Chapitre 6 : Graphes, définition, représentations, parcours

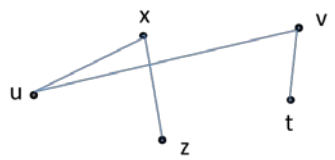
Les graphes sont des outils de modélisation très fréquemment étudiés et utilisés en mathématiques et en informatique. On peut par exemple les utiliser pour représenter les réseaux de communication, physiques ou virtuels : routes, autoroutes, réseau Internet, etc.

1. Graphe : quelques définitions

Graphe non orienté

Un graphe non orienté est un couple <S, A> où S est un ensemble fini de sommets et A un ensemble fini de paires de sommets, les arêtes : a = {u, v}.
Dans un graphe non orienté, une arête a={u,v} est dite connecter les sommets u et v et peut être notée : u–v ou u–^{a}v ou v–u ou v–^{a}u.

Exemple de graphe non orienté

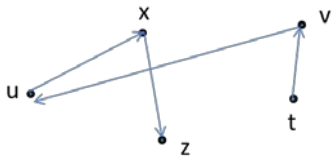


S = {u,x,z,v,t}
A = [{u,x} ; {u,v} ; {x,z} ; {v,t}]

Graphe orienté

Un graphe orienté est un couple <S, A> où S est un ensemble fini de sommets et A est un ensemble fini de couples de sommets, a = (v, u), appelés arcs.
Dans un graphe orienté, si l'arc a est égal à (v,u), on dit que a **connecte** les sommets v et u, avec v comme origine (ou source, ou extrémité initiale) et u comme but (ou extrémité terminale).
On peut également noter a=v→u ou encore v→^{a}u.
Toujours dans un graphe orienté, si l'arc a=(v,u) existe, on dit que u est un **successeur immédiat** de v, et que v est un **prédécesseur immédiat** de u.

Exemple de graphe orienté

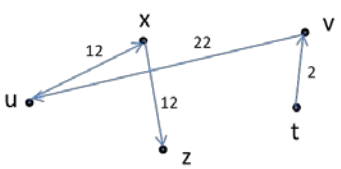


S = {u,x,z,v,t}
A = {(u,x) ; (v,u) ; (x,z) ; (t,v)}

Graphe valué (orienté ou non)

Un graphe valué orienté (respectivement **non orienté**) est un triplet <S, A, C> où <S, A> est un graphe orienté (resp. non orienté) et C est une fonction de l'ensemble A des arcs (resp. des arêtes) dans l'ensemble des réels. C est appelée fonction de **coût** (ou encore **valuation** ou **poids**).
Par défaut, le coût d'une arête (ou d'un arc) est égal à 1 si l'arête existe, à zéro sinon.

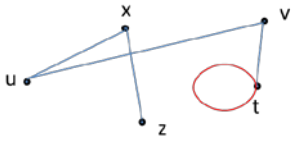
Exemple pour le graphe ci-dessus



Boucle

Une **boucle** est une arête ou un arc connectant un sommet à lui-même.

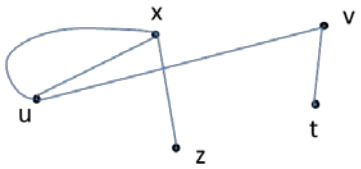
Exemple de boucle



Multigraphe

Un **multi graphe orienté** (respectivement non orienté) est un couple <S, A> où S est un ensemble fini de sommets et A est un ensemble fini de triplets (a, u, x) où u et x sont des sommets et a est une lettre d'un alphabet donné, respectivement (a, {u, x}) dans le cas non orienté.
Cela signifie que plusieurs arcs ou arêtes distincts peuvent joindre deux sommets : il peut ainsi y avoir deux triplets, (a, u, x) et (b, u, x) avec a ≠ b : on parle alors d'un **multi-arc** ou d'une **multi-arête**.

Exemple de multi graphe non orienté

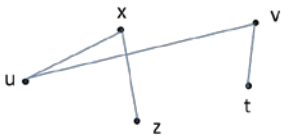


Deux arêtes différentes relient u et x.

Graphe simple ARECTIFIER ARETES/GRAPHES

Un **graphe simple** est un graphe sans boucle et sans multi-arc ou multi-arête. **Nous ne considérerons dans ce cours que des graphes simples.**

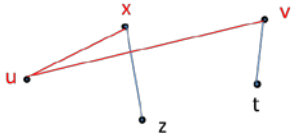
Exemple de graphe simple



Sous-graphe

Si G = <S, A> est un graphe et S' est une partie de S (l'ensemble des sommets), **le sous-graphe engendré par S'** est le graphe G' = < S', A'> où A' est l'ensemble des arêtes {u, v} de A telles que u et v soient dans S'.

Exemple de sous-graphe : un sous-graphe possible pour le graphe ci-dessus (en rouge ici). Il est constitué des sommets {u,x,v} et des arêtes [{u,x} ; {u,v}]



Arcs et sommets adjacents

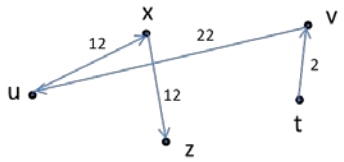
Deux **arêtes** (respectivement **arcs**) sont dits **adjacentes** si elles ont une extrémité commune. Dans le graphe ci-dessus les arêtes {u,x} et {u,v} sont adjacentes.
Deux **sommets** d'un graphe non orienté sont dits **adjacents** s'il existe une arête les reliant. Dans le graphe ci-dessus les sommets x et z sont adjacents.

Exemple avec le graphe ci-dessus

- Les arêtes {u,x} et {u,v} sont adjacentes en u.
- Les sommets u et v sont adjacents.

Arc incident (graphe orienté)

Si le graphe est orienté (cf. figure ci-dessous), l'arc a = v → u est dit **incident** à v **vers l'extérieur** et à u **vers l'intérieur**.



Degré d'un sommet dans un graphe orienté

Le nombre d'arcs incidents à un sommet « v » vers l'intérieur est **son demi-degré intérieur**, noté d°-(v) alors que le nombre d'arcs incidents vers l'extérieur pour un sommet « u » est dit être son **demi-degré extérieur**, noté d°+(u).

Dans le graphe orienté ci-dessus, le demi-degré intérieur de z est 1 et son demi-degré extérieur est 0 ; le demi-degré intérieur de u est 1 et son demi-degré extérieur est 1 également.

Chemin ou chaîne entre deux sommets et sa longueur

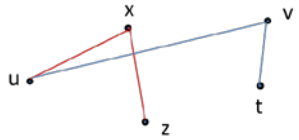
Dans un graphe non orienté (resp. orienté), on appelle **chaîne** (resp. **chemin**) **de longueur l** une suite de l+1 sommets u_0, u_1, …, u_l tels que, pour tout i, u_i - u_{i+1} est une **arête** (resp. un arc).

Par convention, on dit qu'il y a une chaîne (resp. un chemin) de longueur 0 de tout sommet vers lui-même.

Chemin ou chaîne élémentaire

Une **chaîne** (resp. un chemin) est dite **élémentaire** si elle ne contient pas plusieurs fois le même sommet.

Exemple

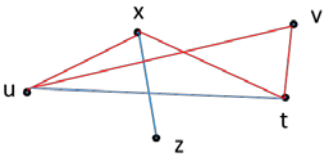


Dans ce graphe, la chaîne u-x-z est une chaîne élémentaire de longueur 2 (elle contient 3 sommets).

Circuit ou cycle

Une **chaîne** (resp. un chemin) dont les **arêtes** (respectivement les arcs) sont deux à deux distinctes et dont les extrémités sont égales est dit être un **cycle** (respectivement un circuit).

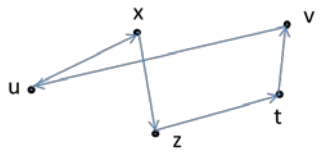
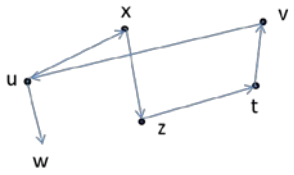
Exemple : la chaîne u-x-t-v-u est un cycle



Graphe fortement connexe et composante fortement connexe

Un graphe **orienté** est dit être **fortement connexe** si, pour toute paire de sommets u et v, *il existe un chemin allant de u à v et un chemin allant de v à u*.

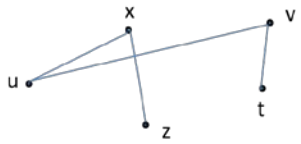
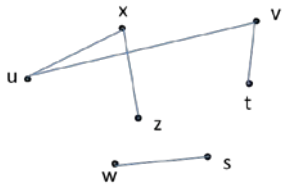
Une **composante fortement connexe** d'un graphe orienté est un sous-graphe fortement connexe maximal (pour l'inclusion).

	
Graphe orienté fortement connexe	Identifier la composante fortement connexe

Graphe connexe et composante connexe

Un graphe **non orienté** est dit être **connexe** si *pour toute paire de sommets u et v il existe une chaîne d'extrémités u et v*.

Une **composante connexe** d'un graphe non orienté est un sous-graphe connexe maximal (pour l'inclusion).

	
Graphe non orienté connexe	Identifier la composante connexe

Dans la suite, nous allons parler de la façon dont on peut représenter les graphes en algorithmique (pour pouvoir ensuite leur appliquer des algorithmes), puis choisir la représentation qui nous convient et découvrir quelques algorithmes sur les graphes.

2. Représentation des graphes par des matrices d'adjacence

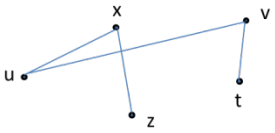
La représentation par des matrices d'adjacence convient bien pour des graphes très pleins, c'est-à-dire des graphes contenant beaucoup d'arcs ou d'arêtes.

On peut dire qu'un graphe est très plein lorsque P, le nombre d'arcs (ou d'arêtes) est de l'ordre de N^2, où N est le nombre de sommets (en clair, chaque sommet du graphe est relié à presque tous les autres sommets).

Lorsque le graphe est orienté, la position de la valeur de l'arc indique le sens de ce dernier (v,u) ou (u,v).

Lorsque le graphe est valué, la valeur placée dans la matrice correspond au poids (coût) de l'arc ou de l'arête.

Exemple de représentation par matrice d'adjacence :



	t	u	v	x	z
t	0	0	**1**	0	0
u	0	0	**1**	**1**	0
v	**1**	**1**	0	0	0
x	0	**1**	0	0	**1**
z	0	0	0	**1**	0

On suppose que le graphe admet au plus N sommets, N étant une constante donnée et on nomme « Indices » ou « Sommets » l'intervalle 1..N considéré.
On peut aussi choisir de numéroter les sommets avec n'importe quel type énuméré, par exemple des lettres, et Indice sera alors l'intervalle A..Z.

Les sommets de S, numérotés dans Indices ou Sommets, sont les indices d'un tableau à double entrée de booléens (graphe non valué), G.T, tel que :

G.T[u, v] = Vrai ⇔ (u, v) est un arc.

Une matrice d'adjacence est un tableau de ce type.

Voici un exemple de définition d'une matrice d'adjacence en langage algorithmique :

```
Type    Constante   taille : entier
        Sommets     1..Max                  {intervalle des entiers entre 1 et taille}
        Graphe =    enregistrement
                    {constante} taille : entier entre 1 et Max
                    T : tableau[Sommets, Sommets] de booléens
```

Si le graphe est pondéré, on utilise de la même façon une matrice G.T, telle que G.T[u,v] vaut le poids de l'arc ou bien 0 s'il n'y a pas d'arc.
Avec une telle représentation, les diverses fonctions nécessaires à la manipulation des graphes peuvent être définies plus ou moins facilement.

Nous utiliserons cependant dans la suite du cours une représentation par listes d'adjacence, qui permet une meilleure complexité dans les cas qui nous intéressent (graphes moins pleins que ci-dessus).

3. Représentation des graphes par des listes d'adjacence

Ici, on a une représentation adaptée aux graphes dont le nombre d'arcs ou d'arêtes est relativement faible : c'est le cas si P, le nombre d'arcs (resp. arêtes) est significativement plus petit que N^2 (avec N = G.taille = le nombre de sommets).

Le graphe <S, A, C> est toujours donné par une collection de sommets (sous forme de liste ou de tableau, par exemple), des arcs (resp. arêtes) et une fonction de coût.

On peut par exemple considérer les sommets comme les indices d'un tableau, G.T.

A chaque sommet, on associera alors la liste de ses successeurs immédiats (on parle de liste d'adjacence).
On aura donc pour les sommets u et v :

(u,v) est un arc (resp. arête) de A⇔ v apparaît dans un champ de la liste pointée par G.T[u].

Si le graphe représenté peut avoir des tailles évoluant beaucoup (reminder : la taille d'un graphe est le nombre de sommets), on préfèrera utiliser des listes de sommets plutôt que des tableaux (cf. plus loin).

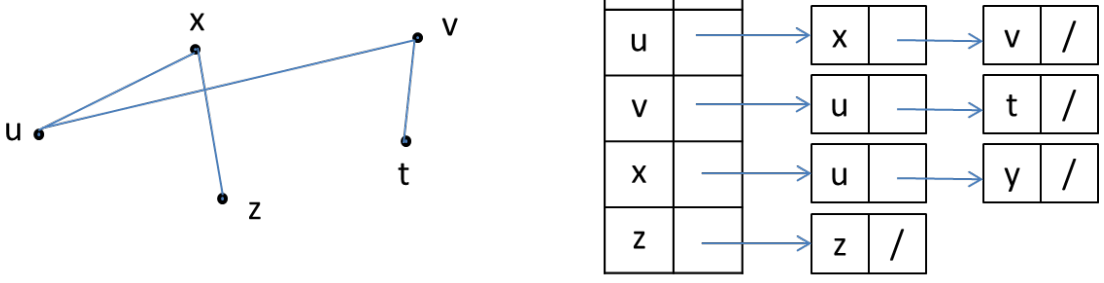
Exemple de déclaration de listes d'adjacences en langage algorithmique, avec un tableau de sommets :

```
Types       Constante : taille                                  /*entier*/
            Sommets : 1..taille
            Graphe = enregistrement
                        {constante si tableau} taille : entier
                        T : tableau[Sommets] de Liste_adja
                     fin enregistrement

            Liste_adja = enregistrement
                        n°: Sommets
                        poids : entier
                        suiv: Liste_adja
                      fin enregistrement
```

Exemple « dans la vraie vie » : les sommets pourraient représenter des processeurs répartis sur des sites (éventuellement distants) et leurs successeurs ou prédécesseurs immédiats seraient les voisins auxquels ils sont reliés par un canal (physique ou virtuel, orienté ou non, lent ou rapide). Localement, un processeur connaît ses portes d'entrée et de sortie, numérotées par lui, mais n'a pas forcément connaissance du réseau global et de sa numérotation.

Exemple de représentation par liste d'adjacence :



A partir d'une telle représentation des graphes, on peut proposer directement un certain nombre d'algorithmes.

- Autour de la notion de successeur immédiat d'un sommet

Pour les sommets u et v de S, v est un successeur (immédiat) de u ⇔ (u, v) est un arc du graphe, donc est dans A.
On a donc : (u, v) est un arc de A ⇔ v apparaît dans un champ de la liste d'adjacence pointée par G.T[u].
Cela se traduit par les algorithmes ci-après…
e.g. : dans le graphe ci-dessus, les successeurs immédiats de u sont x et v.

- Vérifier que v est un successeur de u

```
fonction est_successeur(u, v: Sommets; G: Graphe) : booléen
                                                    {rend vrai si v est un successeur de u}
        rendre est_dans(v, G.T[u])
fin_fonction
```

avec la fonction est_dans :

```
fonction est_dans(v: Sommets; L: liste_adja) : booléen
                                                    {rend vrai si v est dans L}
                                        {où L est la liste d'adjacence d'un sommet donné}
        si L = vide alors rendre faux
        sinon   si L.n° = v alors rendre vrai
                sinon rendre est_dans(v, L.suiv)
fin_fonction
```

- Une autre version possible…

```
fonction être_successeur(u, v: Sommets; G: Graphe) : booléen
                                                    {rend vrai si v est un successeur de u}
        P ← G.T[u]                          {P est une variable locale de type Liste_adja}
        tant que P ≠ Nil et P.n° ≠ v faire
                P ← P.suivant
        rendre P ≠ Nil
fin_fonction
```

Complexité : dans les deux cas, il faut accéder à la case u du tableau, ce qui se fait en temps constant. Ensuite, on parcourt la liste d'adjacence issue de u (linéaire en la taille de la liste) ; la complexité de ce parcours est bornée par le nombre de sommets (u est relié au plus à chacun des n-1 autres sommets). Le bornage par le nombre de sommet vient du fait qu'on a un graphe simple.
La complexité au pire est donc linéaire en le nombre de sommets du graphe.

- Autour de la notion d'arc

Plus précisément, vérifier l'existence d'un arc entre u et v… ce qui revient à vérifier que v est bien un successeur de u.
e.g. : dans le graphe ci-dessus, (u,v) est un arc, mais (z,t) n'en est pas un.

```
fonction être_arc(u, v: Sommets; G : Graphe) : booléen
                                                    {rend vrai si (u, v) est un arc de G}
        rendre être_successeur(u, v, G)
fin_fonction
```

complexité : la même que pour être_successeur.

- Autour de la notion de prédécesseur immédiat

Pour les sommets u et v de S, u est un prédécesseur immédiat de v si et seulement si (u,v) est un arc de A… autrement dit si v est un successeur immédiat de u.

```
fonction être_prédécesseur(u, v: Sommets; G : Graphe) : booléen
                                                    {rend vrai si u est un prédécesseur de v}
        rendre être_successeur(u, v, G)
fin_fonction
```

complexité : la même que pour être_successeur.

- Notion de degré sortant d'un sommet

Le degré sortant d'un sommet est le nombre de ses successeurs, c'est-à-dire le nombre d'éléments dans la liste d'adjacence… on en conclut facilement la forme de l'algorithme…
e.g. : dans le graphe ci-dessus, le degré sortant de v est 2.

```
fonction degré_sortant(u: Sommets; G: Graphe) : entier
                                                {rend le nombre de successeurs de u}
        rendre longueur(G.T[u])
fin_fonction

fonction longueur(L : liste_adja) : entier
                                                {rend la longueur de la liste}
        si L = vide alors
                rendre 0
        sinon
                rendre 1+longueur(L.suiv)
fin_fonction
```

Une variante possible avec une seule fonction

```
fonction degré_sortant(u: Sommets; G: Graphe) : entier
                                                {rend le nombre de successeurs de u}
        P ← G.T[u]                      {P est une variable locale de type Liste_adja}
        degre ← 0                               {degre est une variable locale entière}
        Tant que P ≠ Nil faire
                P ← P.suiv
                degre ← degre+1
        fin tant que
        rendre degre
fin_fonction
```

Complexité : il faut accéder à la case u du tableau (complexité constante), puis parcourir la liste d'adjacence, d'où à nouveau une complexité au pire bornée par le nombre de sommets.

- **Ensemble des arcs ou des arêtes**

La définition mathématique privilégie pour un graphe orienté les notions de sommets et d'arcs, et respectivement celle d'arêtes dans le cas non orienté.
Dans une représentation par liste d'adjacence, les sommets sont immédiatement accessibles :
Sommets : 1..taille, ce sont les entiers de 1 à taille, indices du tableau.
Pour les arcs c'est moins direct : (u, v) est un arc si v apparaît dans un champ de la liste pointée par G.T[u].

Un arc est donc désigné par une paire d'entiers (j,k) et peut être décrit comme un enregistrement :

```
Type    arc =   enregistrement
                source, but : entier
                fin enregistrement
```

Afin d'établir la liste des arcs, il faut donc pouvoir extraire les arcs un à un en parcourant les listes d'adjacence et les ranger dans une liste d'arcs…

```
Fonction Ensemble-arcs (G : Graphe) : Liste d'arcs
        New (L)
        Arcs (G, L)
        rendre L
{fin fonction}
```

```
Procédure Arcs (G : Graphe ; var L : Liste d'arcs)
        Pour k allant de 1 à G.taille faire
                P ← G.T[k].suiv                          {P variable locale, Liste_adja}
                Tant que P <> Nil faire
                        L ← cons( (k, P.n°), L)
                        P ← P.suiv
                {fin tant que}
        {fin pour}
{fin procédure}
```

Complexité : il faut accéder à toutes les cases du tableau, ce qui est d'une complexité en le nombre de sommets du graphe. Ensuite, il faut parcourir chaque liste d'adjacence, ce qui est également borné par le nombre de sommets. On a donc une complexité totale bornée par n x n, soit n^2.

Remarque : si l'on doit considérer également la fonction de poids, il suffira d'ajouter le poids dans la structure arc, ce qui n'augmente pas la complexité globale.

- **Ajouter un sommet**

Ajouter un sommet est évidemment plus simple si la représentation du graphe place les sommets dans une liste et non dans un tableau. Nous allons donc définir cette nouvelle représentation.

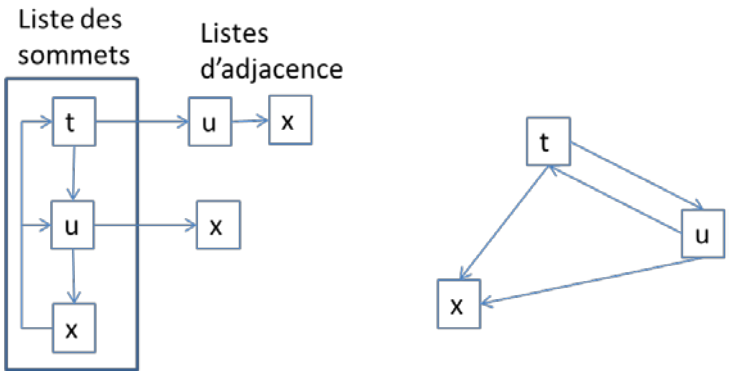
```
Types       liste_Sommets =
                    enregistrement
                        suiv, pred : liste_Sommets
                        n° : étiquette          /* par exemple des entiers*/
                        l : Liste_adja
                    fin enregistrement

            Graphe = enregistrement
                    taille : entier             /* non indispensable*/
                    g : Liste_Sommets           /* sommets du graphe*/
                fin enregistrement

            Liste_adja = enregistrement
                    n°: étiquette
                    poids : nombre
                    suiv : Liste_adja
                fin enregistrement
```

Ici, il n'est pas nécessaire d'imposer la taille comme champ du graphe, mais cela peut être pratique. Dans certains cas, on imposera que la liste Liste_Sommets G.g soit triée par étiquettes (champs n°).

Voici un schéma de la modélisation proposée :



Avec une telle représentation, ajouter un sommet u revient à ajouter un élément dans la Liste_Sommets G.g et à incrémenter la taille. On suppose que u n'est pas dans la liste.

```
Procédure Ajouter_sommet (u: étiquette; var G : Graphe)
        New(h)                                  {variable locale, h : Liste_Sommets}
        h.suiv ← G.g
        h.pred ← Nil
        h.n° ← u
        h.l ← Nil
        Si G.g <> Nil alors G.g.pred ← h     {la liste de départ n'était pas vide}
        G.taille ← G.taille+1
        G.g ← h                    /* mise à jour premier élément de la liste */
{fin procédure}
```

- Ajouter un arc

Pour ajouter un arc de poids p entre les sommets u et v, par exemple de u vers v, il faut ajouter un élément (v) dans la liste d'adjacence de u.
On suppose que l'arc de u vers v n'est pas déjà dans la liste, que u et v sont bien utilisés et que les listes d'adjacence ne sont pas triées… dans ce cas, il suffira d'accéder à la case où est le sommet u, et d'ajouter le couple (v,p) à la liste placée en cet endroit.

```
Procédure Ajouter_arc (u, v: étiquette; p : nombre ; var G : Graphe)
        H ← G.g                                 {variable locale, H : Liste_Sommets}
        Tant que H.n° <> u faire                {on parcourt les sommets}
                H ← H.suiv
        New(li)                                          {variable locale, li:Liste_adja}
        li.n° ←  v
        li.poids ← p
        li.suiv ← H.l              // ajout en tête de liste d'adjacence
        H.l ← li           // mise à jour de la liste d'adjacence
{fin procédure}
```

- Traitement des successeurs (immédiats) d'un sommet

Le traitement des successeurs immédiats d'un sommet u du graphe G se fait en parcourant sa liste d'adjacence (ici avec le tableau de sommets).

```
fonction traiter_successeurs (u: Sommets; G : Graphe) : booléen
                                        {traite tous les successeurs de u dans G}
        Si u<1 ou u>G.taille alors retourner FAUX
        P ← G.T[u]
        Tant que P ≠ Nil faire          {simple parcours de liste, cf. début de cours}
                v ← P.n°
                Traiter le sommet v
                P ← P.suiv
        Fin_tant_que
        Retourner VRAI
Fin Fonction
```

Pour traiter tous les successeurs d'un sommet, il faut examiner un nombre de cases égal à son degré sortant. Les fonctions d'accès aux différents successeurs dans la liste d'adjacence seront (premier_successeur et successeur_suivant) sont très similaires à la fonction ci-dessus.

- Traitement du graphe complet

Pour traiter tout le graphe, il suffit d'examiner un nombre de doublets égal au nombre P d'arcs du graphe, avec P inférieur ou égal à N^2.
En clair : il faut trouver un moyen judicieux de ne pas passer deux fois par un même sommet (si l'on passe par u *via* une liste d'adjacence, on doit le considérer comme définitivement traité).

Lors du parcours d'un graphe, il s'agit de passer une seule fois par tous les sommets en y effectuant un traitement donné et en utilisant seulement les arcs du graphe.
Cela peut être réalisé par deux types classiques de parcours : en profondeur ou en largeur.

4. Parcours en profondeur récursif

Le parcours en profondeur consiste à partir d'un sommet donné et à suivre un chemin le plus loin possible. On fait ensuite des retours en arrière pour parcourir tous les chemins ignorés précédemment.
Ce parcours est naturellement récursif ; c'est le retour de la fonction récursive à la procédure appelante qui gère le parcours arrière.

```
Procédure Parcourir (G : graphe)
                                                    {parcours en profondeur du graphe}
        {variables locales : Vu, tableau de booléens ; s sommets : entier entre 1 et G.taille}
                                                    {Vu dit quels sont les sommets déjà traités}
Pour s allant de 1 à G.taille faire
    Vu[s] ← faux
                                                    {on explore le graphe successivement à partir
                                                        de tous les sommets non encore traités}
Pour s allant de 1 à G.taille faire
    si non Vu[s] alors
        Prof_récursif(s, G, Vu)
fin_Procédure
```

La procédure Prof_récursif explore le graphe à partir d'un sommet donné (s'il n'a pas encore été traité). Elle a été écrite pour une représentation par listes d'adjacence.

```
Procédure Prof_récursif (s: Sommets ; G : Graphe ; var Vu : tableau[Sommets] de booléens)
                                        {exploration du graphe en profondeur à partir du sommet s}
    Vu[s] ← vrai
    Traiter (s)
    P ← G.T[s]                          {on va examiner tous les successeurs de s}
    Tant que P ≠ Nil faire
        v ← P.n°
                                                    {si ce successeur n'a pas encore été visité,
                                                        on explore le graphe à partir de lui}
        si non Vu[v] alors Prof_récursif(v, G, Vu)      {v n'a pas été visité}
        P ← P.suiv
    Fin_tant_que
fin_Procédure
```

Complexité : La complexité de Prof_récursif est linéaire en max(N,P), où N est le nombre de sommets et P le nombre d'arcs. En effet, on alterne entre le parcours du tableau de sommets (lorsqu'on est en bout de course) et les listes d'adjacence (donc les arêtes).

Le parcours obtenu ne dépend pas que de la structure du graphe mais aussi de l'ordre dans lequel les sommets ont été numérotés et de l'ordre dans lequel on les place dans les listes d'adjacence.

5. Parcours en profondeur itératif

On obtient une version itérative du parcours en profondeur récursif en utilisant une pile pour gérer les remontées le long des branches.

```
Procédure Parcourir_un_graphe (G : graphe)              {parcours en profondeur du graphe}
                {variable locale : on utilise un tableau Vu pour marquer les sommets à traiter}
        Pour s allant de 1 à G.taille faire
                Vu[s] ← faux
        Pour s allant de 1 à n faire
                si non Vu[s] alors Prof_itératif(s, G, Vu)
                    {si le sommet n'a pas encore été visité, on explore le graphe à partir de lui}
Fin_Procédure
```

Nb : la fonction englobante est identique à celle de la procédure récursive.

```
Procédure Prof_itératif(s : Sommets ; G : Graphe ; var Vu : tableau[Sommets] de booléens)
                                        {exploration du graphe en profondeur à partir du sommet s}
Vu[s] ← vrai ; Traiter (s)
Q ← pile_vide                                                   {On initialise une pile vide}
Empiler s sur Q
Poursuite ← faux

Si degre(s, G) ≠ 0 alors                                {ce sommet a des successeurs}
        poursuite ← vrai
        v ← premsucc(s, G)

Tant que la pile Q n'est pas vide faire
        Tant que poursuite faire
                Si (non(Vu[v])) alors
                        Vu[v] ← vrai
                        Traiter (v)
                        Empiler v sur Q         {on sauvegarde les sommets pour remonter}
                        s ← v
                        Si degré(s, G) ≠ 0 alors
                                v ← premsucc(s, G)
                        Sinon poursuite ← faux
                Sinon d ← degre(s, G)
                        Si v = succ(d, s, G) alors      {le d-ième successeur}
                                poursuite ← faux
                        Sinon poursuite ← vrai
                                v ← succsuivant(s, v, G)        {les autres successeurs}
        {fin de la poursuite}
                                        {on est allé au bout d'un des chemins, on remonte}
        v ← sommet de Q ; dépiler Q ;
        Si la pile Q n'est pas vide alors
                s ← sommet de Q ; dépiler Q;
                d ← degre(s, G)
                Si v ≠ succ(d, s, G) alors
                        v ← succsuivant(s, v, G)
                        Poursuite ← vrai
fin de la boucle de pile
Fin_Procédure
```

On se contente d'utiliser ici les fonctionnalités de base des piles et des graphes définies précédemment.

6. Parcours en largeur

A partir d'un sommet donné, on visite d'abord tous ses successeurs immédiats, puis tous les successeurs immédiats non encore visités du premier visité, puis du second, et ainsi de suite à partir de tous les successeurs.
C'est une opération simple si l'on utilise une file.

```
Procédure Parcourir_en_largeur_un_graphe (G : graphe) {parcours en largeur du graphe}
                                {On utilise un tableau «Vu » pour marquer les sommets traités}
        Pour s allant de 1 à G.taille faire
                Vu[s] ← faux
        Pour s allant de 1 à G.taille faire
                si non Vu[s] alors Largeur(s, G, Vu)
                    {si le sommet n'a pas encore été visité, on explore le graphe à partir de lui}
Fin_Procédure_ Parcourir_en_largeur_un_graphe


Procédure Largeur (s : Sommets ; G : Graphe ; var Vu : tableau[Sommets] de booléens)
                                    {exploration du graphe en largeur à partir du sommet s}
        Vu[s] ← vrai
        Traiter (s)
        New(F)                                  {On utilise une file F locale initialisée à vide}
        Ajouter s à F

        Tant que F n'est pas vide faire
                w ← tête(F)                     {variable locale w, de type sommets}
                Décapiter F
                Liste ← G.T[w]                  {variable locale Liste, de type liste_adja}
                Tant que Liste <> Vide faire
                        v ← Liste.n°
                        si non Vu[v] alors
                                Vu[v] ← vrai
                                traiter (v)
                                ajouter v à F
                        Liste ← Liste.suiv
                {fin Tant que}
        {fin Tant que}
Fin_Procédure_Largeur
```

Là encore, le parcours qu'on obtient ne dépend pas seulement de la structure du graphe, mais également de l'ordre dans lequel on aura numéroté les sommets dans le tableau de sommets et dont on les aura placés dans la liste d'adjacence.

7. Une application possible des graphes : le code de Gray

Le code de Gray (ou code Gray, ou code binaire réfléchi) est un type de codage binaire. Il permet de ne modifier qu'un seul bit à la fois lorsqu'on augmente un nombre d'une unité.

Le code Gray est donc une fonction qui associe à chaque nombre une représentation binaire dans laquelle le codage de deux nombres consécutifs ne diffère que d'une position (ce n'est pas le cas dans un codage binaire naturel).

Voici un exemple sur 4 bits :

Codage décimal	Codage binaire naturel	Codage gray/ binaire réfléchi
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

L'intérêt d'un tel code est d'éviter des états transitoires générateurs d'erreurs ; il a des applications pour l'encodage des angles, pour l'identification des mouvements de la souris et bien d'autres.

On peut associer des graphes aux codes de Gray et donc utiliser sur ces derniers les parcours que nous avons vus dans ce chapitre.
Pour tout entier positif, n, on considère le graphe G_n à 2^n sommets.
Les sommets de G_n sont les parties de l'ensemble {1, 2, …, n}.
Ainsi, si n = 2, les quatre sommets sont ø, {1}, {2}, {1, 2}, pour n=3 on a huit sommets, etc.
On ne passe d'un sommet à un autre que s'il s'agit de parties qui ne diffèrent que d'un élément.

Chaque sommet est toujours sur exactement n arêtes, ces n arêtes étant, localement pour le sommet considéré, numérotées de 1 à n, et telles que :

- si X est un sommet, donc X est un sous-ensemble de {1, 2, …, n},
- si k n'appartient pas à X, alors l'arrête de numéro k issue du sommet X relie X à X ∪ {k} ;
- de façon duale, si le nombre k appartient au sommet X, l'arrête de numéro k issue du sommet X relie X à X-{k}.
-

Les graphes G_n sont dits être des graphes de Gray.