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

Théorie des Graphes

MINI Projet : Implémentations

Réalisé par : | Hassan ZEKKOURI | 13 Décembre 2018

Responsable :

Prof. M. Driss Aouragh

Introduction

Dans ce rapport je vais présenter les représentations de graphes j’ai implémenté dans le cadre de notre premier mini-projet du module « Théorie des Graphes ». Pendant le cours on a vu différentes représentation à l’instar de :

* La matrice d’adjacence ;
* La matrice d’incidence ;
* Les listes d’adjacence ;
* Les tableaux TS (Tableau de Successeur) et PS (Pointeur premier Successeur);
* Les tableaux APS (Adresse Premier Successeur) et LS (Liste de Successeurs).

Il faut noter que ses implémentations sont lié au graphes sans poids de liaison.

Bonne lecture,

# Représentations machines

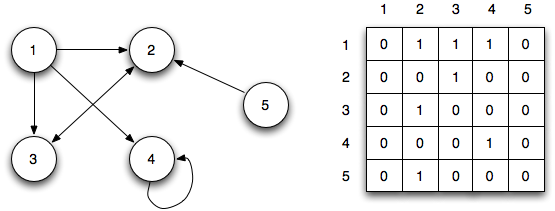
On peut représenter un graphe sur machine avec plusieurs structures de données dont on peut citer les suivantes.

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

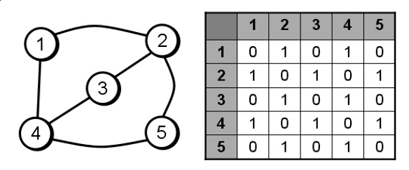
1. La matrice d’adjacence

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

* Pour un graphe non orienté G = (X, E) :
* Pour un graphe orienté G = (X, U) :
* Exemples des testes : (voir CD/ fichier texte)
* Graphe Orienté :



* Graphe Non Orienté :

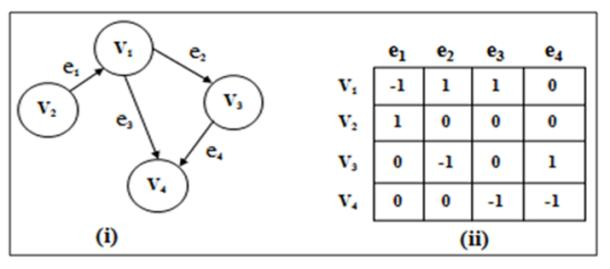


1. Matrice d’incidence

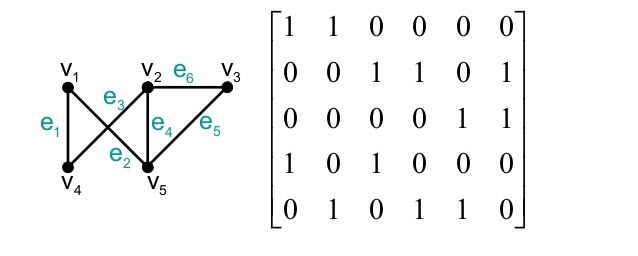
La matrice d’incidence est une matrice, où n est le nombre de sommets du graphe et p est le nombre de liens (arêtes ou arcs).

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

* Graphe Orienté G = (X, U) :
* Graphe Non Orienté G = (X, E) :
* Exemples de teste : (voir CD/ fichier texte)
* Graphe Orienté :



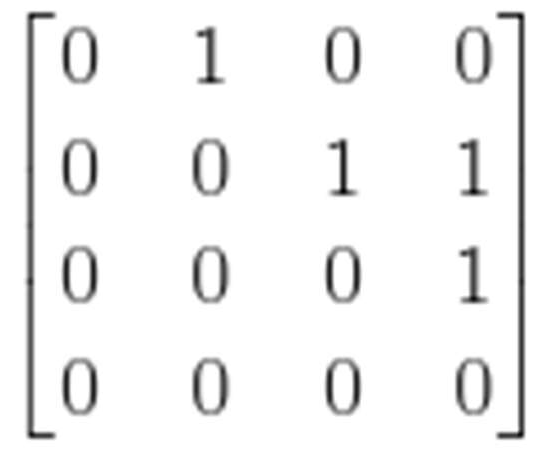
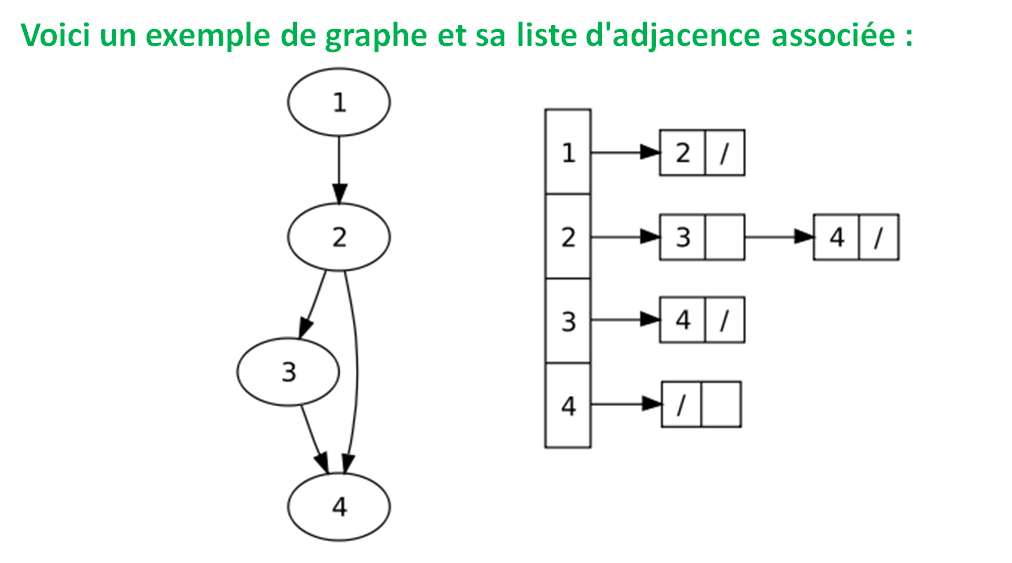
* Graphe Non Orienté :



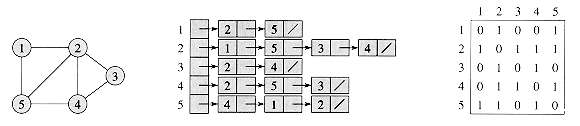
1. Liste d’adjacence

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 :
* Graphe Non Orienté :
* Exemples et testes : (voir CD/ fichier texte)
* Graphe Orienté :



* Graphe Non Orienté :



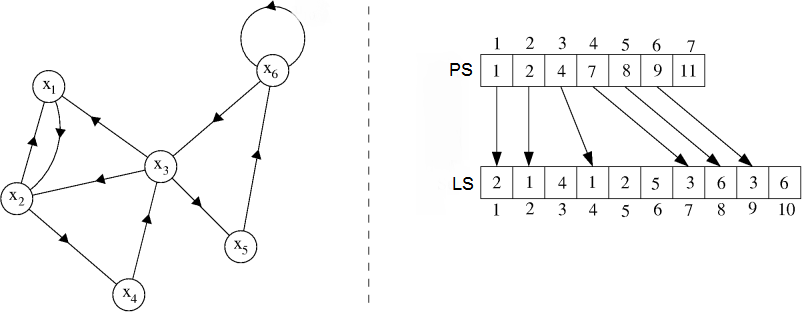
1. Les tableaux PS et LS

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

;

* 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 : (voir CD/ fichier texte)



1. Les tableaux FS 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 = ;

* Graphe Orienté :

APS[s+1]-APS[s] = Nombre de successeurs de sommet s =.

1. représentation avec des arcs

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.

# 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).

1. Parcours en largeur

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

Algorithme 1 :

**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) ;**

**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) ;**

**Finsi**

**Finpour**

**tête = tête + 1 ;**

**finTQ**

**Fin**

Algorithme 2 :

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)**

**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**

1. 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 :

**Procédure BFS (G, s)**

**Variable n = order(G)**

**marque: Tableau**

**Debut**

**Pour i = 1 à n faire**

**marque[i].couleur = blanc ;**

**marque[i].père = null ;**

**Finpour**

**marque[s].couleur = gris ;**

**marque[s].dist = 0 ;**

**temps = 0 ;**

**pour tout u sommet de G faire**

**si (marque[u] = 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)**

**Variable**

**marque: Tableau**

**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.

Ici on va choisir le dernier sommet successeurs dans la liste comme sommet suivant, il sera dans la tête de la pile.

**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 :

Ici c’est le premier successeur qu’on choisit comme sommet suivant.

**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**

# Arbre couvrant de poids minimal

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

* Algorithme de Kruskal ;
* Algorithme de Prim ;

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**

1. 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 et y**

**M = Union(M, y)**

**F = Union(F, a)**

**FinTQ**

**Fin**

1. Plus court chemin

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

* Le tableau lambda , contient la longueur du plus court chemin à chaque sommet de graphe.
* Le tableau Pi contient les prédécesseurs ou les pères de chaque sommet dans le plus court chemin.

**Principe générale :**

* On initialise les tableaux et  , Initialisation() ;
* On calcule les et  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 (y) obtenue à l’étape précédente avec la valeur qu’on obtiendrait en passant par x, i.e. v(x, y). C’est la technique de relâchement, Relâcher (G, i, j).

**Algorithme 1 :**

**Initialisation (G,)**

**Variable**

**Lambda, Pi : Tableau**

**Debut**

**Pour i à n faire**

**Lambda[i] = infini;**

**Pi[i] = null;**

**Finpour**

**Lambda [] = 0;**

**Marquer**

**Fin**

**Algorithme 1 :**

**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 [] = 0 ;**

**Marquer**

**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 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 est minimale parmi tous les sommets non encore marqués.

**Algorithme de Dijkstra :**

**Dijkstra (G,)**

**Variable**

**E : ensemble de sommets marqués**

**Début**

**Initialisation (G,)**

**E = vide**

**Tant que E  S faire**

**i = un sommet non marquer de 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**

1. Algorithme de bellman

Cet algorithme s’applique aux graphes sans circuits et pour la recherche de plus court chemin d’un sommet  **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,)**

**numéroter 1 le sommet**

**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 Dijkstra :**

**Dijkstra (G,)**

**Variable**

**n : ordre de G**

**Début**

**triTopo (G,)**

**Initialisation (G,)**

**Pour j = 2 à n faire**

**Pour tout i prédécesseur de j faire**

**Relâcher (G, i, j)**

**Finpour**

**Pourpour**

**Fin**

1. 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,)**

**Variable**

**n : ordre de G**

**Début**

**Initialisation (G,)**

**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**

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.

**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.**