Stratégies et Techniques de Résolution de Problèmes

Chapitre II-1 : Graphes et Algorithmes de Parcours

S7 - ECL – 2A – MI

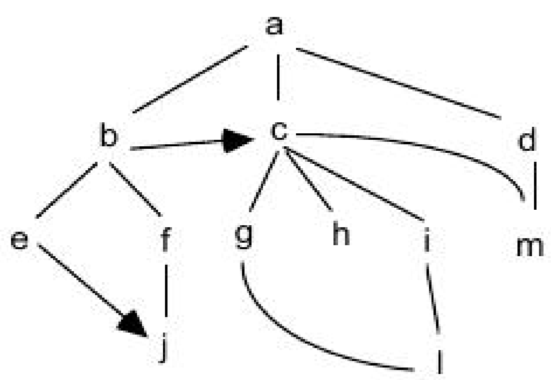
2022-2023

Alexandre Saidi

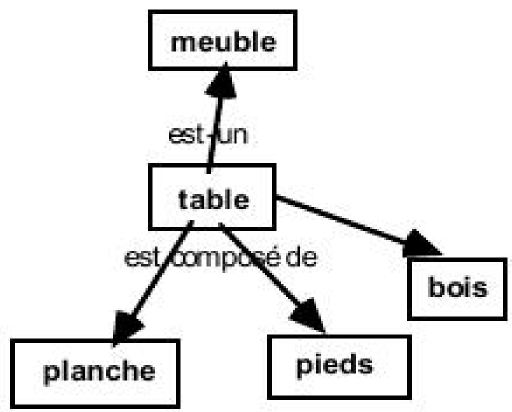
I- Introduction aux Graphes

- Moyen de structuration et de représentation (hiérarchie, composition, structure, modèle, etc.)
- Outil de modélisation de problèmes
- Une généralisation des arbres

Exemples



un graphe représentant des liens entre différents noeuds (villes)



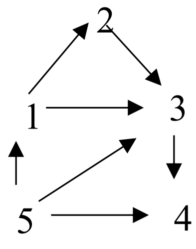
un graphe representant des liens d'héritage et de composition

II- Graphes : quelques notions

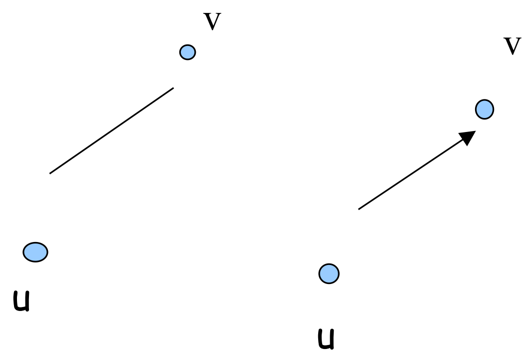
- Un graphe **G = (V, E)**

  V : ensemble de **nœuds** (*vertex*)

  E $\subseteq$ (V x V) : ensemble d'**arêtes** ou **arcs** (*edges*)



Un Digraphe



- Chaque **arc** = une paire (v,w) $\in$ **E**, v,w $\in$ V

  ≡ **Arête**, *adjacents* ,*successeur* (resp. *Prédécesseur*)

- **Graphe orienté (digraphe)** ≡ *directed graph*

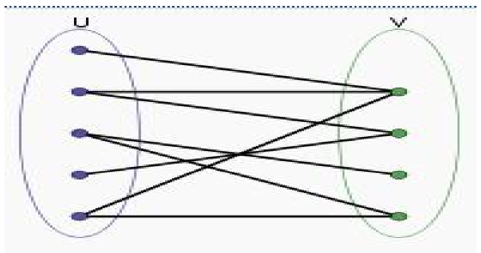
- **Poids** (weight) : valeur d'un arc/arête

  ≡ Graphe *valué (pondéré)* : *les arcs / arêtes portent un poids : temps, distance, prix,…*

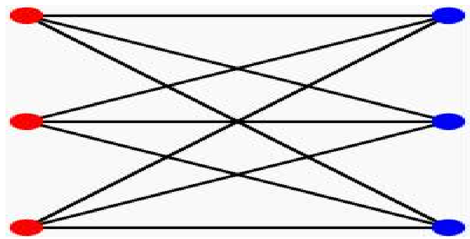
- **Chemin** (branche, path)
- Nœud *accessible* depuis un nœud X *si ...*
- Longueur d'un chemin contenant N nœuds = nombre d'arcs = N - 1
- Un chemin non nul entre (v,w) dans un graphe orienté est un **circuit** (un **cycle**) si v = w.
- Si le graphe contient un arc **(v,v),** le chemin (v,v) est une **boucle** (***loop)***

concerne 1 seul noeud

- Graphe **biparti**, **biparti** complet (ou **biclique**)



graphe biparti



biparti complet (tout est lié à tout)

- Un graphe orienté est **acyclique** s'il n'a pas de cycle ;

  --> Un **DAG** : *directed acyclic graph.*

- Un graphe non orienté est **connexe** s'il y a un chemin entre toute paire de nœuds

- Un graphe orienté et connexe est appelé **fortement connexe** (*Complet* ou *Dense*) si $|E| \cong O(|V|^2)$

  → cf. un réseau d'aéroports / ferroviaire où toute paire de villes est desservie

  --> Un graphe peut être peu connexe (peu dense)

- **Réseau**, **Arbre** : graphe connexe sans boucle (acyclique)

Ordre dans un graphe :

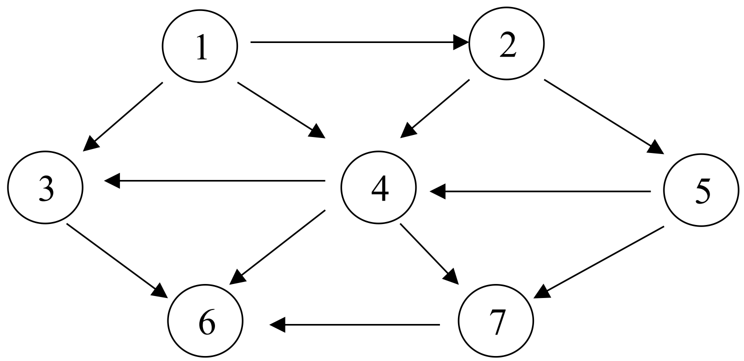
- La relation d'ordre binaire '$\leq$' est réflexive, antisymétrique et transitive sur un ensemble de nœuds d'un graphe.
- L'ensemble S est *totalement* ordonné sous la relation $\Re$ si tous ses couples d'éléments sont ordonnés : $(\forall X,Y \in S, (X \Re Y) \vee (Y \Re X))$.

  --> La relation $\Re$ est alors appelé une **relation d'ordre total**.

  --> Par Exemple, '$\leq$' est un ordre total sur les entiers naturels.
- **Treillis** : est un arbre (G connexe, acyclique) avec une relation d'ordre totale sur les nœuds.

II.1- Exemple (de graphe)

- Notion du chemin (et le chemin le plus court) :



Graphe G1 : graphe orienté du trafic

Dans un graphe similaire représentant un réseau de rues, si chaque intersection concernait par exemple 4 rues avec des rues à double sens :

--> on aurait **|E|** $\cong$ **4|V|**.

III- Matrice d'adjacence

Pour simplifier, la matrice d'adjacence d'un graphe **G=(V,E)** avec $|V| = n$ sommets est une matrice A de taille ($n \times n$) avec "1" dans la case A_{ij} si les sommets i et j sont connectés.

Plus formellement, la case non-diagonale A_{ij} contient le nombre d'arêtes liant le sommet i au sommet j.

L'élément diagonal A_{ii} est le nombre de boucles au sommet i.

Pour des graphes simples, ce nombre est donc toujours égal à 0 ou 1.

Propriétés de la matrice d'adjacence :

- Un graphe **G(V,E)** peut donc être représenté par sa matrice d'adjacence.
- Avec dans chaque case une valeur booléenne (Vrai / Faux) ou une **pondération** (valeur) :
  - Espace $O(|V|^2)$
  - Convient à un graphe **dense** (i.e. $|E| \cong O(|V|^2)$)

- Si beaucoup de "faux" (ou zéro) ⇒ un graphe **peu dense** (*sparse*)

Exemple :

Pour 3000 carrefours routiers, matrice de 9000000 éléments avec beaucoup de zéro / faux (lorsque 2 carrefours ne sont pas reliés !)

La représentation précédente regroupe les deux cas :

- <u>Graphe non valué</u> (non pondéré) : une matrice de booléens.
- <u>Graphe valué</u> (pondéré) : chaque case contient une valeur qui représente la valeur de l'arc (arête) reliant les deux nœuds.

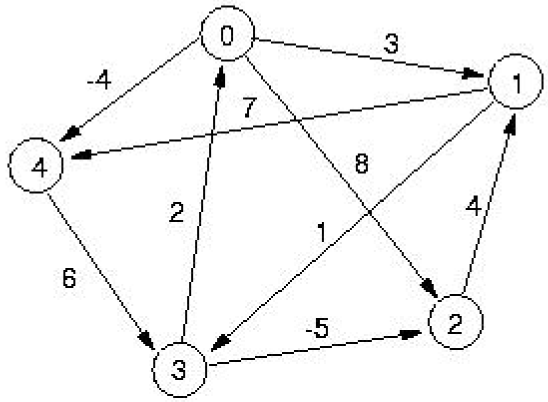
--> Une constante prédéfinie (par exemple -1) représente l'absence de lien.

--> L'orientation des arêtes est exprimée par le contenu des intersections des lignes et des colonnes.

Exemple de matrice :

	A	B	C	D	E	F	G
A							
B			V				
C		F					
D		15			3		
E				-1			
F							
G							

Exemple d'un graphe orienté pondéré :



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

Les cases vides : infini

Si matrice creuse, on perd de l'espace → représentation dynamique ../..

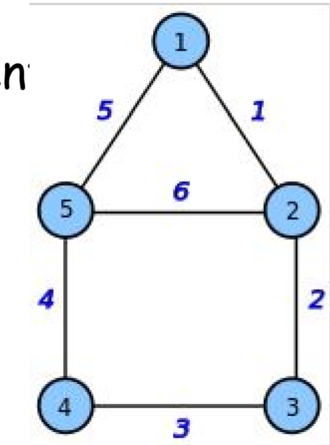
IV- *Matrice d'incidence*

Pour un graphe *G*=(V,E) avec |V| = *n* sommets et *p=|E|* arêtes (arcs), la **matrice d'incidence** B(*G*) est une matrice (*n x p*).

☞Cette matrice est différente pour un graphe orienté.

<u>Cas du graphe non orienté G=(V,E) :</u>

La matrice B appelée "matrice d'incidence sommets-arêtes" contien



valeurs dans les cases B_{ij} comme suit :

1 si le sommet v_i est une extrémité de l'arête x_j
2 si l'arête x_j est une boucle sur v_i
0 sinon

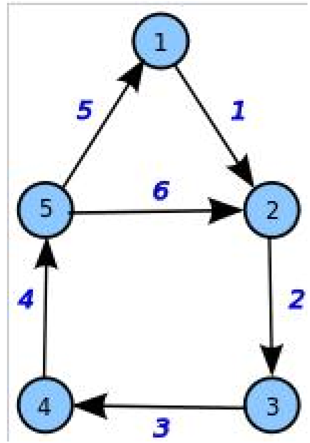
<u>Exemple :</u>

le sommet 1 connecte les arête .	1 et 5
le sommet 2 connecte les arêtes	1, 2 et 6
le sommet 3	2 et 3
le sommet 4	3 et 4
le sommet 5	4, 5 et 6

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}
\end{array}
$$

Cas du graphe orienté G=(V, A) :

La matrice B(G) appelée "matrice d'incidence sommets-arcs"



Contient dans les cases B_{ij} :

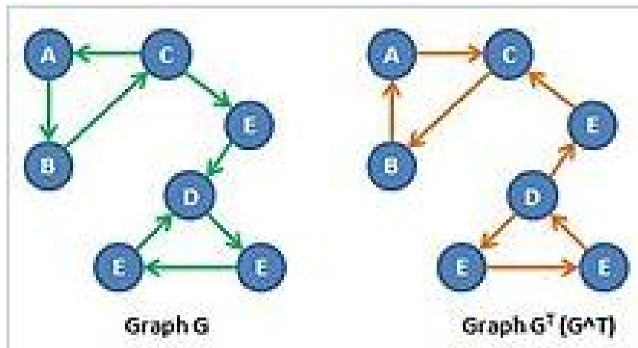
- -1 si l'arc x_j sort du sommet v_i
- 1 si l'arc x_j entre dans le sommet v_i
- 0 sinon

le sommet 1 est connecté aux arcs 1 (qui sort) et 5 (qui entre)
le sommet 2 est connecté aux arcs 1 et 6 (entrants)et 2 (sortant)
le sommet 3 est connecté aux des arcs 2 (entre) et 3 (sort)
le sommet 4 est l'aboutissement des arcs 3 (entre) et 4 (sort)
le sommet 5 est connecté aux arcs 4 (qui entre), 5 et 6 (sortants)

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{pmatrix}
-1 & 0 & 0 & 0 & 1 & 0 \\
1 & -1 & 0 & 0 & 0 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & -1
\end{pmatrix}
\end{array}
$$

Propriété de la matrice d'incidence :

La transposée de la matrice d'incidence d'un graphe orienté G est la matrice d'incidence du G^T = le graphe transposé de G (<u>arcs inversés</u>).



Graphe transposé :

Le **graphe transposé** G^T ou le "graphe inverse" d'un graphe orienté $G=(V,A)$ s'obtient en conservant tous les nœuds de G et en inversant tous les arcs A.

$$G^T=(V, A^T) \text{ avec } A^T= =\{(y,x) \mid (x,y) \in A\}.$$

Propriétés

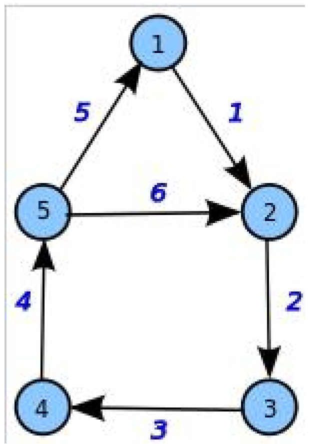
- La transposé de la transposée d'un graphe G est le graphe G ;
- La matrice d'incidence du graphe transposé est la transposée de la matrice d'incidence du graphe original.
- Un graphe égal à sa transposée est dit **symétrique**.

Applications :

Certains algorithmes utilisent la transposée du graphe d'entrée, par exemple l'algorithme de *Kosaraju* (calcul des composantes fortement connexes d'un graphe orienté) effectue un parcours en profondeur du graphe et de sa transposée.

V- Matrice Laplacienne

La matrice laplacienne **K(G)** d'un graphe G est une matrice ($n \times n$) où n = |V| est le nombre de sommets.



Dans K(G), on a :

- ✓ Le **degré** des sommets du graphe sur la diagonale,
- ✓ En ligne i, colonne j on a :
  - -1 si les sommets i et j sont liés
  - 0 s'ils ne le sont pas.

Si B(G) est la matrice d'incidence d'un graphe orienté G, on peut en déduire la matrice laplacienne K(G) en multipliant B(G) par sa transposée $B(G)^T$:

$$K(G)=B(G)\,B(G)^T$$

Pour le graphe orienté ci-dessus, on a :

$$K(G) = \begin{pmatrix} -1 & 0 & 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & 0 & -1 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ -1 & -1 & 0 & -1 & 3 \end{pmatrix}$$

Matrice des degrés : une dernière matrice utilisée dans les graphes.

Il s'agit de D(G) une matrice diagonale (n x n) où en a en ligne i et en colonne i le degré du sommet i, tous les autres coefficients valent 0.

Liens entre les 3 matrices (adj (A), inc (B), deg (D)) :

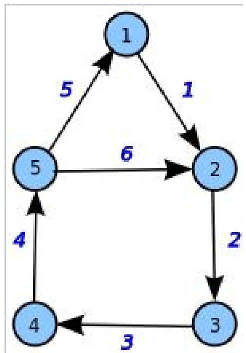
Si B(G) est la matrice d'incidence d'un graphe non orienté G, si A(G) est sa matrice d'adjacence et si D(G) est sa matrice des degrés, alors :

$$A(G)+D(G)=B(G)^T B(G).$$

$$A(G) = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}; D(G) = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

Pour le graphe orienté ci-dessus :

$$A(G)+D(G) = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 & 0 & 1 \\ 1 & 3 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 1 & 1 & 0 & 1 & 3 \end{pmatrix}$$



VI- Structure de données pour représenter un graphe

On présente ici la matrice d'adjacence.

On peut implanter les graphes de diverses manières :

- Par une matrice carrée d'adjacences
- Par un tableau / liste principal + tableau / liste des adjacents
- Par un tableau principal + listes des adjacents regroupés dans une même structure

--> Selon la nature du graphe (orienté ou non, valué ou non), les structures peuvent être plus ou moins complexes.

VI.1- Représentation dyn. par les listes d'adjacences

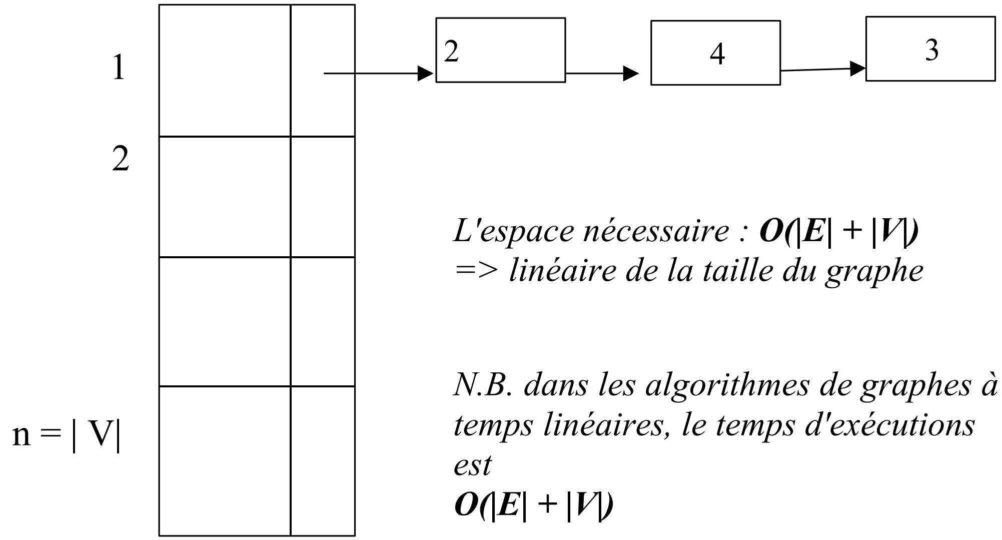
Une structure de données plus dynamique (adaptée pour un graphe peu dense) :

Pour un graphe non orienté :

Chaque arête *(u, v)* apparaît dans deux listes d'adjacences

--> l'espace nécessaire est **double**.

--> Représentation la plus utilisée.



*L'espace nécessaire : **O(|E| + |V|)***
=> linéaire de la taille du graphe

N.B. dans les algorithmes de graphes à temps linéaires, le temps d'exécutions est
O(|E| + |V|)

Par contre, cette repr. n'est pas directement possible dans tout langage de prog.

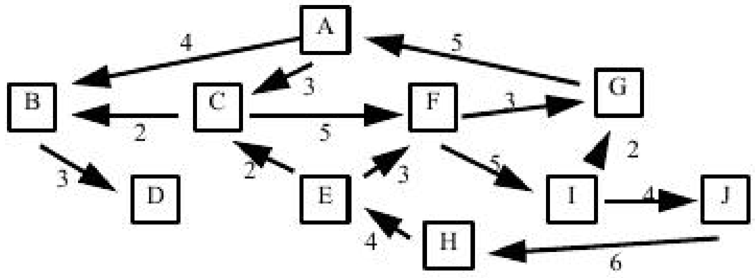
Remarque : le tableau principal (le premier tableau) peut être une liste.

De même, la liste des adjacents peut être un tableau (statique / dynamique).

--> En Python, on utilisera une liste principale et des tableaux secondaires.

VI.2- Représentation alternative Hétérogène

Le graphe G=(V,A) :



Représenté par deux tableaux / Listes :

- **V pour les nœuds (toutes infos sur noeuds)**
- *A* pour les arcs + valeurs (pondérations)

Dans le tableau secondaire (2e tab. droite),

la première ligne indique (1,2,4) :

--> il y a un arc du nœud "A" (indice 1 dans V)

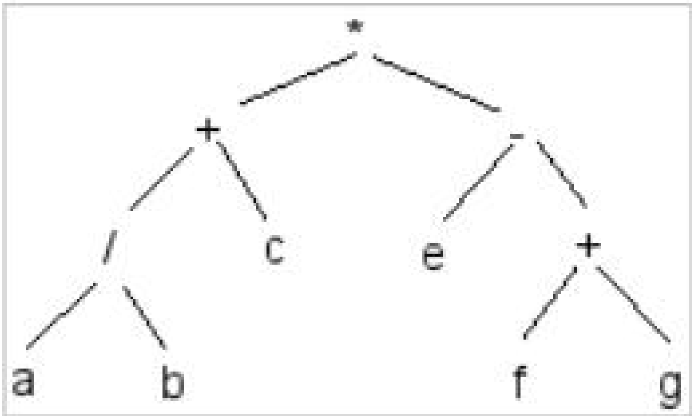
vers le noeud "B" (indice 2) avec un coût de 4.

	Noeuds
1	A
2	B
3	C
4	D
5	E
	F

début	*fin*	*coût*
1	2	4
1	3	3
2	4	3
3	2	2

VI.3- Un autre exemple de représentation Hétérogène (Python)

Cas binaire :



indice	Noeuds	Fils
0	'*'	(1,6)
1	'+'	(2,5)
2	'/'	(3,4)
3	'a'	(None, None)
4	'b'	(None, None)
5	'c'	(None, None)
6	'-'	(7,8)
7	'e'	(None, None)
8	'+'	(9,10)
9	'f'	(None, None)
10	'g'	(None, None)

→ Il est possible de représenter un graphe par une liste de listes en Python.

↪ Voir aussi le TDA Graphe en annexes.

VII- Algorithmes notables de parcours de graphes (récursifs / itératifs)

Deux types majeurs de parcours :

1- **En profondeur**

Avantages en inconvénients

2- **En largeur**

Avantages et inconvénients

⇒ Il y a également des parcours **ad-hoc**

Remarque : selon le "moment" où l'on traite l'information d'un nœud, on a différents types de traitements : *pré-ordre, post-ordre* et *mi-ordre*.

Remarque : on se place dans le cas d'un parcours en *pré-ordre (préfixé)*.

⇨ Des deux types de parcours notables, on peut obtenir des stratégies variées :

(A, A*, …).

→ Voir plus loin, en particulier dans le cadre d'un parcours en **largeur**.

VIII- Le principe du parcours récursif en profondeur (pré-ordre)

- Traiter chaque nœud puis traiter son premier adjacent récursivement avant de traiter les autres adjacents (cf. BE Cavalier)
- Éviter de retraiter un nœud en **marquant** les nœuds visités

- Ce parcours est semblable au parcours en profondeur des arbres.
- Ce parcours est en général assez performant mais si le graphe contient un cycle "à gauche", alors aucune réponse ne pourra être produite.

Parcours récursif en profondeur avec marquage :

```
Procédure profondeur (G, nœud) =
Début
   Si noeud != NIL alors              # NIL = None / NULL / sans information / .....
      marquer(nœud);
      traiter(nœud);                     //traitement  quelconque
      Pour X dans adjacents(G, noeud)
         Si X n'est pas marqué alors
               profondeur(G, X);
         Fin si;
      Fin pour;
   Fin si;
Fin profondeur ;
```

N.B. : Ci-dessus, pour un graphe G non_vide, **nœud** représente le nœud actuellement référencé (comme la racine dans un arbre) donnant accès aux informations du nœud, ses adjacents, etc.

- Les mécanismes de **marquage** :

  - Marquage à l'intérieur du nœud

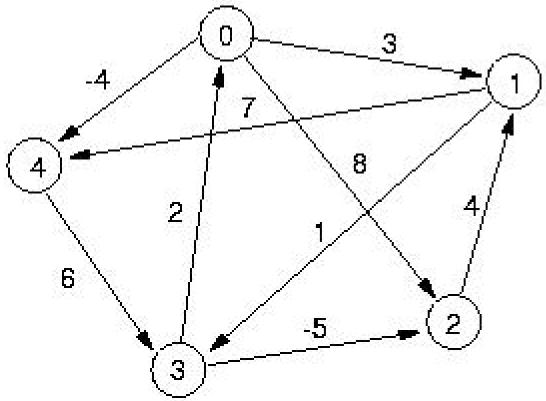
  - Marquage par une structure de données externe au graphe (un tableau, ...)

Une autre version :

```
Procédure profondeur (G , nœud) =
Début
  Si noeud != NIL ET nœud n'est pas marqué alors
    marquer(nœud);
    traiter(nœud);                // traitement  quelconque
    Pour X dans adjacents(G, nœud)
      profondeur(G, X);
    Fin pour;
  Fin si;
Fin profondeur ;
```

Dans cette version, plus besoin de tester le marquage avant l'appel récursif.

Exemple :

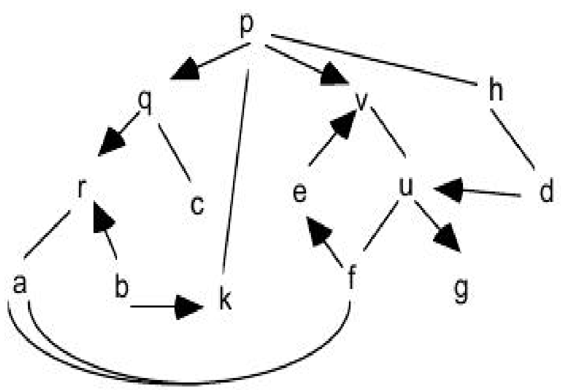


Pour ce graphe, un parcours en profondeur depuis le nœud 0 (en balayant les successeurs de gauche à droite) visiterait les nœuds dans cet ordre :

0 → 4 → 3 → 2 → 1

N.B. : le trajet partiel 0 → 4 → 3 → 0 n'a pas lieu puisque le noeud 0 a été marqué comme étant déjà visité.

VIII.1- Trace de parcours en profondeur



- Trace du parcours en profondeur de **p** à **u**

  profondeur(p) = profondeur(q) = profondeur(r) = profondeur(a) = profondeur(f) = profondeur(e) = profondeur(v) = profondeur(u)

Principe de l'algorithme itératif en Profondeur :

```
Procédure profondeur_iteratif(G, noeud)
   déclarer Pile=vide
   empiler(noeud)
   Tant que Pile != vide
      Noeud ← dépiler(Pile)
      Si NON  est_marqué(X) Alors      # parfois, un noeud peut se trouver 2 fois dans la Pile (voir trace ci-dessous).
         marquer et traiter(noeud)
         Pour X dans adjacents(G, noeud)
            Si  NON  est_marqué(X) Alors empiler(Pile, X)   Fin Si            – empiler dans le désordre
         fin pour
      FinSi
   Fin Tant que
Fin  profondeur_iteratif
```

Trace de la pile de **p** cette fois à **d**:

- Trace de parcours en profondeur de **p** à **d** (on souligne si traité) :

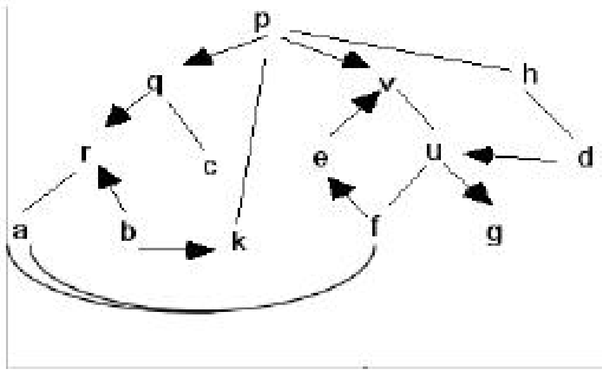
  $[\underline{p}] \to [h,v,k,\underline{q}] \to [h,v,k,c,\underline{r}] \to [h,v,k,c,\underline{r}] \to [h,v,k,c,\underline{a}] \to [h,v,k,c,\underline{f}]$

  $\to [h,v,k,c,e,\underline{u}] \to [h,v,k,c,e,g,\underline{v}]$ ➙ v se trouve 2 fois dans la pile sans être marqué

  $\to [h,v,k,c,e,\underline{g}] \to [h,v,k,c,\underline{e}] \to [h,v,k,\underline{c}] \to [h,v,\underline{k}] \to [h,\underline{v}]$

  ➙ **v** : la 2e fois ! déjà traité. puis

  $\to [\underline{h}] \to [\underline{d}] \to []$



IX- Le principe de parcours en largeur

Traitement par niveau :

- traiter chaque nœud
- Puis traiter chacun de ses successeurs avant de traiter les successeurs du prochain niveau.

- Parcours moins performant que le précédent et plus gourmand en mémoire.
- S'il existe un cycle dans le graphe, des réponses seront néanmoins produites.
- Ce parcours s'adapte mieux à une solution itérative :

  → On utilise **une file d'attente** pour la construction de la liste des nœuds à traiter.

IX.1- Le Type (TDA) File

- A propos des TDAs : outil de spécification algébrique de type.
- Exemple : le TDA d'une file (d'attente)

File_vide :		$\rightarrow$ File	
Enfiler :	Élément x File	$\rightarrow$ File	# est aussi appelé "constructeur"
Défiler :	File	/$\rightarrow$ File	# différent des utilisations habituelles
Sommet :	File	/$\rightarrow$ Élément	
Est_file_vide :	File	$\rightarrow$ Booléen	

Pré-conditions : pour f : File;

Défiler(f) : non Est_file_vide(f)

Sommet(f) : non Est_file_vide(f)

Axiomes : pour f : File; e : Élément

Sommet(Enfiler(e, File_vide))=e # quand on enfile e, e se retrouve à la fin de la file

Sommet(Enfiler(e, f))=Sommet(f) # il faut donc aller à la fin de la file et renvoyer le dernier

Est_file_vide(File_vide) =vrai

Est_file_vide(Enfiler(e, f))=faux,

Ce TDA donne lieu à une classe "File" qui doit respecter la spécification.

IX.2- L'algorithme récursif de parcours en largeur

Cet algorithme fonctionne pour les graphes connexes.

Voir la suite pour les autres.

```
File : file d'attente =File_vide;
Procédure largeur_récursif (G, nœud) =
Début
   Si noeud != NIL  alors
      traiter(nœud);                # une opération quelconque
      Pour X dans adjacents(G, nœud);
               File=Enfiler(X, File);
      Fin pour;
      X=Sommet(File);
      File=Défiler(File);                     # car défiler ne renvoie pas un élément (cf. TDA File)
      largeur_récursif (G, X);
   Fin si;
Fin largeur_récursif ;
```

N.B. : Ici, pas de **marquage.**

→ en l'absence de marquage, certains nœuds sont traités **plusieurs fois.**

Cas de graphe connexe : on travaille avec une file d'attente

```
File : file d'attente =File_vide;
enfiler( la racine du graphe G, File)

Procédure largeur_récursif (File) =      # Si File est globalement visible, pas besoin de paramètre.
Début
   Si Non est_vide(File) alors
      X=Sommet(File);
      File=Défiler(File);                // car défiler ne renvoie pas un élément (cf. TDA File)
      traiter(nœud);                     //une opération quelconque
      Pour X dans adjacents(G,nœud);
               File=Enfiler(X, File);
      Fin pour;
      largeur_récursif (File);
   Fin si;
Fin largeur_récursif ;
```

Initialement, il faut enfiler le nœud de départ.

N.B. : voir la version Python du problème de la monnaie (1A).

On peut récupérer le chemin par le mécanisme **Coming-From (CF).**

```
File : file d'attente =File_vide;
enfiler(la racine du graphe G, File)
CF : tableau indicé par les nœuds initialisé à 0;
CF[Départ]=Départ
deja_vu_marquage = vide

Procédure chemin_largeur_récursif (File) =
Début
   Si Non est_vide(File) alors
      X=Sommet(File);
      File=Défiler(File);              // car défiler ne renvoie pas un élément (cf. TDA File)
      traiter(nœud);                   //une opération quelconque
      Pour X dans adjacents(G, nœud);
               Si X n'est pas dans deja_vu_marquage :
                  File=Enfiler(X, File);
                  CF[X]=nœud
      Fin pour;
      largeur_récursif (File);
   Fin si;
Fin largeur_récursif ;
```

IX.2.1- Implantation Python via un exemple

Exemple de recherche dans un labyrinthe par un parcours en largeur.

- ✓ Le graphe du labyrinthe est donné sous forme d'une matrice.
- ✓ On chercher un chemin entre une case départ et la case arrivée
- ✓ Le tableau **tab_voisins** permet de connaître les voisins d'une case (cf. BE1 AES)
- ✓ On utilise une file (FIFO) contenant les cases à exploiter (parcours en largeur)
- ✓ L'ensemble cases_deja_vues contient les cases déjà traitées (marquage)
- ✓ Dico_CF (coming from) est un dictionnaire contenant les couples X : Y où

  Pour aller à "X", on vient de "Y". Il permet de construire un trajet.
- ✓ La fonction **trouver_un_voisin_pour(X)** trouvera un voisin pour la case X.

```
def chemin_en_largeur_avec_trajet(graphe, case_depart, case_arrivee) :
    file_des_cases = [case_depart]
    cases_deja_vues={case_depart}
    dico_CF={}; dico_CF[case_depart]=case_depart          # un Dictionnaire Python

    # fonction utilitaire
    def chemin_aux() :
        if file_des_cases == [] :   return  False, _       # Pas de chemin
        case_actuelle= file_des_cases.pop(0)               # Prendre le 1er élément de la file
        if case_actuelle ==  arrivee :  return True, dico_CF
        for i in range(nb_voisins_possibles) :
            case_voisin=trouver_un_voisin_pour(case_actuelle)
            if  prometteur(case_voisin) and case_voisin not in cases_deja_vues :
                file_des_cases.append(case_voisin)          # ajout à la file
                cases_deja_vues.add(case_voisin)            # ajout à un ensemble par "add"
                dico_CF[case_voisin]= case_actuelle         # On va de case_actuelle à case_voisin
        return chemin_aux()

    return chemin_aux()
```

Voir plus loin pour une trace de la file dans un parcours en largeur.

Dans dico_CF, on aura les couples [départ : départ, C1 : C2, C2 : C3, ... Ci : Cj, ..., arrivee : Ck].

On reconstitue le trajet via ce dictionnaire en remontant depuis arrivée à ... départ.

IX.3- L'algorithme itératif de parcours en largeur

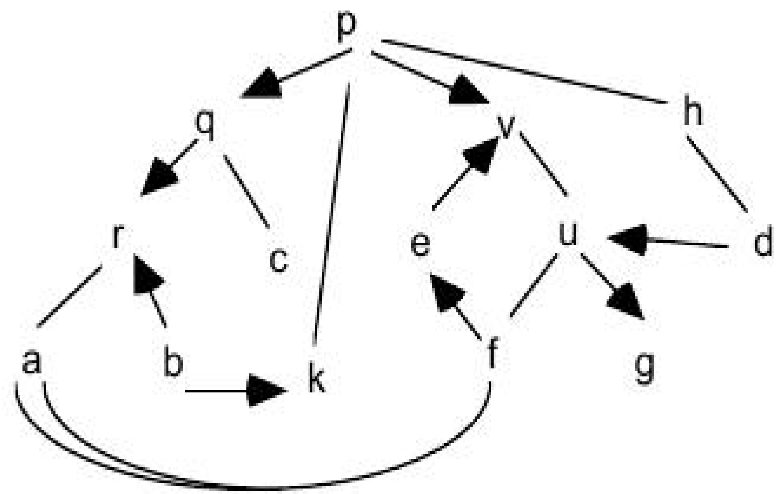
Ajouter le mécanisme de marquage.

```
File : file d'attente=File_vide;
Procédure largeur_itératif (G : graphe, noeud : Node)
    déclarer File=vide
    Début
      Si noeud != NIL  alors
          Enfiler(nœud, File);
      Fin si;
      Tant que Non est_vide(File)
          sommet=Sommet(File);
          File =Défiler(File);
          traiter(sommet);          // ou visiter(sommet)
          Pour X dans adjacents(G, sommet);
              File=Enfiler(X, File);
          Fin pour;
      Fin Tant que;
    Fin largeur_itératif ;
```

N.B. : version sans marquage

IX.3.1- Trace de parcours en largeur

- **Exemple** : différents parcours illustrés sur le graphe suivant.



<u>Trace de la version non marquée</u>.

Parcours en largeur de **p** à **u**

(noter le cycle P --> ... --> P ...) :

largeur(p) --> largeur(q) --> largeur(k) -->

largeur(v) --> largeur(h) --> largeur(r) -->

largeur(c) --> <u>largeur(p)</u> --> largeur(u)

L'évolution de la File lors de ce parcours (ajout tant que le sommet $\neq$u) :

[] [<u>p</u>] [<u>q</u>, k, v, h] [<u>k</u>, v, h, r, c] [<u>v</u>, h, r, c , p] [<u>h</u>, r, c , p, u]

[<u>r</u>, c , p, u, p, d] [<u>c</u>, p, u, p, d, a] [<u>p</u>, u, p, d, a] [<u>u</u>, p, d, a, q, k, v, h]

➥ Destination atteinte.

IX.4- L'algorithme itératif de parcours en largeur avec marquage

Ici, la File peut contenir des doublons qui ne seront pas traités une seconde fois.

```
File : file d'attente=File_vide;
Procédure largeur_itératif_marquage (G, noeud)
Début
    Si noeud != NIL Alors        Enfiler(nœud_courant(G), File);
    Fin si;
    Tant que Non est_vide(File)
        sommet=Sommet(File);
        File =Défiler(File);
        Si est_marqué(sommet) alors passer à l'itération suivante ;
        Sinon marquer(sommet);
        Fin si;
        traiter(sommet);        // ou visiter(sommet)
        Pour X dans adjacents(G, sommet);
            File=Enfiler(X, File);
        Fin pour;
    Fin Tant que;
Fin  largeur_itératif_marquage ;
```

IX.4.1- Une variante parcours en largeur itératif (avec marquage)

Une seconde version avec marquage :

--> on marque les nœuds (non marqués) avant de les placer dans la file.

La file ne contiendra pas de doublon.

```
File : file d'attente=File_vide;
Procédure premier_chemin_largeur_itératif (G, noeud)
Début
    Si noeud != NIL alors
        Enfiler(nœud, File);
        marquer(nœud);
    Fin si;
    Tant que Non est_vide(File)
        sommet=Sommet(File);
        File =Défiler(File);
        traiter(sommet);            // ou visiter(sommet)
        Pour X dans adjacents(G, sommet) tel que X non marqués
            File=Enfiler(X, File);
            marquer(X);
        Fin pour;
    Fin Tant que;
Fin premier_chemin_largeur_itératif ;
```

X- Applications des parcours de graphes

Rappel : pour éviter les boucles dans les algorithmes, on peut appliquer :

marquage des nœuds ou encore mémorisation du trajet.

X.1- Exemple simple : comptage du nombre de nœuds de G

```
Fonction taille(G, nœud) : entier =          // en profondeur
   nb_nœuds : entier=0;        ADJ : Liste(nœuds)
   Début
      Si noeud != NIL alors
         Si Non est_marqué(nœud)
         Alors    marquer(X);
                  nb_nœuds=nb_nœuds + 1;
                  ADJ=adjacents(G, nœud);
                  Pour X dans ADJ
                     nb_nœuds = nb_nœuds +taille(X);
                  Fin Pour;
         Fin si;
      Fin si;
      Retourne nb_nœuds ;
   Fin Taille;
```

X.2- Fonction recherche d'un Élément

La fonction de recherche de l'élément **X** dans un graphe connexe
→ renvoie le nœud trouvé sinon une réf. Vide (NIL):

```
Fonction recherche(X:  élément; G, nœud) renvoie un noeud  :     # En profondeur
  Début
    Si noeud = NIL       Alors       retourne  NIL  Fin si;     ,
    Si est_marqué(nœud)
    Alors     retourne  NIL;            // déjà visité --> on tourne en rond !
    Sinon
      Si (nœud = X) alors retourne noeud;
      Sinon
        marquer(nœud);
        ADJ=adjacents(G, nœud);
        Pour N dans ADJ
            noeud=recherche(X,G, N) ;
            Si (noeud != NIL ) alors retourne noeud;  Fin si;
        Fin Pour;
        Retourne  NIL
      Fin si;
    Fin si;
Fin recherche;
```

X.3- Recherche de chemin en Profondeur

- La fonction ***chemin*** entre deux nœuds dans un graphe qui produit un trajet s'il y a un chemin entre les nœuds.

- Le trajet=vide si pas de chemin entre ces deux nœuds.
- On suppose que le *nœud=Départ* (sinon, on s'y place d'abord).

```
Fonction chemin(Dép, Arr : nœud; G : Graphe) : le trajet =      // parcours en profondeur
trajet= VIDE;                                                   // trajet = une liste vide
Début
    Si est_vide(G) alors retourne VIDE; Fin si;                 // la liste trajet vide
    Si (Dép=Arr) alors retourne [Arr]; Fin si;                  // une liste contenant Arr
    marquer(Dép);
    ADJ=adjacents(G, Dép);
    Pour N dans ADJ
      Si Non est_marqué(N) Alors
        trajet=chemin(N, Arr, G) ;                              // de la forme  <N,..., Arr> si non_vide
        Si (trajet ≠ VIDE) alors retourne <Dép>.trajet;         //cons(Dép, tr) : on place Dép puis trajet dans une liste
        Fin si;
    Fin Pour;
    Retourne VIDE;
Fin chemin;
```

X.3.1- Amélioration du calcul du chemin (en Profondeur)

⇒ Autre solution (trajet en paramètre) avec une meilleure gestion du trajet.

⇒ On n'a plus besoin de marquer : le trajet sert aussi de mémoire de parcours.

```
Fonction chemin(Dép, Arr : nœud; G : Graphe; T : Trajet) : booléen =     // T est modifié par cet algorithme
Début                                                                     // (passage par référence)
        Si est_vide(G) alors retourne faux; Fin si;
        Si (Dép=Arr) retourne vrai;Fin si;
        ADJ=adjacents(G, Dép);
        Pour N dans ADJ
            Si N ∉T alors       T1 = T.<N>;                 # Ajouter N à l afin de T
                                Si       chemin(N, Arr, G, T1)
                                alors    T = T1; retourne vrai;
                                Fin si;
            Fin si;
        Fin Pour;
        retourne faux;
Fin chemin;
```

Appel : T=[D];

Si chemin(D,A,G, T)=vrai --> T contient au retour le trajet

X.4- Exercice : parcours A*

Voir aussi le sujet du TD du même sujet.

On dispose d'un graphe orienté pondéré (généré aléatoirement).

Les noeuds sont présentés avec leur coordonnées (x,y) sur un plan.

Un noeud de départ et un noued d'arrivée sont initialement précisés.

On souhaite trouver un chemin par un parcours en largeur de la manière suivante :

- Sur un noeud courant, on établi une liste de tous les successeurs
- On ordonne ces successeurs selon la fonction suivante (voir ci-après) :

f(n) = g(n) + h(n) :

g(n) : la distance (cout) du chemin déjà parcouru (appelée "*distance de Dijkstra*")

h(n) = une estimation du chemin restant jusqu'à l'arrivée

- On choisira en premier le meilleur noeud selon ce critère.

A propos du critère :

h(n) peut être une distance {linéaire, Manhattan, Euclidienne, à vol d'oiseau, ...} :

Manhattan: la distance entre 2 points sur un plan en se déplaçant seulement horizontalement ou verticalement (comme la distance entre deux croisements à Manhattan !).

A vol d'oiseau : sur le même plan, c'est la distance de "Pythagore".

Linéaire : entre deux points (x,y) et (x',y'), c'est la distance |x'- x|+|y'- y|

Notons que si h(n)=0, alors on aura un algorithme de **Dijkstra**.

On se donne une classe **Noeud** avec les attributs

g : la valeur de g(n) : un entier
h : la valeur de h(n) : un entier # f = g + h
x, y : coordonnées du noeud dans le plan

Pour le trajet, on utilise ici le principe du "coming_from". Mais on peut aussi p. ex, utiliser une liste des noeuds parents d'un noeud à stocker dans la classe Noeud.

XI- Table des matières