Stratégies et Techniques de Résolution de Problèmes

Chapitre II-1 : Graphes et Algorithmes de Parcours

S7 - ECL – 2A – MI

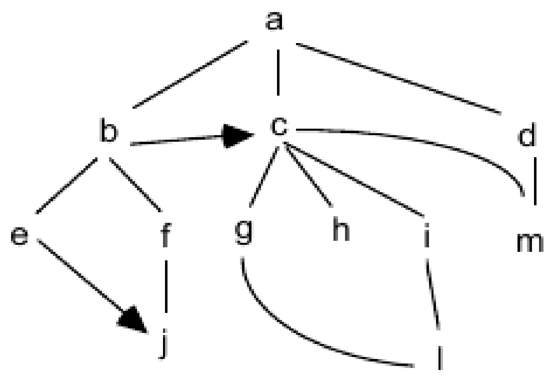
2021-2022

Alexandre Saidi

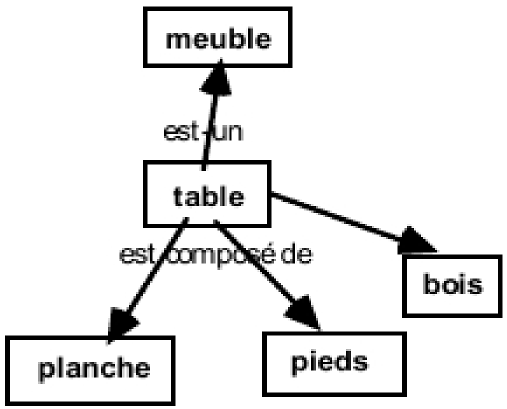
I- Introduction aux Graphes

- 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)



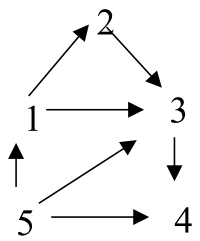
un graphe representant des liens d'héritage et de composition

II- Graphes : quelques notions

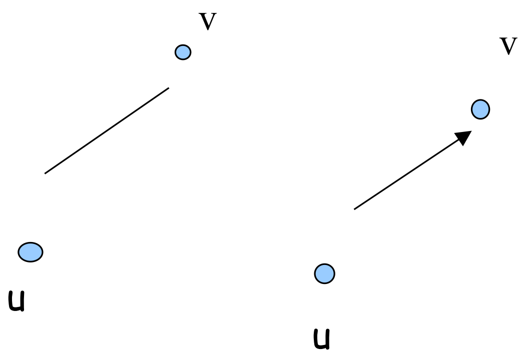
- Un graphe **G = (V, E)**

  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*)

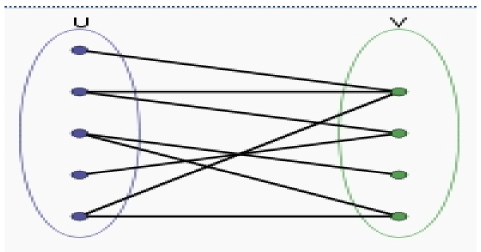
- **Graphe orienté** (**digraphe**) ⇒ *directed graph*
- **Poids** (weight) : valeur d'un arc/arête

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

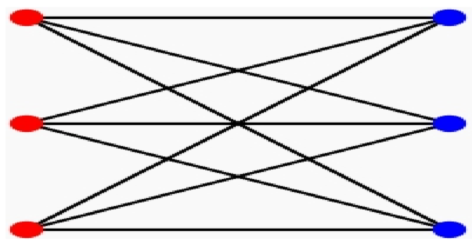
- **Chemin** (branche, path)
- Nœud *accessible* depuis un nœud X *si …*
- Longueur d'un chemin contenant N nœuds = nombre d'arcs = N - 1
- Un chemin non nul entre (v,w) dans un graphe orienté est un **circuit** (un **cycle**)

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

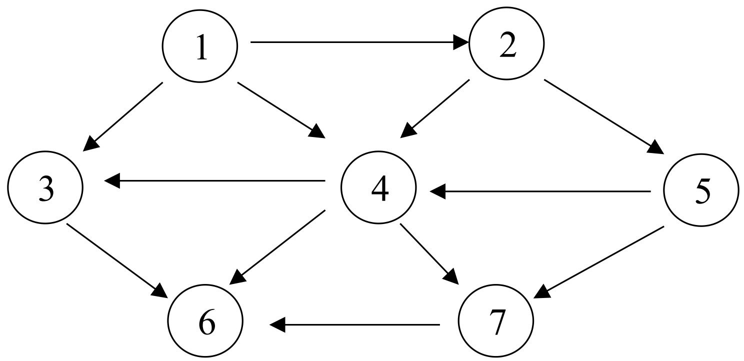
  --> La relation $\Re$ est alors appelé une **relation d'ordre total**.

  --> Par Exemple, '$\leq$' est un ordre total sur les entiers naturels.

- **Treillis** : est un arbre (G connexe, acyclique) avec une relation d'ordre <u>totale</u> 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- Structure de données pour représenter un graphe

On peut implanter les graphes de diverses manières :

- Par une matrice carrée
- 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.

III.1- Représentation statique par une matrice d'adjacence

- 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) $\equiv$ un graphe **peu dense** (*sparse*)

Exemple :

pour 3000 carrefours matrice de 9000000 éléments avec beaucoup de zéro / faux

La représentation précédente regroupe les deux cas :

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

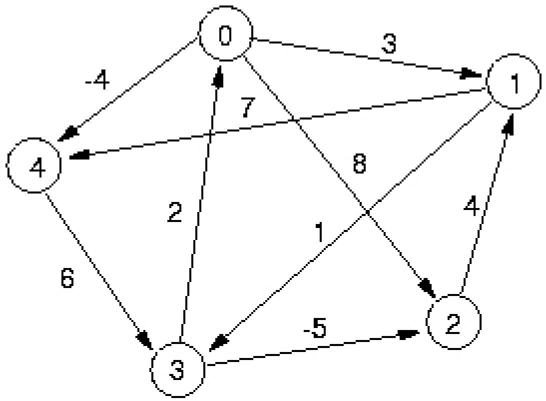
--> Une constante prédéfinie (par exemple -1) représente l'absence de lien.

--> 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 valué :



	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 ../..

III.2- Représentation dynamique par les listes d'adjacences

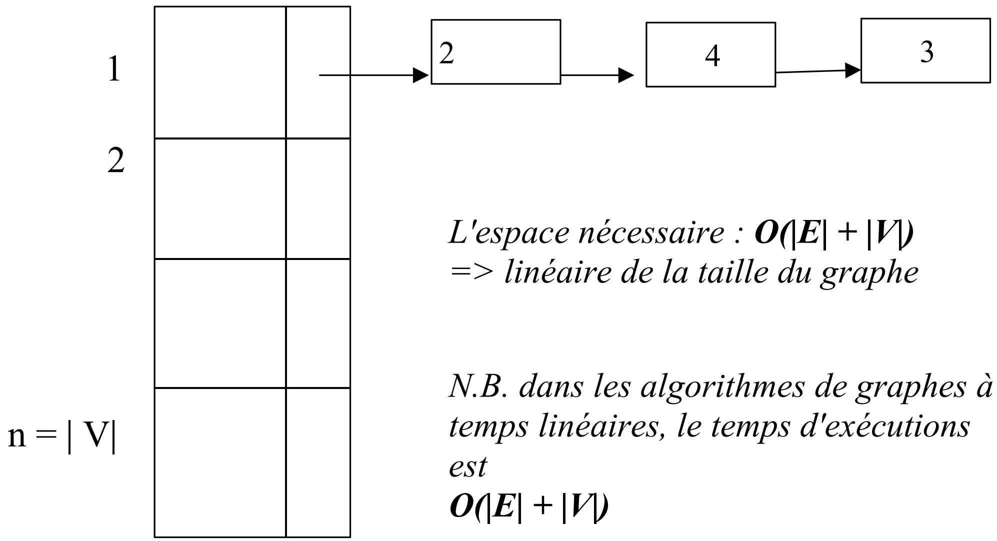
Une structure de données plus dynamique (adaptée pour un graphe peu dense) :

Pour un graphe <u>non orienté</u> :

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

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

Cette représentation est 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*|)**

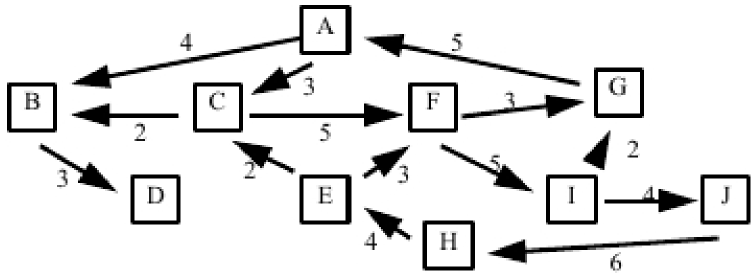
Par contre, elle n'est pas directement possible dans tout langage de progr.

Remarque : le premier tableau peut être également une liste.

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

III.3- Représentation alternative Hétérogène

Le graphe :



Représenté par deux tableaux / Listes :

- **V pour les nœuds (toutes infos sur noeuds)**
- E pour les arcs + valeur

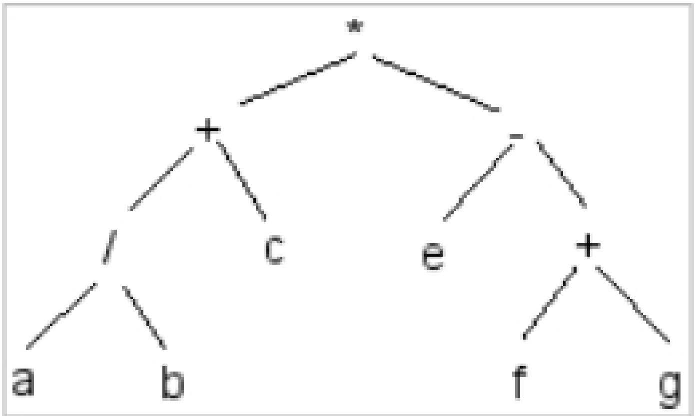
Dans le tableau E (à droite), la première ligne indique qu'il y a un arc du nœud A (indice 1 dans V) vers B (indice 2) avec un coût de 4.

	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

III.4- 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 TDA Graphe en annexes.

IV- 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**.

V- 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 : ref nœud) =
Début
   Si Non est_vide(G) alors
      marquer(nœud_courant(G));
      traiter(nœud_courant(G));          //traitement quelconque
      Pour X dans adjacents(nœud_courant(G))
         Si X n'est pas marqué alors
         profondeur(X);
         Fin si;
      Fin pour;
   Fin si;
Fin profondeur ;
```

N.B. :Ci-dessus, pour un graphe G non_vide, **nœud_courant(G)** représente le nœud actuellement référencé (comme la racine dans un arbre) contenant l'information du nœud, les adjacents, etc.

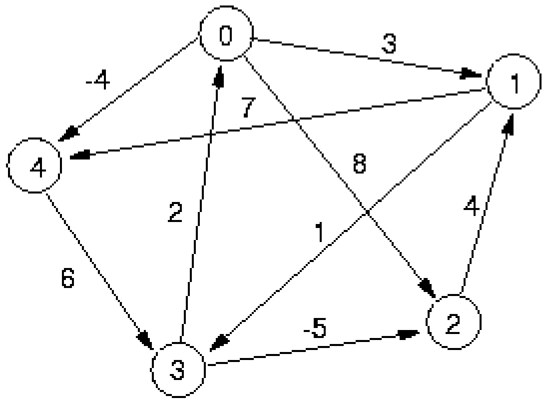
- Les mécanismes de **marquage** :
    - Marquage à l'intérieur du nœud
    - Marquage par une structure de données externe au graphe (un tableau, …)

Une autre version :

```
Procédure profondeur (G : ref nœud) =
Début
   Si Non est_vide(G) ET nœud_courant(G) n'est pas marqué alors
      marquer(nœud_courant(G));
      traiter(nœud_courant(G));    // traitement  quelconque
      Pour X dans adjacents(nœud_courant(G))
         profondeur(X);
      Fin pour;
   Fin si;
Fin profondeur ;
```

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

Exemple :

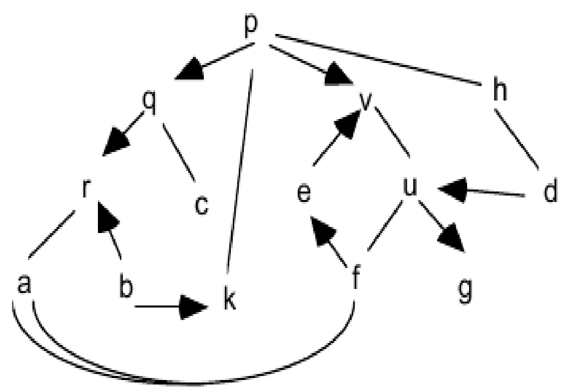


Pour ce graphe, un parcours en profondeur depuis le nœud 0 (en balayant les successeurs de gauche à droite) visiterait les nœuds dans cet ordre :

$$0 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$$

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

V.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)
   déclarer Pile=vide
   empiler(noeud_courant(G))
   Tant que NON vide(Pile)
      Noeud ← dépiler(Pile)
      Si NON  est_marqué(X) Alors  – dans certains cas, un noeud peut se trouver 2 fois dans la Pile (voir trace ci-dessus).
         marquer et traiter(noeud)
         Pour X dans adjacents(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) :

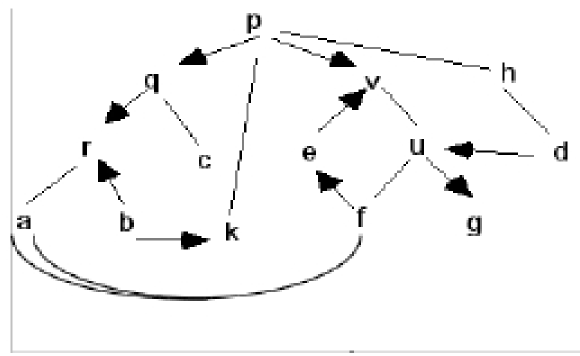
  [$\underline{p}$] → [h,v,k,$\underline{q}$] → [h,v,k,c,$\underline{r}$] → [h,v,k,c,$\underline{r}$]→ [h,v,k,c,$\underline{a}$]→ [h,v,k,c,$\underline{f}$]

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

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

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

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



VI- 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.

VI.1- Le Type (TDA) File

- A propos des TDAs : outil de spécification algébrique de type.
- Les opérateurs de la File d'attente *File(Élément)* utilisée :

File_vide :		→	File
Enfiler :	Élément x File	→	File
Défiler :	File	/→	File
Sommet :	File	/→	Élément
Est_file_vide :	File	→	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

Sommet(Enfiler(e, f))=Sommet(f)

Est_file_vide(File_vide) =vrai

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

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

VI.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 : ref nœud) =
Début
    Si Non est_vide(G) alors
        traiter(nœud_courant(G));          //une opération quelconque
        Pour X dans adjacents(nœud_courant(G));
                File=Enfiler(X, File);
        Fin pour;
        X=Sommet(File);
        File=Défiler(File);                // car défiler ne renvoie pas un élément (cf. TDA File)
        largeur_récursif (X);
    Fin si;
Fin largeur_récursif ;
```

N.B. : 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) =
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_courant(X));         //une opération quelconque
    Pour X dans adjacents(nœud_courant(G));
          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 = liste_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_courant(X));            //une opération quelconque
      Pour X dans adjacents(nœud_courant(G));
               Si X n'est pas dans deja_vu_marquage :
                  File=Enfiler(X, File);
                  CF[X]=nœud_courant(G)
      Fin pour;
      largeur_récursif (File);
   Fin si;
Fin largeur_récursif ;
```

VI.2.1- Implantation Python via un exemple

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

- ✓ 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={};                                        # un Dictionnaire Python
    dico_CF[case_depart]=case_depart

      # fonction utilitaire
      def chemin_fonction_utilitaire() :
           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_fonction_utilitaire()

return chemin_fonction_utilitaire()
```

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.

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

Ajouter le mécanisme de marquage.

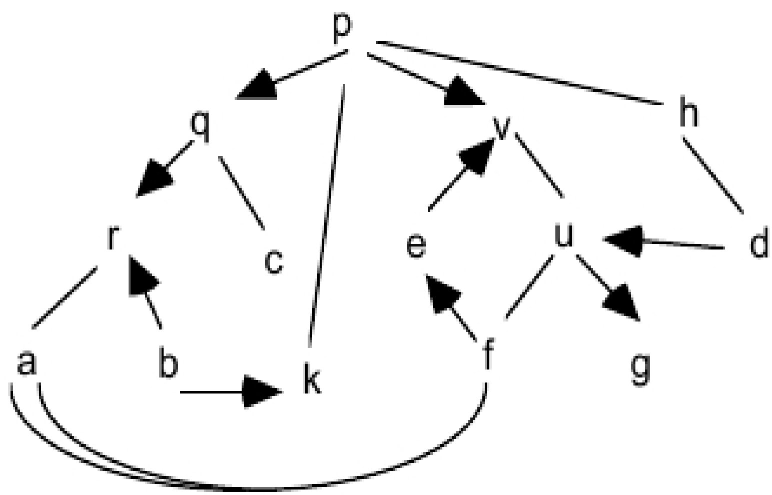
```
File : file d'attente=File_vide;
Procédure largeur_itératif ( G : graphe)
déclarer File=vide
Début
    Si Non est_vide(G) alors
        Enfiler(nœud_courant(G), File);
    Fin si;
    Tant que Non est_vide(File)
        N=Sommet(File);
        File =Défiler(File);
        traiter(N);      // ou visiter(N)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

N.B. : version sans marquage

VI.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.

VI.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 : graphe)
Début
    Si Non est_vide(G)
    Alors    Enfiler(nœud_courant(G), File);
    Fin si;
    Tant que Non est_vide(File)
        N=Sommet(File);
        File =Défiler(File);
        Si est_marqué(N) alors passer à l'itération suivante ;
        Sinon marquer(N);
        Fin si;
        traiter(N);     // ou visiter(N)
        Pour X dans adjacents(N);
            File=Enfiler(X, File);
        Fin pour;
    Fin Tant que;
Fin  largeur_itératif_marquage ;
```

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

Une seconde version avec marquage :

--> on marque les nœuds non marqués placés dans la file.

La file ne contiendra pas de doublon.

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

VII- Applications des parcours de graphes

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

marquage des nœuds ou encore mémorisation du trajet.

VII.1- Exemple simple : comptage du nombre de nœuds

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

VII.2- Fonction recherche d'un Élément

La fonction de recherche de l'élément **X** dans un graphe connexe

→ renvoie une référence (pointeur) sur le nœud trouvé sinon une réf. vide:

```
Fonction recherche(X:  élément; G: ref_nœud): ref_nœud =          //En profondeur (ref_nœud = itérateur)
   Début
      Si est_vide(G)
      Alors      retourne  ref_nœud_vide;
      Fin si;
      Si est_marqué(nœud_courant(G))
      Alors      retourne ref_nœud;                  // déjà visité
      Sinon
         Si (nœud_courant(G)=X) alors retourne G;
         Sinon
            marquer(nœud_courant(G));
            ADJ=adjacents(nœud_courant(G),G);
            Pour N dans ADJ
                  ref_nœud It=recherche(X,N) ;
                  Si (It ! =  ref_nœud_vide) alors retourne It;  Fin si;
            Fin Pour;
            Retourne  ref_nœud_vide;
         Fin si;
      Fin si;
Fin recherche;
```

VII.3- Exercice : parcours A*

On dispose d'un graphe orienté valué (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 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 (dite 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 vole 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

L'algorithme de principe suivant ne dit rien sur le trajet (à prévoir, p. ex, une liste des noeuds parents d'un noeud à stocker dans la classe Noeud).

VII.4- Algorithme de principe

L'algorithme suivant est plus complet pour la stratégie de parcours A_star.

--> En particulier, si la valeur h(n) = 0, on sera en présence d'un algorithme de Dijkstra.

Fonction parcours_A_star :

```
Entrée : G : Graphe, Départ, Destination : Noeud
Sortie : Trajet : Chemin (= vide si pas de soluton)
Debut
    Pour tout noeud N : N.g=infini
    Pour tout noeud N : N.f=infini
    Pour tout noeud N : N.h=distance(N –> Destination)     # euclidienne par défaut
    Départ.g=0
    Départ.f=Départ.h          # distance euclidienne par défaut
    Coming_From={}      # pour le Trajet
    déjà_visités={}     # ensemble vide
    File_attente=[Départ]
```

```
Tant que File_attente 6= [] :
    trier File_attente selon la valeur de f des noeuds
    noeud = défiler(File_attente) # la tête de la file est retirée
    Si noeud = Destination # identité des points (x,y) et (x',y')
    Alors renvoyer construire_trajet(noeud, Coming_From)
    liste_des_voisins=voisins(noeud, G)
    Si liste_des_voisins = []
    Alors passer à l'itération suivante # "continue" en Python
    Pour tout V dans liste_des_voisins :
            estimation_valeur_g_de_V= noeud.g + distance(noeud –> V) # euclidienne
            Si estimation_valeur_g_de_V < V.g
            Alors Coming_From[V] = noeud
                V.g = estimation_valeur_g_de_V
                V.f=V.g+V.h
                enfiler(V)
                ajouter V à déjà_visités
  renvoyer [] # ECHEC
Fin parcours_A_star
```

VII.4.1- Les données

Nous aurons besoin des classes Noeud et Graphe.

```
class Noeud :
    def __init__(self, x_y, liste_coords_sofar=[], val_g=99999999, val_h=0, degre=0) :
        x,y=x_y
        self.x=x; self.y=y
        self.g=val_g; self.h=val_h; self.f= self.h + self.g
        self.degre=degre
    def MAJ_val_de_g(self, val) : ...
    def set_h(self, val) : self.h=val
    def set_g(self, val) : self.g=val
    def setDegre(self, degre): self.degre=degre
    def evaluer_la_fonction_f_d_un_noeud(self, other) :
        self.h=self.distance_euclidienne(other)
        return self.g + self.h
    def distance_euclidienne(self,Noeud) :
        return math.sqrt((self.x−Noeud.x)**2+(self.y−Noeud.y)**2)
    def __str__(self) : ...
```

```
def egal_en_coordonnees(self, Noeud) :
    return self.x==Noeud.x and self.y==Noeud.y
def __lt__(self, other) : # nécessaire pour le tri
    res=self.comparer(other)
    return res==1
def comparer(self, Noeud2) : # renvoie +1 , 0 ou −1
    if self.g + self.h < Noeud2.g + Noeud2.h : return +1
    if self.g + self.h == Noeud2.g + Noeud2.h : return 0
    return −1
```

Et la classe Graphe :

```
class Graphe :
    def __init__(self, is_DAG=False) : # pard défaut "non orienté".
        self.vertices_dico={}
        self.edges_dico={} # entre V1 et V2 (si is_DAG=False, les couples sont répétés, sinon, dans un seul sens)
        self.is_DAG =is_DAG
        self.taille=0
    def addNode(self, noeud : Noeud) : # Le noeud : (x,y) + autres infos
        if (noeud.x, noeud.y) not in self.vertices_dico :
            self.vertices_dico[(noeud.x, noeud.y)]=noeud #.getInfos_sans_coordonnees()
            self.taille+=1
            self.edges_dico[(noeud.x, noeud.y)]=[] # pas de edge pour l'instant pour ce noeud
    def afficheGraphe(self) :
        for k,v in self.vertices_dico.items() : print(k,v)
        for k,v in self.edges_dico.items() : print(k,'−>', v)
    def get_edges_d_un_noeud(self, coord_noeud) :
        return self.edges_dico[coord_noeud]
    def get_coords_des_voisins_d_un_noeud(self, noeud) :
        x,y=noeud.x, noeud.y
        return self.edges_dico[(x,y)]
```

```
def addEdge(self, from_coords_V1, to_coords_V2) :
    assert from_coords_V1 in self.vertices_dico and to_coords_V2 in self.vertices_dico
    if to_coords_V2 not in self.edges_dico[from_coords_V1] :
        self.edges_dico[from_coords_V1].append(to_coords_V2)
        self.vertices_dico[from_coords_V1].degre +=1 # Le degré du noeud augmente
    if self.is_DAG : # et le cas symétrique
        return
    # On met l'arc inverse
    if from_coords_V1 not in self.edges_dico[to_coords_V2] :
        self.edges_dico[to_coords_V2].append(from_coords_V1)
        self.vertices_dico[to_coords_V2].degre+=1 # Le degré du noeud augmente
def construire_trajet(self,CF, dernier_noeud_visité):
    chemin=[(noeud_arrivee.x, noeud_arrivee.y)]
    x_y=(noeud_arrivee.x, noeud_arrivee.y)
    while x_y != (noeud_depart.x, noeud_depart.y) :
        chemin.append(CF[x_y])
        x_y=CF[x_y]
    chemin.reverse()
    return chemin
```

VII.5- Une 2e Algorithme de principe

L'algorithme suivant est plus sommaire et traite un cas (trop) simplifié !

```
def parcours_A_star(G : Graphe, Dép : Noeud, Dest : Noeud) -> trajet :
    Visités=[]
    File_attente=[Dép]
    while File_attente != [] :
        Noeud = défiler(File_attente)
        if egal(Noeud, Dest) : # identité des points (x,y) et (x',y')
            return construire_trajet(Noeud)  # Non fourni !
        else :
            Les_voisins=voisins_ordonnes-selon_fonction_f(Noeud, G)
            if Les_voisins==[]  :  break
            for V in Les_voisins :
                if V not in visistés :
                    V.cout=Noeud.cout +1
                    V.f = v.cout + distance(V, Dest)
                    File_attente.append(V)
            Visités.append(V)
    print("Problème")

    return []
```

Dans la fonction **voisins_ordonnes-selon_fonction_f**, on aura besoin d'un comparateur dont le code peut être :

```
def comparer(Noeud1, Noeud2) : # renvoie +1 , 0 ou -1
    if Noeud1.g + Noeud1.h < Noeud2.g + Noeud2.h :
        return +1
    if Noeud1.g + Noeud1.h == Noeud2.g + Noeud2.h :
        return 0
    return -1
```

VIII- 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_courant(G)=Départ* (sinon, on s'y place d'abord).

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

VIII.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: ES trajet) : booléen =      // ES vaut dire : entrée-sortie
Début                                                                           // (passage par référence)
        Si est_vide(G) alors retourne faux; Fin si;
        Si (Dép=Arr) retourne vrai;Fin si;
        ADJ=adjacents(Dép, G);
        Pour N dans ADJ
              Si N ∉ T alors    T1 = T.<N>;
                                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 le trajet