

Chapitre 1 : Représentations machine

On peut représenter un graphe sur machine avec plusieurs structures de données dont on peut citer les suivantes.

- Matrice d'adjacence ;
- Matrice d'incidence ;
- Listes d'adjacence ;
- Les tableaux LS et PS ;
- Les tableaux LS et APS ;
- Représentation avec les liaisons.

Chaque représentation présente des avantages et des inconvénients, et leurs utilisations dépendent de problème à résoudre.

(L'implémentation : Mini Projet 1)

1) MATRICE D'ADJACENCE

A tout graphe d'ordre |X| = n, on associe une matrice M de n lignes et n colonnes dont les éléments sont notés M_{ij} :

- Pour un graphe non orienté G = (X, E) :

$$M_{ij} = \begin{cases} 1 & si \ \{x_i, x_j\} \in E \\ 0 & sinon \end{cases}$$

Degré d'un sommet s : $d(s) = \sum M_{sj}$

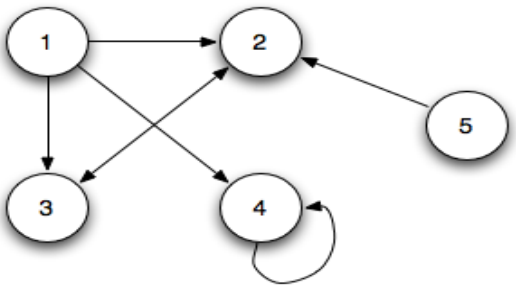
- Pour un graphe orienté G = (X, U) :

$$M_{ij} = \begin{cases} 1 & si \ (x_i, x_j) \in U \\ 0 & sinon \end{cases}$$

Degré d'un sommet s : $d(s) = d^+(s) + d^-(s)$

Avec $d^+(s) = \sum M_{sj}$ et $d^-(s) = \sum M_{is}$

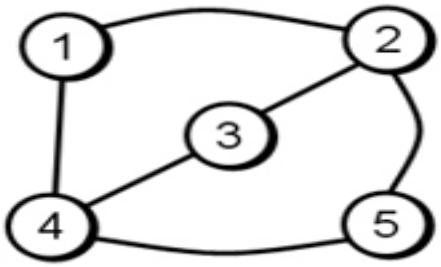
- Exemples des tests :
  - Graphe Orienté :



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

Figure 1.1.1

❖ Graphe Non Orienté :



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

Figure 1.1.2

Application :

```
Voici la matrice d'adjacence de votre graphe:
0       1       1       1       0
0       0       1       0       0
0       1       0       0       0
0       0       0       1       0
0       1       0       0       0


Voici la Liste des arcs :
1 --> 2,        1 --> 3,        1 --> 4,
2 --> 3,
3 --> 2,
4 --> 4,
5 --> 2,
```

```
Voici la matrice d'adjacence de votre graphe:
0       1       0       1       0
1       0       1       0       1
0       1       0       1       0
1       0       1       0       1
0       1       0       1       0


Voici la Liste des aretes :

2 ---- 1,
3 ---- 2,
4 ---- 1,       4 ---- 3,
5 ---- 2,       5 ---- 4,
```

2) MATRICE D'INCIDENCE

La matrice d'incidence est une matrice $n \times p$, où n est le nombre de sommets du graphe et p est le nombre de liens (arêtes ou arcs).

Pour les graphes sans boucle on a les deux cas suivants :

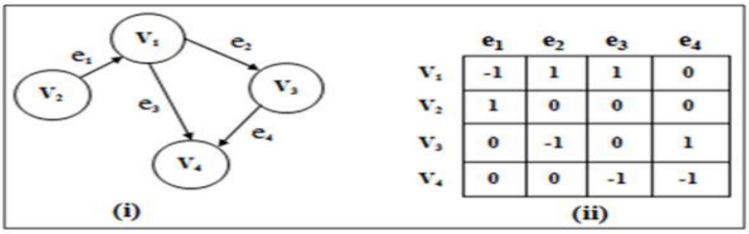
➢ Graphe Orienté G = (X, U) :

$$M_{ij} = \begin{cases} -1 & si\ l'arc\ u_j\ sort\ du\ sommet\ x_i \\ 1 & si\ l'arc\ u_j\ entre\ dans\ le\ sommet\ x_i \\ 0 & sinon \end{cases}$$

➢ Graphe Non Orienté G = (X, E) :

$$M_{ij} = \begin{cases} 1\ si\ le\ sommet\ x_i est\ une\ extrémité\ de\ l'arête\ e_j \\ 0\ \ sinon \end{cases}$$

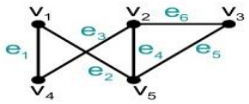
- Exemples de teste :
  - ❖ Graphe Orienté :



	e1	e2	e3	e4
V1	-1	1	1	0
V2	1	0	0	0
V3	0	-1	0	1
V4	0	0	-1	-1

Figure 1.2.1

❖ Graphe Non Orienté :



$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Figure 1.2.2

Application :

```
Voici la matrice d'incidence de votre graphe:
-1      1       1       0
1       0       0       0
0       -1      0       1
0       0       -1      -1


Voici la Liste des arcs :
2 ----> 1
1 ----> 3
1 ----> 4
3 ----> 4
```

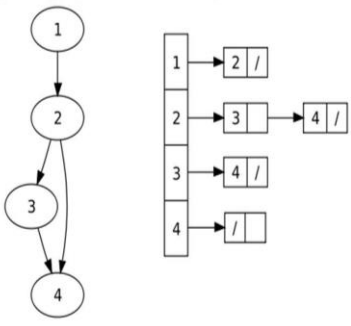
```
Voici la matrice d'incidence de votre graphe:
1       1       0       0       0       0
0       0       1       1       0       1
0       0       0       0       1       1
1       0       1       0       0       0
0       1       0       1       1       0


Voici la Liste des aretes :
1 ---- 4
1 ---- 5
2 ---- 4
2 ---- 5
3 ---- 5
2 ---- 3
```

3) LISTES D'ADJACENCE

On peut représenter un 1-graphe en donnant pour chacun de ses sommets, la liste des sommets qu'on peut atteindre directement en suivant un arc (dans le sens de la flèche). Dans le cas de non orienté on peut donner à chacun de ses sommets la liste des sommets auxquels il est adjacent. Ce sont les listes d'adjacences.

- ➢ Graphe Orienté :
  x_i : [*liste de sommets* qu'on peut atteindre directement en suivant un arc.]
- ➢ Graphe Non Orienté :
  x_i : [liste des sommets adjacent.]

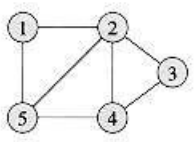
- o Exemples et testes :
  - ❖ Graphe Orienté :

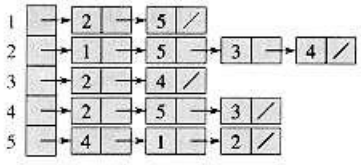


$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 1.3.1

❖ Graphe Non Orienté :





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

Figure 1.3.2

Application :

```
---------- Voici vos listes d'adjacences
1 :  2,
2 :  3, 4,
3 :  4,
4 :

Voici la matrice d'adjacence de votre graphe:
0       1       0       0
0       0       1       1
0       0       0       1
0       0       0       0
```

```
---------- Voici vos listes d'adjacences
1 :  2, 5,
2 :  1, 5, 3, 4,
3 :  2, 4,
4 :  2, 5, 3,
5 :  4, 1, 2,

Voici la matrice d'adjacence de votre graphe:
0       1       0       0       1
1       0       1       1       1
0       1       0       1       0
0       1       1       0       1
1       1       0       1       0
```

4) LES TABLEAUX LS ET PS

A tout graphe orienté G = (X, U) avec |X| = n, |U| = m et on peut associer deux tableaux (vecteurs) PS et LS.

- PS : Tableau de pointeurs (‘Position’) à n+1 éléments où PS[i] pointe sur la case contenant le premier successeur du sommet i dans LS.
- LS : La liste de tous les sommets vus comme successeur d’un autre sommet. C’est un tableau de m éléments où les successeurs d’un sommet i se trouvent entre la case numéro PS[i] et la case PS [i+1]-1 du tableau LS.

➢ Règles : On pose

- PS [0] = 1 ;
- Pour tout $2 \leq i \leq n$:

$$PS[i] = PS[i-1] + |nombre\ des\ successeur\ de\ i-1| \ ;$$

- PS[n] = m+1 ; sommet virtuel pour achever les calculs de nombres des successeurs.
- Si un sommet i n’a pas de successeurs, on aura PS [i+1] = PS[i], c’est-à-dire on lui associer la même valeur que le sommet suivant.

- Exemples et testes :

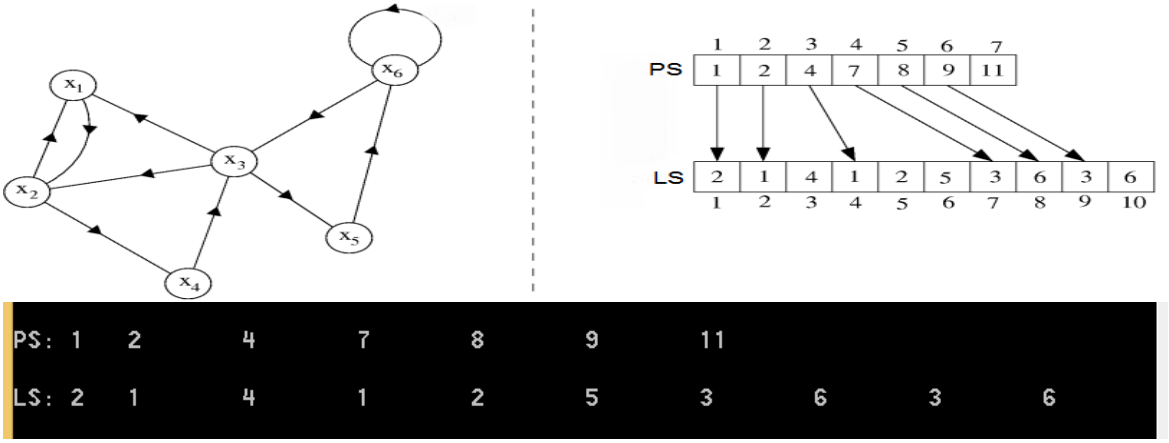


Figure 1.4.1

- Remarque :
  On a les formules suivantes :
  - ✓ Graphe Orienté :
    PS[s+1] - PS[s] = Nombre de successeurs de sommet s = $d^+(s)$;
  - ✓ Graphe Orienté :
    PS[s+1] - PS[s] = Nombre de successeurs de sommet s = $d(s)$.
  - ✓ Les successeurs d'un sommet s se trouvent entre la case numéro PS[s] et la case numéro PS[s+1]-1 du tableau LS.

5) LES TABLEAUX LS ET APS

- Principe :

Soit G = (X, U) un 1-graphe orienté tel que |X| = n, |U| = m. On définit :

- FS : La File de Successeurs. Un tableau linéaire contenant successivement la liste de successeurs.
- APS : Tableau indicé par n° de sommet et donnant l'adresse du premier successeur ; APS[x] (Adresse de Premier Successeur de x).
- Même règles déjà vu avec les tableaux PS et LS.

- Remarque :
  On a les formules suivantes :
  - ✓ Graphe Orienté :
    APS[s+1] - APS[s] = Nombre de successeurs de sommet s = $d^+(s)$;
  - ✓ Graphe Orienté :
    APS[s+1] - APS[s] = Nombre de successeurs de sommet s = $d(s)$.

Application : Graphe page 26

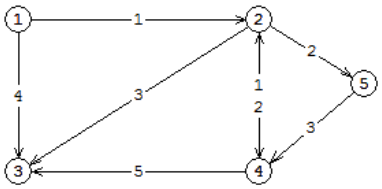
```
APS: 1   3       5       7

FS: 1   3       1       3       1       2
```

6) REPÉSENTATION AVEC DES LIAISONS

On peut encore faire mieux et représenter un graphe avec une structure de trois champs :

- Nombre de sommets ;
- Nombre de liaisons (arcs ou arêtes) ;
- Un tableau de la structure Arc qui est composé de trois champs :
    - Extrémité initiale
    - Extrémité terminale
    - Poids.

Application :



Information about the Edges

Edge	Name	Type	Weight	Flow
(1,2)	No Name	Line	1	0
(1,3)	No Name	Line	4	0
(2,3)	No Name	Line	3	0
(2,4)	No Name	Line	2	0
(2,5)	No Name	Line	2	0
(4,2)	No Name	Line	1	0
(4,3)	No Name	Line	5	0
(5,4)	No Name	Line	3	0

Procedure complete.

```
Voici la liste des liaisons de votre graphe:
0 ---(1)--- 1
0 ---(4)--- 2
1 ---(3)--- 2
1 ---(2)--- 3
1 ---(2)--- 4
3 ---(5)--- 2
3 ---(1)--- 1
4 ---(3)--- 3
```

Conclusion : On peut faire plusieurs représentations machines pour un graphe est chaque représentation présente ses avantages pour tirer certain information à partir d'un graphe. On n'a pas fait toutes les implémentations permettant de tirer ses informations pour ne pas rendre le rapport trop volumineux et ça reste à faire en avenir comme poursuite de ce modeste travail.

Chapitre 2 : Exploration d'un graphe

Dans la théorie des graphes l'exploration d'un graphe peut être conçue comme une promenade le long des arcs/arêtes au cours de laquelle on visite les sommets.

Les parcours d'un graphe servent de base à bon nombre d'algorithmes. Le plus souvent, un parcours de graphe est un outil pour étudier une propreté globale du graphe :

- Connexité ;
- Partition (graphe biparti) ;
- Connexité forte ;
- Etc…

En générale, on a deux types d'exploration :

- Parcours en largeur BFS (Breadth First Search);
- Parcours en profondeur DFS (Depth First Search).

(L'implémentation : Mini Projet 2)

1) PARCOURS EN LARGEUR

Dans ce type de parcours, on utilise une file (premier entré, premier sortie) FIFO.

Algorithme 1 : (Utilisé ici pour l'affichage Mini Projet 2)

```
Procédure largeur (G, r)
      Variable n = order(G)
               marque: Tableau
               file: File
      Debut
            Pour i = 1 à n faire
                  marque[i] = 0
            Finpour
            tête = 1
            queue = 1
            enfiler (file, r) // et afficher r
            marque[r] = 1
            Tant que (tête <= queue) faire
                  x = défiler (file)
                  Pour y successeur de x faire
                        Si (marque[y] = 0) faire
                              marque[y] = marque[x] + 1
                              queue = queue + 1
                              enfiler (file, y) //afficher y
                        Finsi
                  Finpour
                  tête = tête + 1
            finTQ
      Fin
```

On utilise une file de taille n+1, où n = |X|.

- On associer à chaque sommet un numéro ;
- La racine r aura le numéro 1 ;
- Les sommets marqués à partir d'une couche C_i auront le numéro i+1.

Algorithme 2 : (Utilisé ici pour stocké les résultats pour une autre exploitation)

Lors de parcours on matérialise l'état d'un nœud par une couleur :

- Blanc : sommet non découvert
- Gris : sommet découvert
- Noir : sommet dont tous les successeurs sont découverts

```
Procédure BFS (G, s)
        Variable n = order(G)
                 marque: Tableau de type structuré
                 file: File
        Debut
               Pour i = 1 à n faire
                      marque[i].couleur = blanc
                      marque[i].père = null
                      marque[i].dist = infini
               Finpour
               marque[s].couleur = gris
               marque[s].dist = 0
               enfiler (file, s)
               Tant que (fileNonVide(file)) faire
                      u = défiler (pile)
                      Pour tout v successeur de u faire
                             Si (marque[v] = blanc) faire
                                    marque[v].couleur = gris
                                    marque[v].père = u
                                    marque[v].dist = marque[u].dist + 1
                                    enfiler (file, v)
                             Finsi
                      Finpour
                      défiler(file, u)
                      marque[v].couleur = noir
               finTQ
        Fin
```

Cette algorithme est utilisé dans la partie de vérification « Graphe biparti ou non ! ».

2) PARCOURS EN PROFONDEUR

Le parcours en profondeur consiste à :

- Choisir un sommet de départ s et le marquer comme découvert (gris) ;
- Suivre un chemin issu de s aussi loin que possible en marquant les sommets au fur et à mesure qu'on les découvre ;
- En fin du chemin (marquer en noir) revenir au dernier choix fait est prendre une autre direction.

On utilise une structure constituée de quatre champs : couleur, père, découverte et fin.

Algorithme 1 : (Cette algorithme prend le chemin selon le premier successeur rencontré)

```
Procédure BFS (G, s)
	Variable n = order(G)
		  marque: Tableau de type structuré
	Debut
		Pour i = 1 à n faire
			marque[i].couleur = blanc
			marque[i].père = null
		Finpour
		temps = 0
		pour tout u sommet de G faire
			si (marque[u].couleur = blanc) faire
				marque[u].couleur = gris ;
				marque[u].découverte = temps ;
				temps = temps + 1 ;
				DFS_visite(G, u) ;
			Finsi
		Finpour
	Fin


Procédure BFS_visite (G, u)
	Debut
		Pour tout v successeur de u faire
			Si (marque[v].couleur = blanc) faire
				marque[v].couleur = gris ;
				marque[v].découverte = temps ;
				marque[v].père = u ;
				temps = temps + 1 ;
				DFS_visite(G, v) ;
			Finsi
		Finpour
		marque[u].fin = temps ;
		temps = temps + 1 ;
	Fin
```

Algorithme 2 :

La gestion des sommets est réalisé à l'aide une pile, une structure de donné basée sur le principe du dernier arrivé premier sortie LIFO. Ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Ici on va choisir le dernier sommet successeurs dans la liste comme sommet suivant car il sera dans la tête de la pile. (Dernier sommet à être empiler) !

Contrairement à l'algorithme 3 qu'on verra tout de suite, cet algorithme ne nécessite pas les deux tableaux PS et LS qui rend les choses un peu compliqué avec l'implémentation.

Ce qu'il faut remarquer est que le tableau « marque » garde le vrai ordre de parcours et la découverte des sommets et peut être utilisé pour la distinction des niveaux ou couches.

```
Procédure Profondeur (G, r)
	Variable n = order(G)
		  marque: Tableau
		  pile: Pile
	Debut
		Pour i = 1 à n faire
			marque[i] = 0
		Finpour
		 tête = 1
		 empiler (pile, r)
		 marque[r] = 1
		Tant que (tête > 0) faire
			x = dépiler (pile)
			Pour y successeur de x faire
				Si (marque[y] = 0) faire
					marque[y] = marque[x] + 1
					tête = tête + 1
					empiler (pile, y)
				Finsi
			Finpour
		finTQ
	Fin
```

Algorithme 3 : (N'est pas implémenter ici, mais sera implémenter en Python)

Ici la mise à jour de PS alterne les données, c'est mieux de le copier, ce qui complique l'algorithme.

```
Procédure Profondeur (G, r)
	Variable n = order(G)
		  marque, PS, LS : Tableau
		  pile: Pile
	Debut
		Pour i = 1 à n faire
			marque[i] = 0 ;
		Finpour
		 tête = 1 ;
		 empiler (pile, r) ;
		 marque[r] = 1 ;
		Tant que (tête > 0) faire
			x = dépiler (pile) ;
			Si (PS[x] != 0) faire
				y = LS[PS[x]]
				Si (marque[y] = 0) faire
					marque[y] = marque[x] + 1 ;
					tête = tête + 1 ;
					empiler (pile, y) ;
				Finsi
				Mettre à jours PS[x]
				(Pointer vers le successeur suivant non
				 Marqué de x, 0 si tous est marqué)
			Sinon
				tête = tête – 1 ;
			Finsi
		finTQ
	Fin
```

Chapitre 3 : Arbre couvrant de poids minimal

Dans les problèmes de minimisation des coûts on peut utiliser les graphes aux liaisons valuées et leur appliquer l'un des algorithmes de recherche de l'arbre couvrant de poids minimal à savoir :

- Algorithme de Kruskal ;
- Algorithme de Prim ;

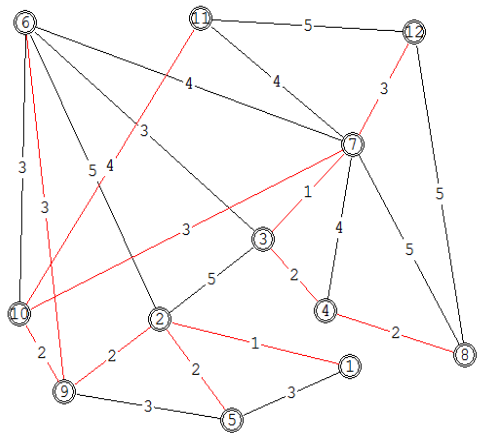
(L'implémentation : Mini Projet 3)

1) ALGORITHME DE KRUSKAL

On construit un sous-graphe en ajoutant des arêtes une par une. A chaque étape, on cherche l'arête de plus petite valuation parmi celles que l'on n'a pas déjà exploitées. Si elle ne crée pas de cycle, on l'ajoute au sous-graphe, sinon on la laisse de côté. On termine dès l'on a sélectionné n-1 arêtes.

```
Kruskal (G)
Variable
          F : les arêtes de l'arbre couvrant
          X : les arêtes du graphe
Debut
       F = vide
       Trier X en ordre croissant de poids
       Pour a dans X faire
              Si Union (F, a) est acyclique alors
                     F = Union (F, a)
              Finsi
       Finpour
Fin
```

Application :




```
          Min. Spanning Tree
          Time : 00:57:11
          Date : 11/03/2019
       NetWork : KRUSKAL
          Type : undirNet
Number of Points : 12
Number of Edges  : 23

-----------------------------------------

Min. Spanning Tree Weight = 25
 Edges of Min. Spanning Tree:
  ( 1,  2),  ( 2,  9),  ( 9, 10),  ( 2,  5),  ( 10,  7),
  ( 7,  3),  ( 3,  4),  ( 4,  8),  ( 7, 12),  ( 9,  6),
  ( 10, 11).
Procedure complete.
```

Résultats de notre programme : le graphe est stocké sous forme d'arêtes (kruskal.txt)

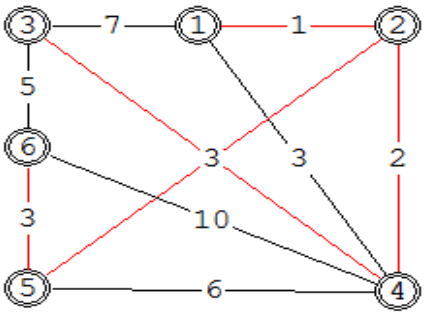
```
Voici les arcs (aretes) de l'arbre de poids minimal = 25
1 --(1)-- 2
3 --(1)-- 7
2 --(2)-- 5
2 --(2)-- 9
9 --(2)-- 10
3 --(2)-- 4
4 --(2)-- 8
9 --(3)-- 6
3 --(3)-- 6
7 --(3)-- 12
7 --(4)-- 11
```

2) ALGORITHME DE PRIM

On construit un sous-graphe en ajoutant des arêtes une par une. A chaque étape, on cherche l'arête sortante de plus petite valuation parmi celles que l'on n'a pas déjà exploitées. Une arête est sortante si elle joint un sommet du sous-graphe à un sommet qui n'est pas dans le sous-graphe. On termine dès l'on a sélectionné n-1 arêtes.

```
Prim (G, x0)
Variable
        F : les arêtes de l'arbre couvrant
        X : les arêtes du graphe
        M : les sommets marqués
Debut
      F = vide
      M = {x0}
      Tant qu'il y a des arêtes sortant de M faire
            Chercher l'arête a=(x,y) de plus petit
            Valuation tel que x∈ M et y∉ M
            M = Union(M, y)
            F = Union(F, a)
      FinTQ
Fin
```

Application :




```
          Min. Spanning Tree
          Time : 01:15:59
          Date : 11/03/2019
       NetWork : Prim
          Type : undirNet
Number of Points : 6
Number of Edges  : 10

-------------------------------------------

Min. Spanning Tree Weight = 13
 Edges of Min. Spanning Tree:
  ( 1,  2),  ( 2,  4),  ( 4,  3),  ( 2,  5),  ( 5,  6).
Procedure complete.
```

Résultats de notre programme : le graphe est stocké sous forme d'arêtes (kruskal.txt)

```
Voici les arcs (aretes) de l'arbre de poids minimal = 13
1 --(1)-- 2
2 --(2)-- 4
4 --(3)-- 3
1 --(3)-- 4
2 --(4)-- 5
```

REMARQUE :

Pour l'arbre couvrant de poids minimal, le choix entre deux liaisons de poids égaux dépend de l'ordre de saisie !

Chapitre 4 : Plus court chemin

Dans cette partie on s'intéresse à des algorithmes qui cherchent le plus court chemin entre un sommet x_0 et tous les autres sommets d'un graphe. Les résultats seront stockés de la façon suivante :

- Le tableau lambda λ , contient la longueur du plus court chemin à chaque sommet de graphe. $\lambda(i) = d(x_0, i)$
- Le tableau Pi πcontient les prédécesseurs ou les pères de chaque sommet dans le plus court chemin.

(L'implémentation : Mini Projet 4)

Principe générale :

- On initialise les tableaux λ et, Initialisation () ;
- On calcule les $\lambda(i)$ et $\pi(i)$ par approximations successives, ce qui signifie qu'à chaque étape, on essaye d'améliorer les valeurs obtenues précédemment.
- L'amélioration au niveau local se vérifie aussi ; pour un sommet x et un sommet y successeur de x, on compare la valeur λ(y) obtenue à l'étape précédente avec la valeur qu'on obtiendrait en passant par x, i.e. $\lambda(y)$ +v(x, y). C'est la technique de relâchement, Relâcher (G, i, j).

```
Algorithme 1 :
       Initialisation (G,x0)
       Variable
                 Lambda, Pi : Tableau
       Debut
              Pour i à n faire
                     Lambda[i] = infini
                     Pi[i] = null
              Finpour
              Lambda [x0] = 0
              Marquer x0
       Fin


Algorithme 2 :
       Relâcher (G, i, j)
       Variable
                 Lambda, Pi : Tableau
       Debut
              Si (Lambda[j] > Lambda[i] + v (i, j)) faire
                     Lambda[j] = Lambda[i] + v (i, j);
                     Pi[j] = i ;
              Finsi
              Lambda [x0] = 0 ;
              Marquer x0
       Fin
```

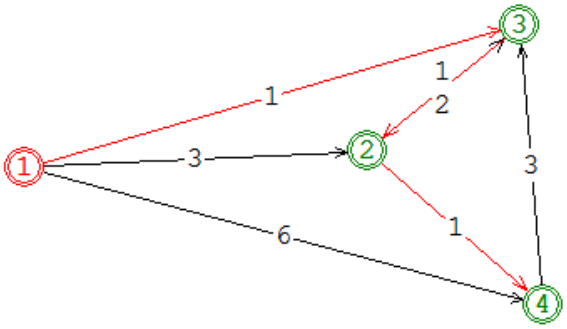
1) ALGORITHME DE DIJKSTRA

Cet algorithme s'applique aux graphes dont les valuations sont positives. A chaque étape un sommet i sera marqué (ses valeur $\lambda(i) et\ \pi(i)$ seront alors définitives), puis on utilisera la technique de relâchement afin d'améliorer les chemins manant aux successeurs de i. Le nouveau sommet marqué sera choisi comme celui dont la valeur du chemin depuis x_0 est minimale parmi tous les sommets non encore marqués.

```
Algorithme de Dijkstra :
        Dijkstra (G,x0)
        Variable
                E : ensemble de sommets marqués
        Début
                Initialisation (G,x0)
                E = vide
                Tant que E ≠ S faire
                        i = un sommet non marquer de λ(i) minimal
                        E = Union(E, i)
                        Pour tout j successeur de i faire
                                Si j n'est pas marqué alors
                                        Relâcher (G, i, j)
                                Finsi
                        Finpour
        Fin
```

Application : la matrice d'adjacence est dans le fichier dijkstra.txt

Considérons le graphe suivant : (sommet) longueur de plus court chemin




```
    All Shortest Path  from Source
           Time : 22:23:10
           Date : 10/03/2019
        NetWork : GRAPHE
           Type : dirNet
Number of Points : 4
Number of Edges  : 7

--------------------------------------------

Source = 1
The Lengths of the Shortest Paths from 1 to other Points :
 (  2) 2              (  3) 1              (  4) 3
Procedure complete.
```

Résultats de notre programme : on commence à zéro (1 devient 0 …)

```
Distances a partir de la source:

 Sommet          Distance        Pere
0                0               0
1                2               2
2                1               0
3                3               1
```

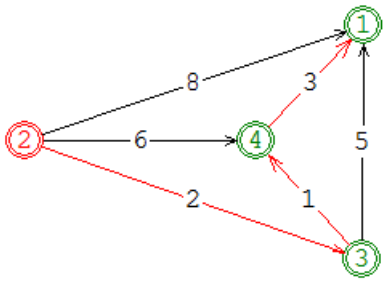
2) ALGORITHME DE BELLMAN

Cet algorithme s'applique aux graphes sans circuits et pour la recherche de plus court chemin d'un sommet x_0 **sans prédécesseurs** vers tous les autres sommets. A chaque étape, on trouve un plus court chemin pour un nouveau sommet en se basant sur les prédécesseurs qui sont déjà traité.

La première étape consistera à renuméroter les sommets de façon à ne jamais revenir en arrière, c'est le rôle de l'algorithme tri topologique d'un graphe.

```
Algorithme de triTopo :
        triTopo (G,x0)
        Numéroter 1 le sommet x0
        k = k + 1
        Tant que k < n faire
                Numéroter k un sommet sans prédécesseurs ou dont tous les
                prédecesseurs sont déjà marqués
                K = k + 1
        finTQ
Fin

Algorithme de Bellman :
        Dijkstra (G,x0)
        Variable
                n : ordre de G
        Début
                triTopo (G,x0)
                Initialisation (G,x0)
                Pour j = 2 à n faire
                        Pour tout i prédécesseur de j faire
                                Relâcher (G, i, j)
                        Finpour
                Pourpour
        Fin
```

Application : la matrice d'adjacence est dans le fichier bellman.txt




```
Report for NetWork Bellman     (GRIN 2000)

    All Shortest Path  from Source
            Time : 22:47:50
            Date : 10/03/2019
         NetWork : Bellman
            Type : dirNet
Number of Points : 4
Number of Edges  : 6

---------------------------------------------

Source = 2
The Lengths of the Shortest Paths from 2 to other Points :
 (  1) 6              (  3) 2              (  4) 3
Procedure complete.
```

Résultats de notre programme : on commence à zéro (1 devient 0 ...)

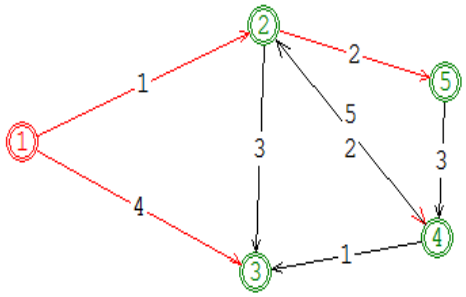
```
Distances a partir de la source:

 Sommet          Distance        Pere
0                6               3
1                0               0
2                2               1
3                3               2
```

3) ALGORITHM DE BELLMAN-FORD

Cet algorithme est applicable sur **tous les types de graphes**. Il détectra même s'il y a un cycle absorbant ou non. L'idée est de parcourir tous les sommets et d'esffectuer la technique de relâchement jusqu'à ce que les distances calculées soient stabilisées i.e. qu'on puisse plus les amélorer. Pour cela, nous verrons qu'au pire il faudra parcourir **n-1** fois tous les arcs.

```
Algorithme de Bellman-Ford :
        Bellman-Ford (G,x0)
        Variable
                n : ordre de G
        Début
                Initialisation (G,x0)
                Pour k = 1 à n-1 faire
                        Pour chaque arc (i, j) de G faire
                                Relâcher (G, i, j)
                        Finpour
                Pourpour
                Pour chaque arc (i, j) de G faire
                        Si (Lambda[j] > Lambda[i] + v (i, j)) alors
                                Retourner FAUX (cycle absoorbant) exit
                        Finsi
                Finpour
        Fin
```

Application :




```
Report for NetWork BELLMANFORD     (GRIN 2000)

    All Shortest Path  from Source
            Time : 23:39:01
            Date : 10/03/2019
         NetWork : BELLMANFORD
            Type : dirNet
Number of Points : 5
Number of Edges  : 8

------------------------------------------------

Source = 1
The Lengths of the Shortest Paths from 1 to other Points :
 (  2) 1              (  3) 4              (  4) 3
 (  5) 3
Procedure complete.
```

Résultats de notre programme :

Sommet	Distance	Pere
0	0	0
1	1	0
2	4	0
3	3	1
4	3	1

Chapitre 5 : Autres algorithmes

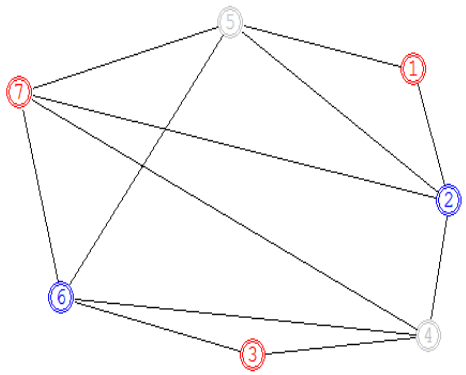
1) COLORIAGE

Colorier un graphe consiste à affecter une couleur à chacun de ses sommets de sorte que deux sommets adjacents ne soient pas de la même couleur. (L'implémentation : Mini Projet 3)

Algorithme de Powell-Welsh :

- **Ranger les sommets du plus haut degré au plus petit ;**
- **Choisissez un couleur pour le premier sommet ;**
- **Colorier tous les sommets non adjacents à cet sommet et qui ne sont pas adjacents entre eux ;**
- **Réitérer ce procédé avec une autre couleur pour le premier sommet non colorié de la liste rangé jusqu'à épuisement des sommets.**

Application :




```
          Chromatic Number
             Time : 00:12:47
             Date : 11/03/2019
          NetWork : NoName
             Type : undirNet
Number of Points : 7
Number of Edges  : 12

------------------------------------------------


Chromatic Number = 3
Minimal Point`s Coloration is :
  Point      1   2   3   4   5   6   7
  Color      1   2   1   3   3   2   1

Procedure complete.
```

Résultats de notre programme C :

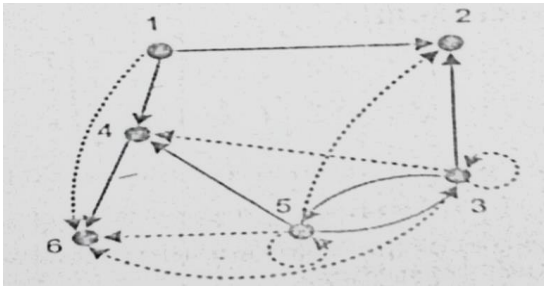
```
########## Debut de Coloriage ########

Sommet S(1) --> couleur 1
Sommet S(3) --> couleur 1
Sommet S(7) --> couleur 1

Sommet S(2) --> couleur 2
Sommet S(6) --> couleur 2

Sommet S(4) --> couleur 3
Sommet S(5) --> couleur 3

###### FIN de COLORIAGE ######
```

2) FERMETURE TRANSITIVE

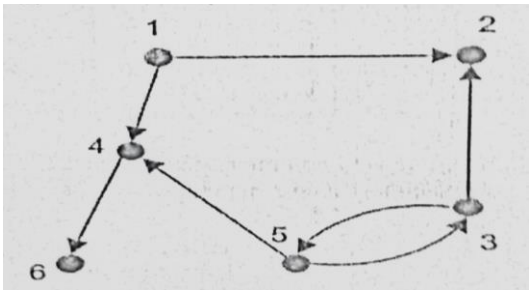
On appelle fermeture transitive d'un graphe G = (X, U), le graphe $G_f(X, U^f)$ tel que pour tout couple de sommets$(s_i, s_j) \in X^2$, l'arc/arête (s_i, s_j) appartient à U^f si et seulement s'il existe un chemin/chaine entre de s_i *vers* s_j . Son calcul peut se faire en additionnant les « puissances » successives de sa matrice d'adjacence ou utilisé l'algorithme de **Warshall**.

Algorithme de Warshall

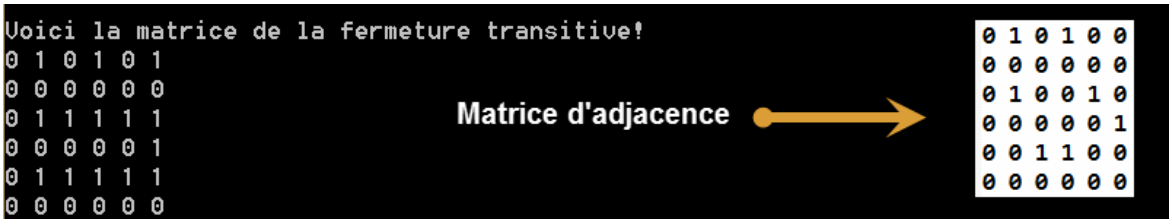
```
Warshall(G)
	Variables M = G.matrice
	Debut
		Pour i de 1 à n faire
			Pour j de 1 à n faire
				Si (M[j,i] == 1) alors
					Pour k de 1 à n faire
						Si (M[i,k] == 1) alors
							M[j,k] = 1
						Finsi
					Finpour
				Finsi
			Finpour
		Finpour
	Fin
```



Application : Graphe de la page 16



Résultat de notre programme ! (L'implémentation : Mini Projet 3)



3) COMPOSANTES FORTEMENT CONNEXES

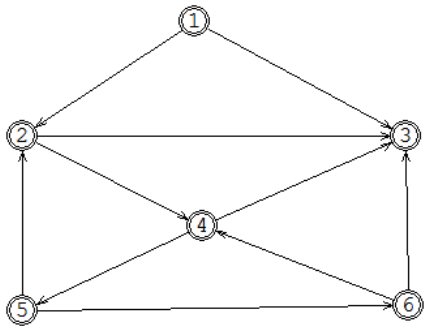
Les composantes fortement connexes d'un graphe orienté sont les sous-ensembles de sommets engendrant un sous-graphe fortement connexe.

Algorithme générale de recherche des composantes fortement connexes :

```
CFC (G(X, U))
    Variables H = X [sommets du graphe]
              C = ∅
              P : Pile
    Debut
        Tant que H ≠ ∅ faire
            Choisir un sommet s ∈ H
            Empiler(P, {s})
            /* P = < X_0, ..., X_k > tel que X_i ≠ ∅ et X_i ⊂ X
            Et X_k est le sommet de la pile */
            Tant que P≠ ∅ faire
                S'il existe s_{∈X_k} --arc f--> s'_{∈H} non marqué alors
                    Marqué arc f
                    Si s' ∉ P alors
                        Empiler(P, {s'})
                    Sinon
                        i = indice tel que s' ∈ X_i
                        X_i = X_i ∪ X_{i+1} ∪ ... ∪ X_k
                        Dépiler P k − i fois
                    Finsi
                Sinon
                    C = C ∪ {X_k}
                    H = C − {X_k}
                    Dépiler P une fois
                Finsi
            FinTQ
        FinTQ

    Fin
```

L'implémentation a nécessité l'utilisation de nombreuses fonctions auxiliaires ! (Mini projet 3)

Application : Graphe page 90 (L'implémentation : Mini Projet 3)




```
          Metrics of the Graph
          Time : 03:27:43
          Date : 12/03/2019
       NetWork : composantes
          Type : dirNet
Number of Points : 6
Number of Edges  : 10

-----------------------------------------------

Number of Connected Component : 1   {2,4,5,6}
  Radius :  Undefined (only for undirected graph)
Diameter :  Undefined (only for undirected graph)

Degrees of the Points:
Point          1    2    3    4    5    6
Degree In      0    2    4    2    1    1
Degree Out     2    2    0    2    2    2

Sorted Degrees:
Sorted In Degrees
    1    1    2    2    4
Sorted Out Degrees
    2    2    2    2    2
```

Résultat de notre application : (une composante de cardinale = 1 n'est pas prise en compte par le logiciel Grin 4.0, ce qui explique le résultat : Number of connected componenent dans le rapport ci-dessus).

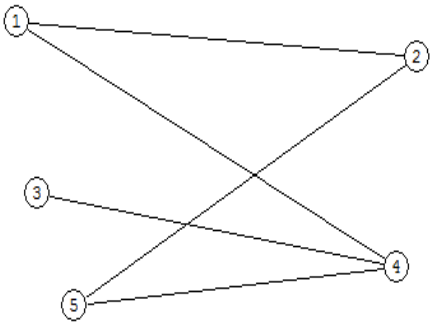
```
Composante 1:
2

Composante 2:
5 4 3 1

Composante 3:
0
```

La numérotaion commence à 0

4) GRAPHE BIPARTI ?

Un graphe est biparti ssi l'ensemble de ses sommets admet une partition en deux sous-ensembles engendrant un sous-graphe stable chacun et de sort que toutes les liens du graphe relient un sommet dans le premier sous-ensemble à un sommet dans le deuxième sous-ensemble.

Considérons le graphe suivant : (Mini Projet 2)



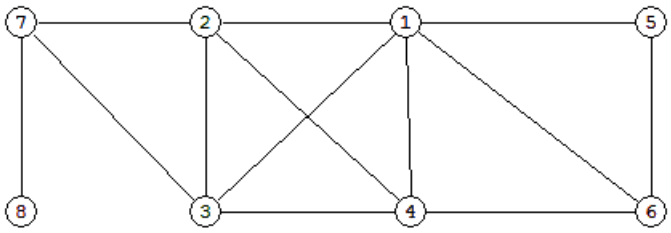

```
>>>>>>> Saisir un type de graphe
----> 1. Graphe Oriente
----> 2. Graphe Non Oriente
2

__________ Graphe Non Oriente ___________

#################### DEBUT ####################

__________ Graphe Bibarti? ___________


 Le graphe est biparti!
```

5) PARTITION DES SOMMETS EN COUCHES

On appelle partition en couches associé à un sommet source (racine r), la suite d'ensembles définie exactement comme suite :

$$C_i = \{x \in X | d(r,x) = i\}$$

Application : graphe page 54 (Parcours en largeur à partir de sommet 7 !)




```
          Metrics of the Graph
          Time : 16:24:02
          Date : 12/03/2019
       NetWork : partition en couches
          Type : undirNet
Number of Points : 8
Number of Edges  : 13

-----------------------------------------

Number of Connected Component : 1
  Radius : 2
Diameter : 4
Center   Points - Green Color Points
Diameter Points -   Red Color Points

Graph is Not Regular.

Degrees of the Points:
Point      1   2   3   4   5   6   7   8
Degree     5   4   4   4   2   3   3   1

Sorted Degrees:
   1   2   3   3   4   4   4   5

Procedure complete.
```

```
4. Construire les couches;
**********************************
**********************************
choix : 4

>>>>>>> Saisir un type de graphe
----> 1. Graphe Oriente
----> 2. Graphe Non Oriente
2

__________ Graphe Non Oriente ___________

#################### DEBUT ####################

__________ Les couches du graphe ____________
C(0) : 7

C(1) : 2 3 8

C(2) : 1 4

C(3) : 5 6


Process returned 0 (0x0)   execution time : 4.521 s
```

Conclusion

Au cours de ce modeste travail on a rencontré des difficulté dans certain tâches, une chose qui a nécessité de faire des recherches afin de les résoudre et apprendre de nouvelles approches et structures de données. Comme perspective, on a l'intention de continuer l'implémentation des autres algorithmes lié aux graphes à savoir l'algorithme de Ford-Fulkerson et encore d'autre algorithmes.

Références

Prof. M. Driss Aouragh, Polycopié du cours : Graphes et algorithmes de graphes, Master SIDI – FST Errachidia 2018/2019

Site internet : https://www.geeksforgeeks.org/category/data-structures/graph/