Structures de données (LU2IN006)

Graphe

Nawal Benabbou

Licence Informatique - Sorbonne Université

2024-2025

Structure non-linéaire et cyclique : les graphes

Question

Comment manipuler des structures non-linéaires et pouvant posséder des cycles ?

Exemples

- La représentation d'un catalogue de formations, ou des appartenances d'une association (cf TD5).
- Une carte routière, un schéma des transports publics...
- Le plan de transport d'une entreprise entre ses usines, ses entrepôts et ses clients.
- La représentation des hyper-liens d'internet.
- La représentation des liaisons électriques entre les composants électroniques d'un circuit intégré.
- Les liens logiques entre les idées dans un texte.
- Les déplacements possibles d'un robot.

Graphe orienté (objet mathématique - théorie des graphes)

Définition : graphe orienté

Un *graphe orienté* est un couple $G = (S, A)$ où :
- S est un ensemble d'éléments appelés *sommets*,
- A est un ensemble de paires (orientées) de sommets de $S \times S$ appelées *arcs*.

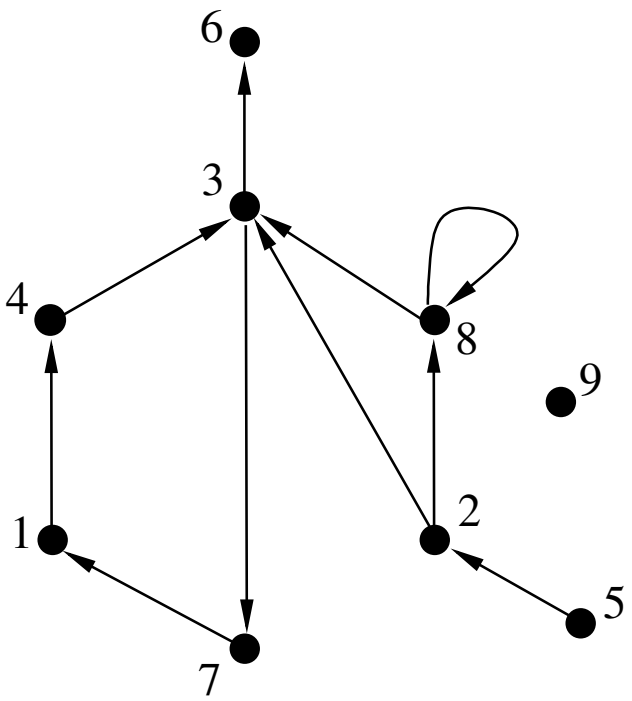
Définition : sommet

Dans un graphe $G = (S, A)$, l'ensemble S est l'ensemble des sommets. Le nombre de sommets est souvent noté n (c-à-d $n = |S|$).
- Un sommet est représenté graphiquement par un rond ou un carré numéroté.
- Le numéro (ou la valeur) d'un nœud est donné à l'intérieur du rond ou à côté.

Définition : arc

Dans un graphe $G = (S, A)$, l'ensemble A est l'ensemble des arcs. Le nombre d'arcs est souvent noté m (c-à-d $m = |A|$).
- Un arc est une paire (orientée) de sommets (appelés les *extrémités* de l'arc).
- Un arc $a \in A$ peut être noté (s_i, s_j) où s_i et s_j sont les extrémités de l'arc.
- Si $(s_i, s_j) \in A$, alors on dit que s_i est un *prédécesseur de* s_j dans le graphe, et que s_j est un *successeur* de s_i.
- L'arc (s_i, s_j) est représenté graphiquement par une flèche allant de s_i vers s_j.

Exemple



Dans cet exemple, on voit que :

- le sommet 5 n'a pas de prédécesseur.
- le sommet 6 n'a pas de successeur.
- le sommet 3 a trois prédécesseurs et deux successeurs.
- le sommet 8 est son propre prédécesseur et successeur.
- le sommet 9 n'a ni prédécesseur, ni successeur (on dit qu'il est *isolé*).

Interprétations

Sommet

Les sommets représentent souvent des lieux (villes, usines) ou des objets (pièces à manipuler en usine, ordinateur). Un sommet est très souvent associé à des données :

- un entier (ou un nom) pour le repérer.
- un ensemble d'informations sur sa description, comme une couleur, des coordonnées, ou encore l'objet qu'il représente.
- un valeur numérique pouvant représenter un coût, un poids, une utilité...

Arc

Les arcs représentent souvent des déplacements, des choix, des liens ou des distances. Un arc est parfois associé à des données :

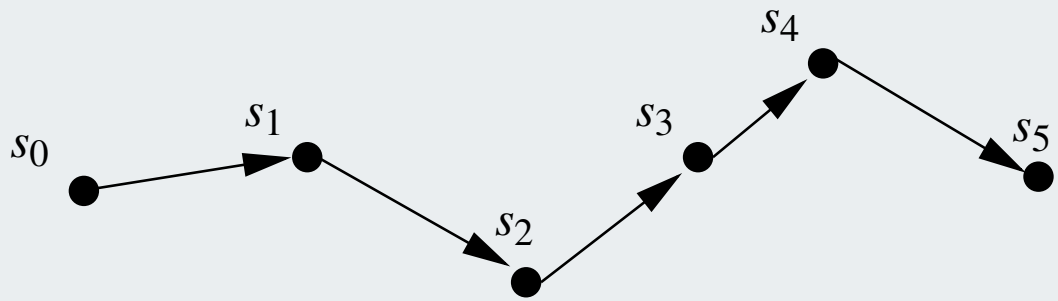
- un entier (ou un nom) pour le repérer.
- un ensemble d'informations sur sa description.
- une valeur numérique représentant une distance, un coût, une capacité...

Chemin dans un graphe orienté

Définition : chemin

Dans un graphe $G = (S, A)$, un chemin est une suite d'arcs reliant des sommets successifs. Plus formellement, un chemin P est une séquence d'arcs $P = (a_1, a_2, \ldots, a_k)$ qui vérifie :

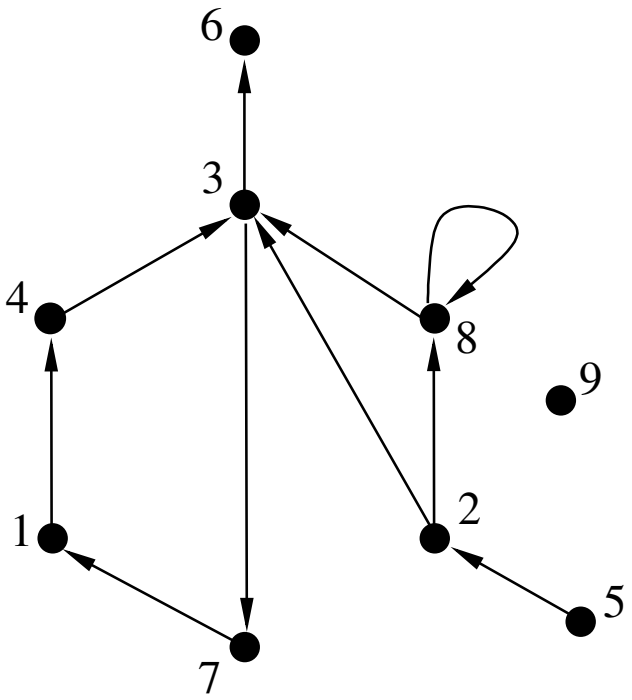
- $a_1 = (s_0, s_1)$,
- $a_2 = (s_1, s_2)$,
- $\ldots$
- $a_k = (s_{k-1}, s_k)$.



Définitions : descendant, ascendant, sommet inaccessible

- Le sommet s_i est un *descendant* du sommet s_j s'il existe un chemin allant de s_j à s_i dans le graphe.
- Le sommet s_i est un *ascendant* (ou ancêtre) du sommet s_j s'il existe un chemin allant de s_i à s_j dans le graphe.
- Le sommet s_i est dit *inaccessible* depuis s_j s'il n'existe aucun chemin allant de s_j à s_i dans le graphe.

Exemple



Dans cet exemple, il y a plusieurs chemins.
Par exemple :

- $P_1 = ((5,2),(2,8),(8,3),(3,6))$
- $P_2 = ((4,3),(3,7))$

On peut voir aussi que :

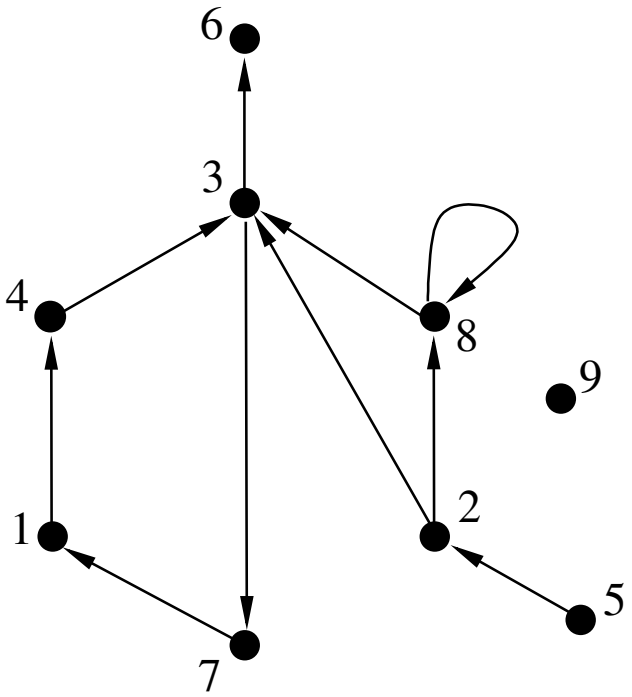
- 6 est descendant de 5 (chemin P_1), mais que l'inverse n'est pas vrai.
- 2 est inaccessible depuis 3.

Circuit

Définition

Un circuit est un chemin reliant un sommet à lui-même.

Remarque : Les sommets d'un circuit sont donc tous ascendants et descendants les uns des autres !



Dans cet exemple, il existe un circuit :
$C = ((7,1),(1,4),(4,3),(3,7))$

Type de données abstrait : Graphe

Définition

Le type de données abstrait “graphe” permet de représenter l'objet mathématique appelé “graphe orienté” (que l'on vient de présenter). Ce type de données contient les sommets et les arcs (ainsi que leurs données associées), et possède différentes opérations :

- ajouter ou de supprimer un arc ou un sommet.
- tester si un sommet est successeur d'un autre,
- lister tous les sommets successeurs d'un sommet,
- etc.

Observation

Le type abstrait “graphe” permet de donner une description d'une structure de données non linéaire et pouvant posséder des cycles. Ce type de données abstrait est donc la généralisation de toutes les structures de données possibles en mémoire d'un ordinateur.

Passage à la structure de données

Remarque : l'implémentation dépend des opérations à réaliser

Le but du type de données abstrait "graphe" n'est pas simplement de stocker des données, mais plutôt de répondre efficacement à des questions précises sur sa structure. On doit donc choisir une implémentation de manière à pouvoir répondre efficacement aux questions posées.

Par exemple, pour trouver rapidement les plus courts chemins, il n'est pas possible de stocker tous les chemins du graphe dans la structure de données pour les comparer, car ils peuvent être en nombre exponentiel.

On peut distinguer deux cas, suivants que les données sont :

- dynamiques (ajout et suppression de sommets/arcs fréquents). Par exemple, si le graphe représente un réseau social, alors les sommets et les arcs changent relativement beaucoup.
- peu dynamiques (c'est le plus fréquent). Par exemple, si les sommets représentent des stations de métro, cet ensemble change peu.

Implémentation

Quels points d'accès ?

Dans une liste chaînée, il suffit d'avoir un accès au premier élément de la liste, pour pouvoir ensuite atteindre tous les autres éléments (pareil pour les arbres). Dans le cas d'un graphe, on ne peut pas se contenter d'une entrée unique car certains sommets sont inaccessibles à partir d'autres.

Plusieurs implémentations possibles

Dans un graphe, il faut pouvoir accéder directement à chaque sommet. On va donc utiliser une structure (matrice, tableau, liste...) pointant sur chaque sommet du graphe. Il existe deux implémentations classiques :

- Matrice d'adjacence.
- Liste d'adjacence.

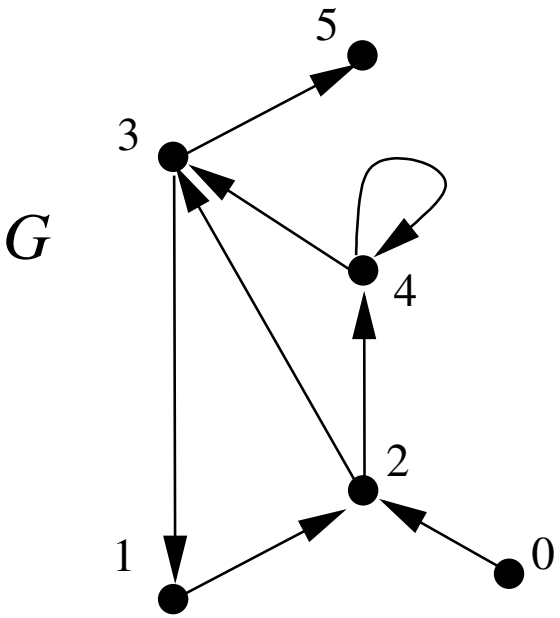
Il existe d'autres implémentations (matrice d'incidence, liste d'arcs) qui peuvent être préférables lors de l'implémentation de certains algorithmes.

Implémentation par matrice d'adjacence

Matrice d'adjacence

Une matrice d'adjacence est une matrice où chaque case (i,j) représente un emplacement possible pour un arc entre les sommets (s_i, s_j). Selon les données à stocker, on peut avoir dans chaque case :

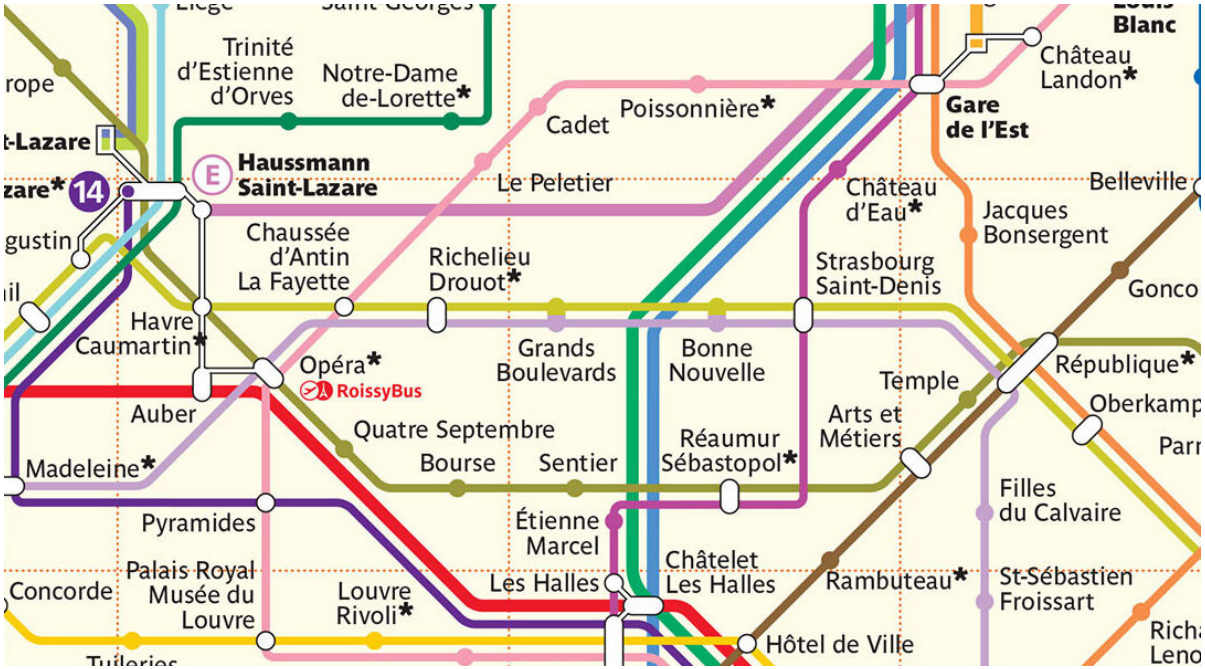
- un booléen qui a pour valeur `vrai` si l'arc existe, et `faux` sinon.
- un `struct arc` permettant de stocker toutes les données de l'arc.



Matrice d'adjacence de G :

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

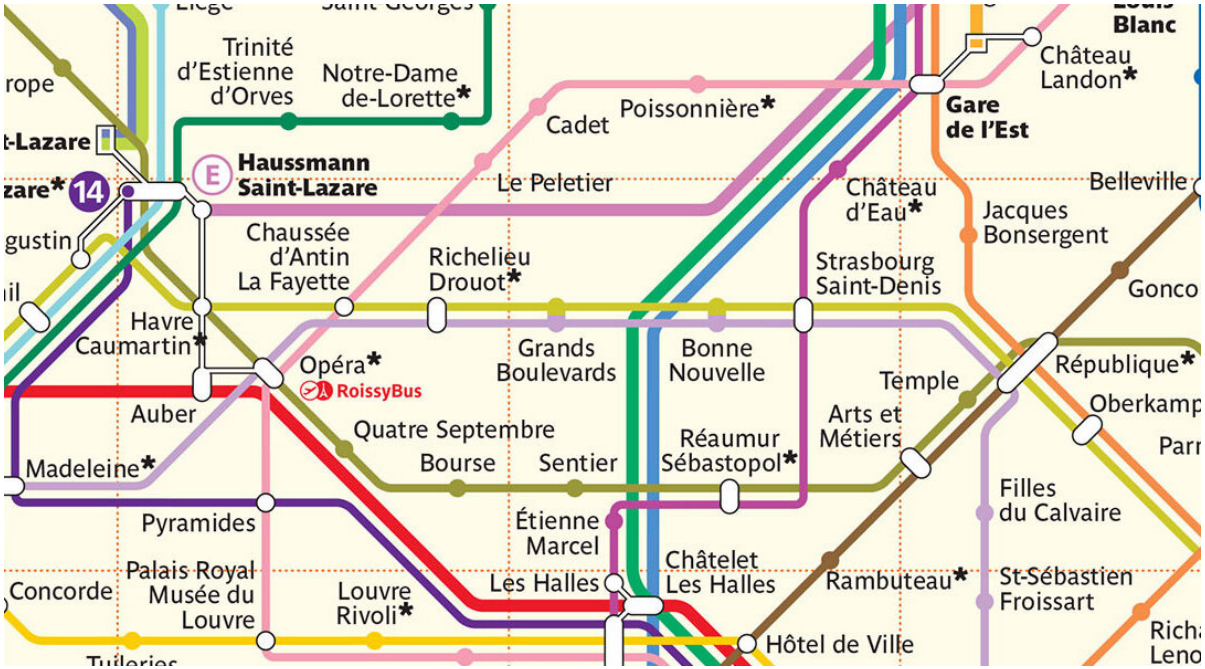
Exemple d'un plan de métro




```c
typedef struct arc {
    int depart;          // indice de la station de depart
    int arrivee;         // indice de la station d'arrivee
    float duree;         // duree du trajet entre les stations
    char* ligne;         // nom de la ligne (ou numero)
} Arc;

typedef struct matriceA {
    Arc** matrice;       // matrice d'adjacence
    char** stations;     // tableau des noms de stations
}
```

Exemple d'un plan de métro



Remarque :

Pour le plan de métro parisien, cette structure ne convient pas totalement car deux stations peuvent être reliés sur des lignes différentes. Une solution serait de stocker un tableau d'arcs dans chaque case :

```
1  typedef struct matriceA {
2      Arc*** matrice;          // matrice d'adjacence
3      char** stations;         // tableau des noms de stations
4  }
```

Implémentation par matrice d'adjacence

Avantages

Cette implémentation possède des avantages :

- Tester si s_i est successeur/prédécesseur de s_j est en

Implémentation par matrice d'adjacence

Avantages

Cette implémentation possède des avantages :

- Tester si s_i est successeur/prédécesseur de s_j est en $\Theta(1)$.
- Lister les successeurs (ou prédécesseurs) de s_i est en

Implémentation par matrice d'adjacence

Avantages

Cette implémentation possède des avantages :

- Tester si s_i est successeur/prédécesseur de s_j est en $\Theta(1)$.
- Lister les successeurs (ou prédécesseurs) de s_i est en $\Theta(n)$.

Défauts

Cette implémentation possède de nombreux défauts :

Implémentation par matrice d'adjacence

Avantages

Cette implémentation possède des avantages :

- Tester si s_i est successeur/prédécesseur de s_j est en $\Theta(1)$.
- Lister les successeurs (ou prédécesseurs) de s_i est en $\Theta(n)$.

Défauts

Cette implémentation possède de nombreux défauts :

- n^2 cases en mémoire, même pour les matrices creuses (c-à-d contenant beaucoup de zéro). Pas très efficace auand $m << n^2$.

Implémentation par matrice d'adjacence

Avantages

Cette implémentation possède des avantages :

- Tester si s_i est successeur/prédécesseur de s_j est en $\Theta(1)$.
- Lister les successeurs (ou prédécesseurs) de s_i est en $\Theta(n)$.

Défauts

Cette implémentation possède de nombreux défauts :

- n^2 cases en mémoire, même pour les matrices creuses (c-à-d contenant beaucoup de zéro). Pas très efficace auand $m << n^2$.
- Les successeurs de s_i sont donnés par la ligne i de la matrice et cette ligne contient toujours n cases, même pour les sommets avec peu de successeurs.

Cas d'utilisation

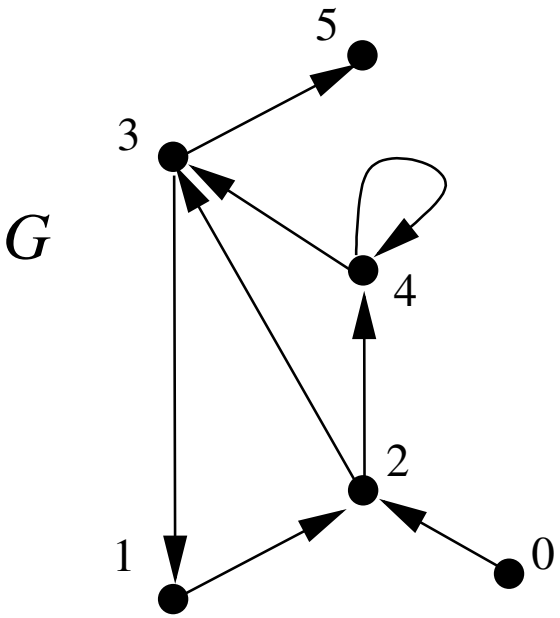
On privilégiera les matrices d'adjacence pour les petits graphes ou pour des graphes denses (pas creux), ou bien lorsque tester "s_i successeur/prédécesseur de s_j" est souvent réalisé.

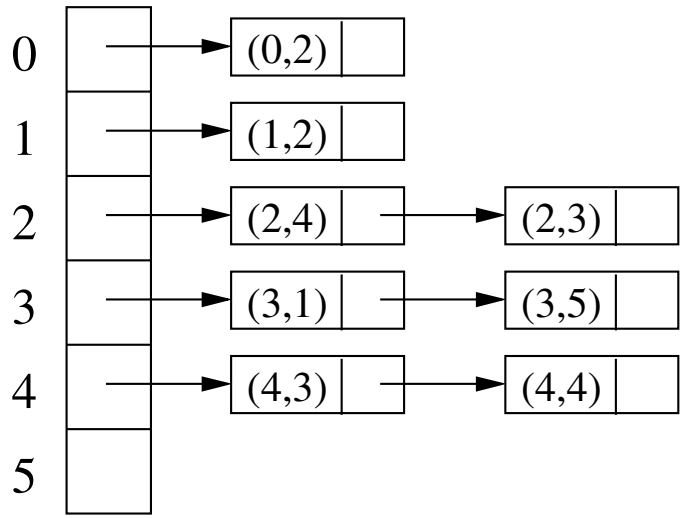
Implémentation par liste d'adjacence

Liste d'adjacence

Une liste d'adjacence est un tableau dont chaque case correspond à un sommet, et contient une liste chaînée de ses successeurs. Selon les données à stocker, on peut avoir :

- un `struct sommet` dans chaque case du tableau, pour stocker toutes informations relatives au sommet.
- des listes chaînées de `struct arc`, pour pouvoir stocker toutes les données relatives aux arcs.





Implémentation par liste d'adjacence

```
1  typedef struct arc {
2      int i;           // indice du sommet de depart
3      int j;           // indice du sommet d'arrivee
4      float poids;     // donnee optionnelle
5      char* nom;       // donnee optionnelle
6  } Arc;
7
8  typedef struct elementListeA{
9      Arc* a;
10     struct elementListeA* suiv;
11 } ElementListeA;
12
13 typedef ElementListeA* ListeA;
14
15 typedef struct sommet {
16     int num;         // indice dans le tableau des sommets
17     float x,y;       // donnee optionnelle
18     ListeA L_adj;    // liste des arcs sortant de ce sommet
19     int nbA ;        // nombre d'arcs sortants
20 } Sommet ;
21
22 typedef struct graphe {
23     int n;           // nombre de sommets
24     int m;           // nombre d'arcs
25     Sommet** tabS;   // tableau de pointeurs sur sommets
26 } Graphe ;
```

Implémentation par liste d'adjacence

Avantages

- Lister les successeurs de s_i est en

Implémentation par liste d'adjacence

Avantages

- Lister les successeurs de s_i est en $O(n)$.
- Pas d'espace mémoire inutilisé (n cases "sommets" et m cases "arcs").

Défauts

- Tester si s_i est successeur de s_j est en $O(n)$. En pratique, ce n'est pas très grave, car on cherche souvent à lister tous les successeurs d'un sommet (même complexité).
- Tester si s_i est prédécesseur de s_j est en $O(n)$, et lister tous les prédécesseurs de s_j est en $O(n+m)$. En effet, il faut parcourir tous les arcs pour connaître tous les prédécesseurs d'un sommet.

Remarque : Si on a réellement besoin de connaître les prédécesseurs, on peut stocker dans chaque sommet la liste chaînée de ses prédécesseurs. Dans ce cas, chaque arc est stockée deux fois, ce qui ne change pas la complexité spatiale, et lister les prédécesseurs devient en $O(n)$.

Cas d'utilisation

On privilégiera cette implémentation pour les grands graphes pour réduire l'espace mémoire et lister rapidement les successeurs d'un sommet.

Graphe non orienté (objet mathématique - théorie des graphes)

Quand on l'enlève l'orientation des arcs, on obtient ce que l'on appelle un graphe non orienté.

Définitions et terminologie

Un graphe non orienté est un couple $G = (S, A)$ où :

- S est un ensemble de n éléments appelés *sommets*.
- A est un ensemble de m paires non orientées de $S \times S$, appelés *arêtes*. Une arête peut être notée $\{s_i, s_j\}$ ou $\{s_j, s_i\}$ indifféremment.

Dans un graphe non orienté :

- un chemin est appelé une *chaîne*.
- un circuit est appelé un *cycle*.
- deux sommets liés par une arête sont dits *voisins* ou *adjacents* (pas de notions de successeur, prédécesseur, descendant et ascendant ici).
- une arrête $a = \{s_i, s_j\}$ est dite *incidente* aux sommets s_i et s_j.
- deux sommets sont dits *inaccessibles* l'un de l'autre s'il n'existe aucune chaîne entre les deux.

Connexité et arborescence

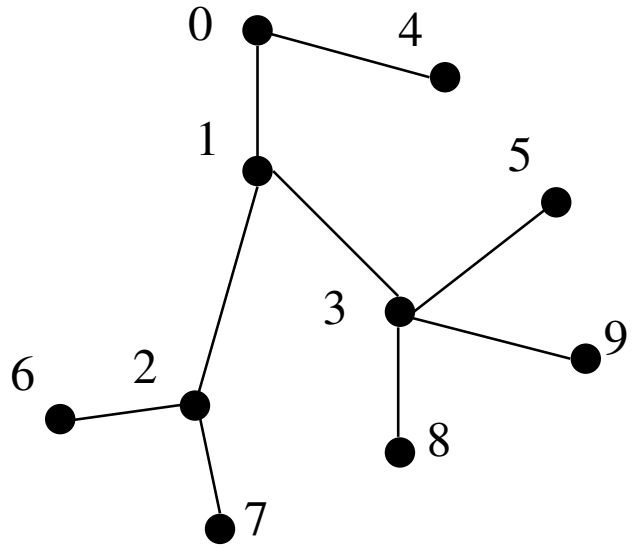
Définition : graphe connexe

Un graphe est dit *connexe* s'il n'existe aucune paire de sommets inaccessibles l'un de le l'autre. Autrement dit, un graphe est connexe s'il existe une chaîne entre toute paire de sommets du graphe.

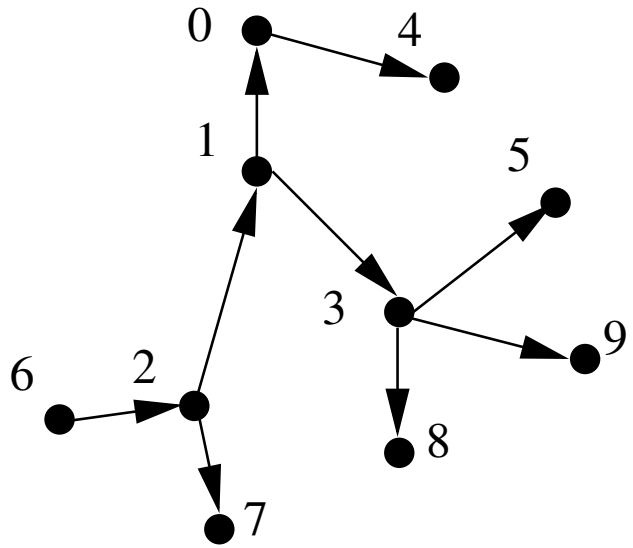
Arbre et arborescence

Un *arbre* est un graphe connexe sans cycle. Lorsque l'on choisit un sommet r, et que l'on oriente les arrêtes de r vers les autres, on obtient un graphe orienté dont r est l'unique racine (tous les sommets sont accessibles depuis r). Cet arbre orienté est appelé une *arborescence*, enracinée en r.

Arbre :



Arborescence enracinée en 6 :



Implémentation des graphes non orientés

Codage par matrice d'adjacence

Le principe du codage par une matrice (pour les graphes orientés) peut s'appliquer au cas non orienté. Dans ce cas, la matrice obtenue est symétrique (on peut alors se contenter d'une matrice triangulaire).

Codage par liste d'adjacence

Il n'existe pas directement d'implémentation d'un graphe non orienté par une structure avec des pointeurs : en effet, les pointeurs sont par nature "orientés". On peut quand même coder un graphe non orienté avec des pointeurs en utilisant une sorte d'équivalence entre graphe orienté et non orienté : une arête entre deux sommets s_i et s_j correspond à mettre à la fois un arc (s_i, s_j) et un arc (s_j, s_i). Dans cette implémentation, il y a deux fois plus de "cases" que dans le cas orienté, mais les complexités des opérations restent inchangées.

Remarque : s'il y a des données à stocker sur les arêtes, il est important que les deux cases d'une même arête pointent sur les mêmes données.