Graphes

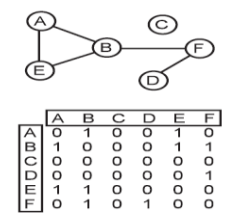
(suite)

Implémentation en C d'un Graphe par Liste d'Adjacence

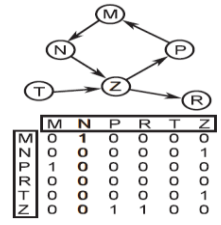
```
#define N_MAX 20
typedef struct cellule {
  int sommet;
  struct cellule* suiv;
} Cellule;
typedef Cellule* Liste;

typedef struct {
  Liste a[N_MAX];
  int n;
} GrapheL;
```

49

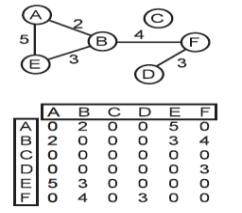
Matrices vs Listes d'Adjacences de Graphes Orientés et Non Orientés



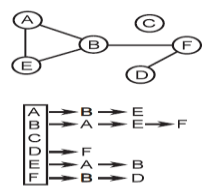
(a) Adjacency matrix of an undirected graph.



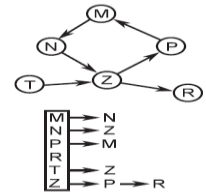
(b) Adjacency matrix of a directed graph.



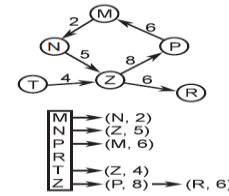
(c) Adjacency matrix of an undirected weighted graph.



(a) Adjacency list of an undirected graph.



(b) Adjacency list of a directed graph.



(c) Adjacency list of a directed weighted graph.

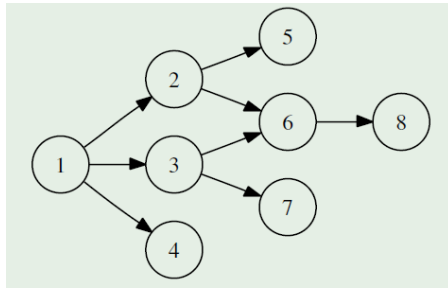
50

Parcours d'un Graphe

- **Parcours de tous les sommets :**
  - **visiter chaque sommet du graphe une seule fois**
  - **appliquer un même traitement en chaque sommet**
- **Parcours à partir d'un sommet s :**
  - **Parcours en profondeur d'abord (Depth First Search)**
    - **le principe consiste à descendre le plus "profondément" dans le graphe à partir de s, en prenant toujours à "gauche", avant de revenir pour prendre une autre direction**
  - **Parcours en largeur d'abord (Breadth First Search)**
    - **le principe consiste à visiter les sommets situés à une distance 1 de s, puis ceux situés à une distance 2 de s, etc...**
    - **d'une autre manière, lorsqu'un sommet x est atteint, tous ses successeurs y sont visités avant de visiter les autres descendants de x.**

51

Parcours d'un Graphe (Exemple)

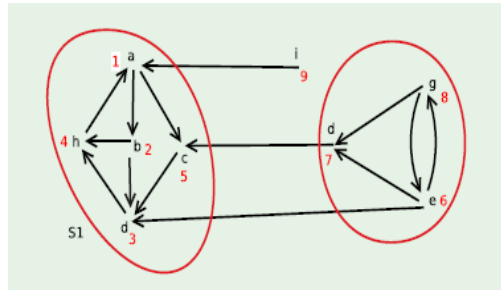


- **Parcours à partir du sommet 1 :**
  - **Parcours en profondeur : 1, 2, 5, 6, 8, 3, 7, 4**
  - **Parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8**

52

Parcours en Profondeur (Exemple)

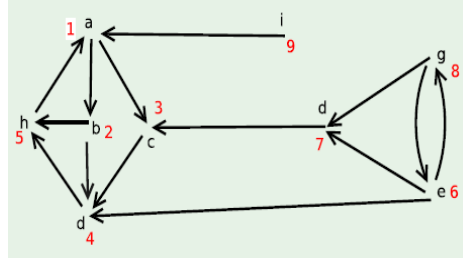
- les numéros correspondants aux sommets donnant l'ordre dans lequel les sommets sont visités.



53

Parcours en Largeur (Exemple)

- La numérotation indiquée correspond à un ordre de visite lors d'un parcours en largeur



54

Comment implémenter les deux parcours dans un graphe ?

- **Le type de parcours est fonction du TAD utilisé pour stocker les sommets à traiter :**
  - **Pile → Parcours en profondeur**
  - **File → Parcours en largeur**
- **Parcours en profondeur**
  - **Algorithme récursif : l'utilisation de la pile est implicite (appels récursifs)**
  - **Algorithme itératif : l'utilisation de la pile est explicite**
- **Parcours en largeur**
  - **Algorithme itératif : l'utilisation de la file est explicite**
- **Dans tous les cas il faut un mécanisme pour éviter de boucler indéfiniment :**
  - **Marquer les noeuds**
  - **Lister les noeud traités**

55

Algorithme de Parcours en Profondeur (1)

```
Algorithme parcoursProfondeur(g : Graphe)
 Entrée : un graphe

 Variables locales
   atteint : tableau[Sommet] de booléens
   (* atteint[x] <=> le sommet x a été atteint *)
   x : Sommet
Début
  pour tout sommet x de g faire
    atteint[x] ← faux
  fpour
  pour tout sommet x de g faire
    si non atteint[x] alors RechercheProf(x)
  fpour
Fin
```

56

Algorithme de Parcours en Profondeur (2)

```
Algorithme RechercheProf(x : Sommet)
 Entrée : un sommet d'un graphe
 (* Étant donnée un sommet x non atteint, cet
    algorithme marque x, et tous les sommets y
    descendants de x tels qu'il existe un chemin
    [x,y] dont aucun sommet n'est marqué *)

 Variables locales :
   y : Sommet

Début
 atteint[x] ← vrai
 pour tout successeur y de x faire
   si non atteint[y] alors RechercheProf(y)
 fpour
Fin
```

57

Algorithme de Parcours en Profondeur (Complexité)

- La phase d'initialisation du tableau atteint est en O(n)
- L'itération de l'algorithme principal est réalisée exactement en n étapes, pour chacune il y'a au moins un test réalisé :
  - O(n+m), soit O(max(n,m)) dans le cas des listes d'adjacences
  - $O(n^2)$ dans le cas des matrices d'adjacences

58

Parcours en profondeur (Ordre de Traitement-Numérotation)

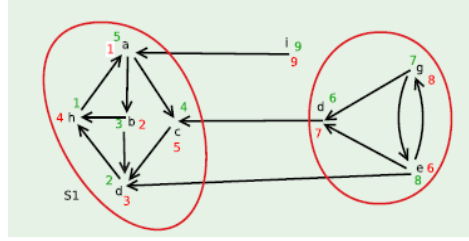
- **L'objet des parcours de graphes concerne des traitements que l'on souhaite opérer sur les graphes :**
  - Les traitements s'opèrent parfois sur les sommets visités. Il est alors possible d'opérer un traitement avant ou après la visite du sommet.
  - Il y'a donc soit **un traitement en préOrdre ou ordre préfixé**, soit **en postOrdre ou ordre postfixé**. Ceci se traduit par une modification de la procédure de recherche en profondeur.

```
Algorithme traitement_en_préOrdre(x : Sommet)
 Variables locales :
   y : Sommet
Début
 atteint[x] ← vrai
 <traiter x>
 pour tout successeur y de x faire
  si non atteint[y] alors rechercheProf(y)
 fpour
Fin
```

59

Parcours en profondeur (Traitement en postOrdre)

```
Algorithme traitement_en_postOrdre(x : Sommet)
  Variables locales :
      y : Sommet
Début
  atteint[x] ← vrai
  pour tout successeur y de x faire
      si non atteint[y] alors rechercheProf(y)
 <traiter x>
Fin
```



Exemple : Numérotation en préOrdre(rouge) et en postOrdre(vert) pour un parcours en profondeur

Mise en œuvre en C du parcours en profondeur (1)

```
#define N_MAX 20

typedef struct cellule {
 int sommet;
 struct cellule *suiv;
} Cellule
typedef cellule* Liste;

typedef struct {
 Liste a[N_MAX];
 int n;
} GrapheL;

typedef int atteint[N_MAX];

/* variables globales déclarées:
GrapheL g;
atteint m; */
```

61

Mise en œuvre en C du parcours en profondeur (2)

```
void parcoursProf(int x) {
 Liste p;
 m[x]=1;
 p=g.a[x];
 while(p!=NULL) {
  if(!m[p->sommet]) parcoursProf(p->sommet);
  p=p->suiv;
 }
}

void main(){
 int x;
 for(x=1;x<g.n;x++) m[x]=0;
 for(x=1;x<g.n;x++)
  if(!m[x]) parcoursProf(x);
}
```

62

Mise en œuvre en C du parcours en Largeur (1)

- **Principe de l'algorithme :**
  - Il repose sur la notion de file.
  - Lors de la visite d'un sommet s, tous ses successeurs non encore atteints vont être rangés dans la file de manière à conserver la priorité liée aux distances depuis le sommet origine.

```
typedef struct {
  Liste tete,queue;
} File;

void enfiler(int x,File* f);
int defiler(File* f);
int fileVide(File* f);
```

63

Mise en œuvre en C du parcours en largeur (2)

```
void parcoursLarg(int x) {
 Liste p;
 initFileVide(&f);
 enfiler(x,&f);
 m[x]=vrai;
 while(!fileVide(f)) {
  x=defiler(f);
  p=g.a[x];
  while(p!=NULL) {
   if (!m[p->sommet]) {
     m[p->sommet]=vrai;
     enfiler(p->sommet,f);
   }
   p=p->suiv;
  }
 }
}
```

```
void main(){
 int x;
 for(x=1;x<g.n;x++) m[x]=0;
 for(x=1;x<g.n;x++){
  if(!m[x])
     parcoursLarg(x);
 }
}
```

64

Algorithme de Parcours en Largeur (Complexité)

- Identique à celle du parcours en profondeur :
  - O(n+m) pour les listes d'adjacence
  - $O(n^2)$ pour les matrices d'adjacence

65

Quelques Applications des Parcours

- **Accessibilité :**
  - Pour connaître les sommets accessibles depuis un sommet donné d'un graphe (orienté ou non), il suffit de faire un parcours en profondeur à partir de ce sommet, en marquant les sommets visités
- **Composantes connexes :**
  - Pour déterminer les composantes connexes d'un graphe, il suffit d'appliquer d'une manière répétitive le parcours DFS ou BFS sur tous les sommets non encore visités. Il est clair qu'une composante connexe est constituée du sous graphe dont les sommets sont visités par un seul appel à DFS ou BFS
- **Graphe orienté sans circuit :**
  - Un graphe orienté comporte un circuit si et seulement si, lors du parcours des sommets accessibles depuis un sommet, on retombe sur ce sommet. Pour savoir si un graphe est sans circuit, il suffit donc d'adapter DFS ou BFS, en maintenant une liste des sommets critiques (en cours de visite)
- **…**

66

Algorithme du Parcours en Profondeur Récursif

```
Algorithme parcoursEnProfondeurRecursif
                        (g : Graphe, s : Sommet )

Entrées : un graphe et un sommet

Début
 si non estMarque(s) alors
   marquer(s)
   traiter(g,s)
   pour chaque successeur s' de s faire
     parcoursEnProfondeurRecursif(g,s')
   fpour
  fsi
Fin
```


Algorithme du Parcours en Largeur

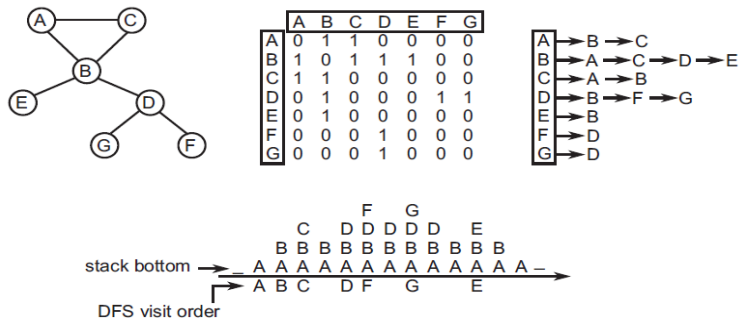
```
Algorithme parcoursEnLargeurIteratif (g : Graphe, s : Sommet )
 Entrées : un graphe et un sommet du graphe
 Variables locales :
  f : File<Sommet>
  sCourant : Sommet

Début
  f ← file()
  f ← enfiler(f,s)
  tantque non estVide(f) faire
    sCourant ← obtenirElement(f)
    f ← defiler(f)
    marquer(sCourant)
    traiter(g,sCourant)
    pour chaque successeur s' de sCourant faire
      si non estMarque(s') alors
        f ← enfiler(f,s')
      fsi
    fpour
  ftantque
Fin
```

Graphes (Applications)

- **Algorithmes sur les graphes :**
  - Algorithmes résolvant les problèmes modélisés par les graphes. Par exemple, les problèmes liés à l'optimisation des connexions et du routage. On peut citer les algorithmes de calcul des arbres de recouvrement minimaux, la recherche des plus courts chemins, …
- **Tri topologique :**
  - Il s'agit d'un tri linéaire des sommets dans un graphe orienté acyclique de telle sorte que tous les arcs vont de gauche à droite. L'une de ses utilisations les plus courantes est de déterminer un ordre acceptable dans l'accomplissement d'un certain nombre de tâches dépendant les unes des autres
- **Coloration de graphes :**
  - On tente de donner une couleur aux sommets de façon à ce qu'il n'y ait pas deux sommets de même couleur reliés par un arc. Parfois, on s'intéresse à déterminer le nombre minimum de couleurs réalisant ce but.
- **Problèmes de cycles hamiltoniens :**
  - On travaille sur des cycles hamiltoniens, des chemins passant exactement une fois par tous les sommets d'un graphe avant de revenir au sommet de départ. Le problème du voyageur de commerce en est un cas particulier, dans lequel on recherche le circuit hamiltonien de coût minimum.
- **Problèmes de clique :**
  - On travaille sur des régions du graphe où chaque sommet est connecté d'une façon ou d'une autre à tous les autres sommets. Ces régions s'appelent des cliques. On recherche une clique maximale dans un graphe, ou une clique d'une certaine taille, …
- **…**

69

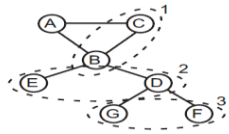
Parcours en Profondeur et Pile (Exemple)



Contents of the queue during a depth first search

70

Parcours en Largeur et File (Exemple)



	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	1	1	1	0	0
C	1	1	0	0	0	0	0
D	0	1	0	0	0	1	1
E	0	1	0	0	0	0	0
F	0	0	0	1	0	0	0
G	0	0	0	1	0	0	0

A	B → C
B	A → C → D → E
C	A → B
D	B → F → G
E	B
F	D
G	D

BFS visit order	queue front
	A
A	B C
B	C D E
C	D E
D	E F G
E	F G
F	G
G	

Contents of the queue during a breadth first search. The vertices grouped by the dotted ovals represent their distance from the start vertex.

71