Chap IV : Les graphes

1. Introduction : les graphes sont des outils très puissants et largement répandus permettant la résolution de nombreux problèmes de façon formelle (mathématique, exacte). Tout problème comportant des objets avec des relations entre ces objets peut être modélisé par un graphe.

- Les réseaux de transport (routiers, ferrés, aériens, · · ·) ;
- Les réseaux téléphoniques, électriques, de gaz, · · · ;
- Les réseaux d'ordinateurs ;
- Les réseaux sociaux ;
- Ordonnancement des tâches ;
- Circuits électroniques ;
- …

2. Définitions : un graphe est un ensemble de nœuds reliés entre eux (nœud, liens).

On appelle les nœuds « sommets » et les liens « arcs ».

Un graphe G est défini par un couple (S, A) où S est l'ensemble de sommets et A est l'ensemble des arcs représentant les relations existant entre les sommets.

Exemple : soit le graphe G représenté par :
S={1, 2, 3, 4, 5, 6, 7} ;
A={(1,3), (1,4), (1,5), (2,3), (2, 4), (3,4), (3, 7), (4, 5), (7, 1)}

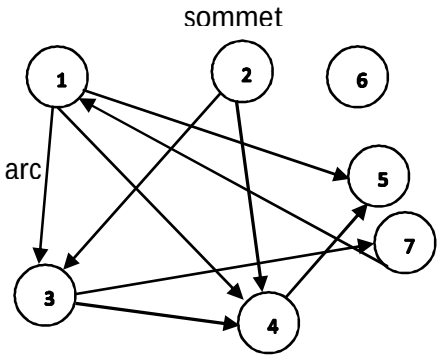


Fig.1 : graphe orienté

- Graphe orienté : lorsque les relations entre les sommets sont à sens unique (définies dans un sens), alors on les appelle « arcs ».
Exemple : réseau social (personnes, confiances).

- Graphe non orienté : lorsque les relations entre les sommets sont définies dans les 2 sens, alors on les appelle « arrêtes ».
Exemple : réseau routier (villes, distances).

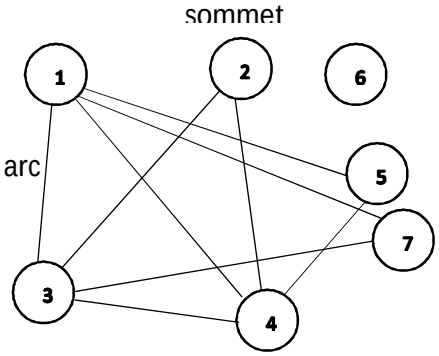


Fig.2 : graphe non orienté

- Graphe pondéré : à chaque arc (ou arrête) est associé une valeur ou un libellé.
Exemple :

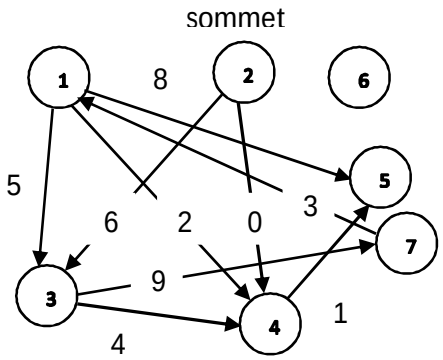


Fig.3 : graphe pondéré

-Origine et extrémité :

- Si a = (x, y) est un arc de x vers y alors :
* x est l'origine de a et y est son extrémité ;
* x est un prédécesseur de y et y est un successeur de x.
- Si l'origine et l'extrémité d'un arc se coïncident on l'appelle une boucle.

- Chemin : un chemin est un ensemble d'arcs $a_1, a_2, \cdots, a_p$
où Origine(a_{i+1}) = Extrémité(a_i).
On dit que le chemin est de longueur p.
Exemple : {(1, 3), (3, 4), (4, 5)}.

- Circuit : un circuit est un chemin $a_1, a_2, \cdots, a_p$ où Origine(a_1) = Extrémité(a_p)
Exemple : {(**1**,3), (3,5), (7,**1**)}.

- Chaine : une chaine est un chemin non orienté.
Exemple : {(1, 5), (5, 4), (4, 2)}.

- Cycle : un cycle est une chaîne fermée.
Exemple : {(1,4), (4,5), (5,1)}.

- Graphe fortement connexe : un graphe orienté est dit **fortement connexe** lorsqu'il existe une chemin entre tout couple de sommets (x,y).
Exemple : Le graphe précédent n'est pas **fortement connexe**, car il n'existe pas de chemin entre pour le couple (3,2).

- Graphe connexe : un graphe non orienté est dit **connexe** lorsqu'il existe une chaine entre toute couple de sommets (x,y).
Exemple : Le graphe précédent n'est pas **connexe**, car il n'existe pas de chaines entre le sommet 6 et les autres sommets.

3. Représentation des graphes : un graphe peut être représenté de deux manières : soit par des listes d'adjacence (dynamique), soit par une matrice d'adjacence (statique)

3.1. Listes d'adjacence (LA) : dans cette représentation, les successeurs d'un nœud sont rangés dans une liste linéaire chainée. Le graphe est représenté par un tableau T où T[i] contient la tête de la liste des successeurs du sommet numéro i. Le graphe (S, A) précédent peut être représenté comme suit :

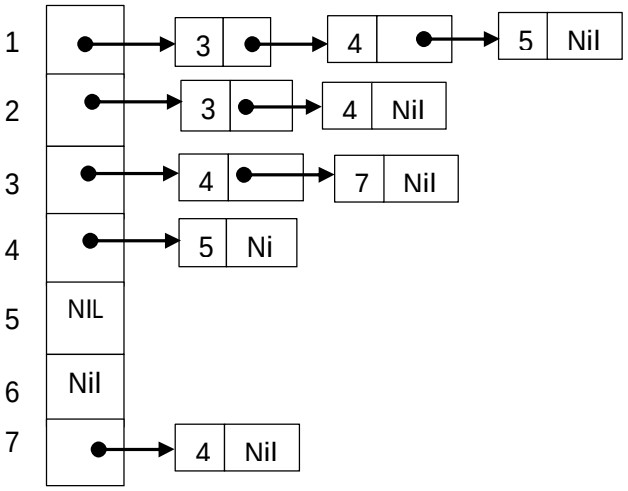


Fig.4 : représentation d'un graphe par des ''LA''

Remarques :
- les listes d'adjacence sont triées par numéro de sommet ($sommet_i \rightarrow liste_i$), mais les successeurs d'un sommet donné peuvent apparaître dans n'importe quel ordre ;
- Les graphes peuvent être exploités en utilisant les outils, puissants, fournis par la théorie des graphes.

Les types ptr (pointeur sur un sommet), sommet et Graphe sont donnés comme suit :

```
type ptr =^sommet ;
     sommet = enregistrement
                    info : entier ;
                    suivant : Ptr ;
              fin ;
```

On déclare un graphe g comme suit : var g : Tableau[1..n] de Ptr ;

3.2. Matrice d'adjacence (MA): Dans cette représentation, le graphe est stocké dans un tableau à deux dimensions de valeurs binaires. Chaque case (x,y) du tableau est égale à 1 s'il existe un arc de du sommet x vers le sommet y, 0 sinon. Le graphe précédent peut être par la matrice d'adjacence suivante :

Fig.5 : représentation d'un graphe par une ''MA''

	1	2	3	4	5	6	7
1	0	0	1	1	1	0	0
2	0	0	1	1	0	0	0
3	0	0	0	1	0	0	1
4	0	0	0	0	1	0	0
5	0	0	0	0	0		0
6	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0

On déclare un graphe g comme suit : var g : Tableau[1..n, 1..n] de booléen ;

Remarques :
- Pour un graphe pondéré, chaque valeur d'une case case (x,y) de la matrice représente le poids de l'arc (x,y), avec une valeur particulière pour les arcs inexistant, par exemple -1. Le graphe pondéré précédent peut être représenté comme suit :

Fig.6 : représentation d'un graphe pondéré par une ''MA''

	1	2	3	4	5	6	7
1	-1	-1	5	2	8	-1	-1
2	-1	-1	-1	0	-1	-1	-1
3	-1	-1	-1	4	-1	-1	9
4	-1	-1	-1	-1	1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1
7	3	-1	-1	-1	-1	-1	-1

- Les matrices peuvent être exploitées en utilisant les calculs matriciels.

4. Parcours de graphes :

De même que pour les arbres, il est important de pouvoir parcourir un graphe afin d'effectuer certains traitement, cependant le parcours des graphes est un peu différent de celui des arbres :

a- Dans un arbre, à partir de la racine on peut atteindre tous les nœuds, ce qui n'est pas le cas pour un graphe où on est obligé reprendre le parcours plusieurs fois jusqu'à atteindre tous les sommets ;

b- Dans un arbre, un sommet s n'est atteint que par son père ; par contre dans un graphe, un sommet s peut être atteint à partir de plusieurs sommets (plusieurs chemin vers s), voir par lui-même (circuit : chemin de s vers). Pour éviter de visiter un sommet plusieurs fois (plus d'une fois), on utilise un tableau de booléens "visité'', initialisé à faux ; où à chaque fois qu'on visite un sommet j, on met visité[j] ←Vrai.

c- Il existe deux types de parcours de graphes : le parcours en profondeur d'abord (Depth First Search : DFS) et le parcours en largeur d'abord (Breadth First Search : BFS).

4.1. Le parcours en profondeur d'abord (DFS) : C'est le parcours le plus utilisé sur les graphes. C'est la généralisation du parcours Préfixe sur les arbres (plusieurs fils pour un sommet au lieu de 2).

Principe :

1- Initialement, tous les nœuds sont marqués " non visités" ;
2- Choisir un sommet s de départ (non visité) et le marquer " visité" ;
3- Chaque nœud adjacent à s, non visité, est à son tour visité en utilisant DFS récursivement ;
4- Une fois tous les sommets accessibles à partir de s ont été visités, alors la recherche de sommets à partir de s (DFS(s)) est complète ;
5- Si tous les sommets ne sont pas visités, on recommence à partir de (2).

{DFS avec matrice d'adjacence : algo1}

```
var g : Tableau[1..n,1..n] de booleen ;
    visité : Tableau[1..n] de booleen ;

procédure DFS( sommet : entier);
var  i : entier ;
début
        visité[sommet] ←Vrai ;
        écrire(sommet) ;
        pour i allant de1 à n faire
                si (g[sommet, i] et non visité[i]) Alors
                        DFS(i) ;
                finSi;
        finPour;
fin;
```

Appel {pp} :

```
pour i allant de 1 à n faire
        visité[i] ←faux ;
finPour ;

pour i allant de 1 à n faire
        si (Non visité[i]) Alors
                DFS(i) ;
        finSi ;
finPour ;
```

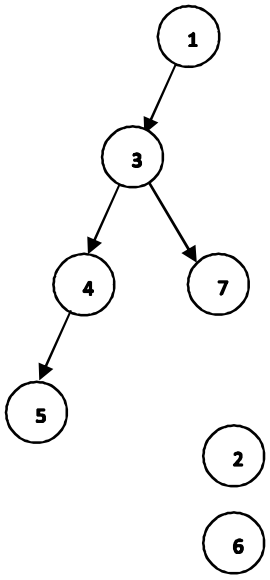
{DFS avec listes d'adjacence : algo2}

```
var g : Tableau[1..n] de ptr ;
procédure DFS( v : entier); //v est un sommet
var  j : entier ;
début
        visité[v] ←Vrai ;
        écrire(v) ;
        Pour chaque sommet j adjacent à v faire
                si non visité[j] Alors
                        DFS(j) ;
                finSi;
        finPour;
fin;
```

Appel {pp} :

```
pour i allant de 1 à n faire
        visité[i] ←faux ;
 finPour ;

pour chaque sommet i de g faire
        si (non visité[i]) Alors
                DFS(i) ;
        finSi;
 finPour;
```

Exemple : le parcours en DFS du graphe précédent (DFS(1), DFS(2), DFS(6)) affiche l'ordre suivant : 1,3, 4, 5, 7, 2, 6.



Remarque : La seule différence entre les deux algorithmes algo1 et algo2 réside dans la manière dont les fils sont visité (couleur jaune). Dans algo1, on vérifie tous sommets du graphe (n opérations) ; par contre dans algo2, on vérifie uniquement les voisins (k opérations). Même chose pour algo3 et algo4 ci-dessous.

4.2. Le parcours en largeur d'abord (BFS) C'est le parcours le moins utilisé sur les graphes.

Dans ce parcours, un sommet s est fixé comme origine et on visite tous les sommets situés à une distance k de s avant de passer à ceux situés à k + 1 en utilisant une file ; si tous les sommets ne sont pas visités à partir de s, on choisit un autre sommet non visité comme origine et on recommence de nouveau.

Principe :
1- Initialement, tous les nœuds sont marqués " non visités" ;
2- Choisir un sommet s de départ (non visité) et le marquer " visité" ;
3- Enfiler s dans f ;
4- On défile un sommet de f et on enfile tous ses successeurs ;
5- On répète (4) jusqu'à ce que la file soit vide (la recherche de sommets à partir de s (BFS(s)) est complète) ;
5- Si tous les sommets ne sont pas visités, on recommence à partir de (2).

{BFS avec matrice d'adjacence} algo3

```
var g : Tableau[1..n,1..n] de booleen ;
    visité : Tableau[1..n] de booleen ;
    f : File d'attente ;

Procédure BFS( sommet : entier);
Var  i,s : entier ;
début
        enfiler(f,sommet) ;
        visité[sommet] ←Vrai ;
        tant que (non file_Vide(f)) faire
                Defiler(f,s) ;
                écrire(s) ;
                pour i de 1 à n faire
                    si (g[s,i] et non visité[i]) alors
                                enfiler(f,i) ;
                                visité[i] ←Vrai ;
                    finSi ;
                finPour ;
        finTQ;
fin ;
```

Appel {pp} :

```
Pour i allant de 1 à n faire
        visité[i] ←faux ;
 finPour ;

pour i de 1 à n faire
        Si (non visité[i]) Alors
                BFS(i) ;
        finSi;
 finPour;
```

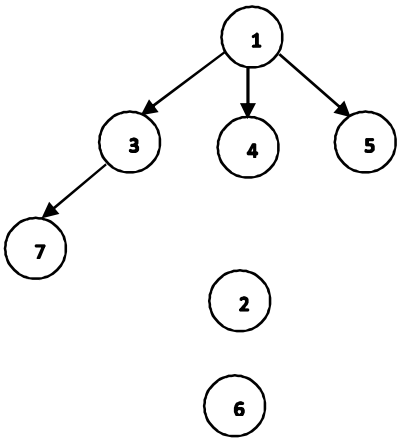
{BFS avec listes d'adjacence} algo4

```
var g : Tableau[1..n] de ptr ;
    visité : Tableau[1..n] de booleen ;
    f : File d'attente ;

Procédure BFS( v : entier);
Var  s, w : entier ;
début
        enfiler(f,v) ;
        visité[sommet] ←Vrai ;
        tant que (non file_Vide(f)) faire
            defiler(f,s) ;
            écrire(s) ;
            pour chaque sommet w adjacent à s faire
                    si (non visité[w]) Alors
                                enfiler(f,w) ;
                                visité[w] ←Vrai ;
                    finSi ;
            finPour ;
        finTQ;
fin ;
```

Appel {pp} :

```
Pour i allant de 1 à n faire
        visité[i] ←faux ;
 finPour ;

pour chaque sommet i de g faire
        si (non visité[i]) Alors
                BFS(i) ;
        finSi;
 finPour;
```

Exemple : le parcours en BFS du graphe précédent (BFS(1), BFS(2), BFS(6)) affiche l'ordre suivant : 1,3, 4, 5, 7, 2, 6.



Remaruqe : le code des 2 boucles ‘‘pour’’ utilisé ci-dessus sont donnés comme suit :

```
pour chaque sommet w adjacent à s faire
        si (non visité[w]) Alors
                enfiler(f,w) ;
                visité[w] ←vrai ;
        finSi ;
finPour ;

var p : ptr ;
p←g[s] ;
tant que p<>nil faire
        si non visité(p^.info) alors
                visité[p^.info] ←vrai ;
                enfiler(f, p^.info) ;
        finSi ;
        p←p^.suivant ;
finTQ ;
```

```
Pour chaque sommet j adjacent à v faire
        si non visité[j] Alors
                DFS(j) ;
        finSi;
finPour;


var p : ptr ;
p←g[v] ;
tant que p<>nil faire
        si non visité(p^.info) alors
                DFS(p^.info) ;
        finSi ;
        p←p^.suivant ;
finTQ ;
```

4.3. Objectifs des parcours de graphes : les procédures DFS et BFS peuvent être utilisées pour divers objectifs :

1- Pour vérifier s’il existe un chemin entre 2 sommet s1 et s2 ;
2- Pour vérifier si un circuit contentent deux sommets s1 et s2 existe ;
3- Pour trouver les chemins minimums d’un sommet s vers tous les autres ;
4- etc.

Exercice :

1- écrire un algorithme qui permet de vérifier s’il existe un chemin d’un sommet x à un sommet y dans un graphe g.
2- Ecrire un algorithme qui permet de renvoyer le niveau d’un sommet x par rapport à un sommet y dans un graphe g.