

Graphes

Un graphe est un couple $G = (S, A)$, où $A \subset S \times S$.

S est l'ensemble (fini) des sommets et A l'ensemble des arêtes.

On note souvent n (noté souvent $|S|$) le nombre de sommets et m le nombre d'arêtes.

Le graphe est non-orienté ssi $(x, y) \in A$ ssi $(y, x) \in A$.

Lorsque $(x, y) \in A$, on note souvent $x \rightarrow y$. On dit que x est un successeur de y et que y est un prédécesseur de x . Lorsque le graphe est non orienté, on dit que x et y sont voisins.

L'arité (ou degré) $d(x)$ d'un sommet x dans un graphe non-orienté est le nombre d'arêtes d'extrémité x .

Dans un graphe orienté, on distingue le degré entrant et le degré sortant (appelé aussi arité).

Remarque : Dans un graphe non orienté $G = (S, A)$, on a $\sum_{x \in S} d(x) = 2m$, où $m = |A|$ est le nombre d'arêtes.

0. Exemple de modélisation par un graphe

Une chèvre, un chou et un loup se trouvent sur la rive d'un fleuve ; un passeur souhaite les transporter sur l'autre rive mais, sa barque étant trop petite, il ne peut transporter qu'un seul d'entre eux à la fois. Comment doit-il procéder afin de ne jamais laisser ensemble et sans surveillance le loup et la chèvre, ainsi que la chèvre et le chou ?

Cette situation peut être modélisée à l'aide d'un graphe. Désignons par P le passeur, par C la chèvre, par X le chou et par L le loup. Les sommets du graphe sont des couples précisant qui est sur la rive initiale, qui est sur l'autre rive. Ainsi, le couple (PCX, L) signifie que le passeur est sur la rive initiale avec la chèvre et le chou (qui sont donc sous surveillance), alors que le loup est sur l'autre rive. Une arête relie deux sommets lorsque le passeur peut passer d'une situation à l'autre. En transportant la chèvre, le passeur passe par exemple du sommet (PCX, L) au sommet (X, PCL) . Le graphe ainsi obtenu est biparti : les sommets pour lesquels le passeur est sur la rive initiale ne sont reliés qu'aux sommets pour lesquels le passeur est sur l'autre rive. Naturellement, on ne considère pas les sommets dont l'une des composantes est CX ou LC car ces situations sont interdites.

Le problème se ramène alors à trouver un chemin entre $(PCXL, -)$ et $(-, PCXL)$.

1. Matrices d'adjacence et chemins

a) Longueur d'un chemin et chemins réduits

Un chemin de longueur p dans un graphe $G = (S, A)$ est une suite

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_p \quad , \quad \text{où } \forall k \in \llbracket 1, p \rrbracket, (x_{k-1}, x_k) \in A$$

Un chemin est dit réduit (ou élémentaire) si les sommets sont deux à deux distincts.

S'il existe un chemin reliant x à y , il existe un chemin réduit de longueur $\leq n - 1$ reliant x à y .

En effet, on considère parmi les chemins possibles un chemin de longueur minimale.

S'il admettait une boucle (c'est-à-dire $x_i \rightarrow \dots \rightarrow x_i$), on obtiendrait en la supprimant un chemin de longueur strictement inférieure. Donc les sommets sont bien tous distincts.

Remarque : La réduction d'un chemin peut se faire en temps linéaire (en la longueur du chemin initial).

b) Matrices d'adjacence

La matrice M d'adjacence d'un graphe G sur l'ensemble $\llbracket n \rrbracket = \{0, 1, \dots, n-1\}$ est définie par

$$M = (m_{ij})_{0 \leq i < n, 0 \leq j < n}, \text{ où } m_{ij} = 1 \text{ si } (i, j) \text{ est une arête et } 0 \text{ sinon}$$

Remarque : La matrice est symétrique ssi le graphe n'est pas orienté.

Prop : Le nombre de chemins de longueur p joignant i à j est donné par le coefficient (i, j) de M^p

Preuve : En effet, les coefficients m_{ij}^p de M^p vérifient $m_{ik}^{p+1} = \sum m_{ij}^p m_{jk}$.

Or, tout chemin de longueur $p+1$ reliant i à k se décompose de façon unique comme le concaténé d'un chemins de longueur p reliant i à un sommet j et d'une dernière arête qui relie j à k .

On conclut par récurrence sur p .

c) Chemin élémentaire et caractérisations de la connexité

G est connexe ssi deux sommets quelconques sont reliés par un chemin, donc par un chemin (réduit) de longueur $\leq n-1$, donc ssi tous les coefficients de $I + M + M^2 + \dots + M^{n-1}$ sont strictement positifs.

Variante : G est connexe ssi $(I + M)^{n-1}$ est à coefficients strictement positifs.

d) Matrices booléennes (complément culturel)

Une variante consiste à considérer la matrice booléenne M (et à remplacer $+$ et \times par **or** et **and**).

On note souvent \vee et \wedge pour **or** et **and**. Ainsi, $(M \vee N)$ est la matrice des booléens ($\mathbb{M}[i, j]$ **or** $\mathbb{N}[i, j]$).

La matrice d'adjacence (booléenne) est donc : $m_{ij} = 1$ ($= \text{True}$) ssi i est relié à j , et 0 ($= \text{False}$) sinon.

Alors le coefficient (i, j) de M^k vaut 1 ssi il existe un chemin de longueur k joignant i à j .

La matrice $I \vee M \vee M^2 \vee \dots \vee M^{n-1} = (I \vee M)^{n-1}$ ne contient que des 1 ssi le graphe est connexe.

Remarque : Ainsi, connaître la fermeture transitive (c'est-à-dire les couples (x, y) tels qu'il existe un chemin de x à y) peut se faire en calculant $(I \vee M)^{n-1}$ ce qui nécessite $O(n^3 \log n)$ opérations. En fait, l'algorithme de Roy-Warshall utilisant le principe de programmation dynamique permet d'obtenir une complexité en $O(n^3)$.

2. Graphe défini par la liste des listes d'adjacence

Au lieu d'utiliser la matrice d'adjacence, on peut définir le graphe par la liste G de sorte que :

pour tout sommet $x \in S$, $G[x]$ est la liste des $y \in S$ tels que $(x, y) \in A$.

Remarque : Ce codage est particulièrement intéressant lorsque le nombre d'arêtes est petit devant n^2 .

Remarque : En pratique, les sommets sont numérotés de 0 à $n-1$.

3. Caractérisation des arbres (non ordonnés) parmi les graphes non orientés

a) Un cycle est un chemin de la forme $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_p = x_0$.

On dit qu'il est de longueur p si les sommets x_0, \dots, x_{p-1} sont distincts.

Un graphe est dit acyclique s'il ne contient aucun cycle.

b) Prop : Un graphe connexe (non orienté) de n sommets admet au moins $n - 1$ arêtes.

Preuve : pour $n \geq 2$, on supprime une arête (x, y) . Le graphe obtenu a alors au plus deux composantes connexes. On applique l'hypothèse de récurrence aux composantes connexes de x et y (si elles sont distinctes). On itère le procédé de suppression des arêtes sinon.

c) Un graphe est un arbre ssi il vérifie les assertions suivantes, qui sont équivalentes :

- (i) G est connexe et $m = n - 1$, où n est le nombre de sommets et m le nombre d'arêtes
- (ii) G est connexe et acyclique (c'est-à-dire sans cycle)
- (iii) G est acyclique et $m = n - 1$
- (iv) Il existe un unique chemin réduit (c'est-à-dire sans cycle) joignant deux sommets arbitraires
- (v) G est connexe minimal dans le sens où toute suppression d'une arête déconnecte le graphe.

d) Un arbre enraciné est un arbre dont on particularise un sommet x_0 , appelé racine.

Pour tout autre sommet x , il existe alors un unique chemin réduit allant de x_0 à x , appelé branche.

La distance d'un sommet à la racine s'appelle hauteur de x , et la hauteur de l'arbre est la hauteur maximale (c'est-à-dire la longueur maximale des branches).

En général, dans un arbre enraciné, on oriente les arêtes depuis la racine ; les successeurs d'un sommet sont appelés fils et les feuilles sont les sommets n'ayant aucun fils.

Tout sommet distinct de la racine a alors un unique parent. **On peut donc coder un arbre par un tableau parent.** On convient alors que la racine est son propre parent.

Pour déterminer l'unique chemin reliant x_0 à x , il suffit d'itérer **parent** à partir de x jusqu'à x_0 .

e) Parcours en largeur et en profondeur d'un arbre

Principe du parcours en largeur :

On visite les sommets par niveau : d'abord la racine, puis ses fils, puis les fils de ses fils, etc.

Principe du parcours en profondeur :

Le nom de parcours en profondeur vient du fait que, contrairement à l'algorithme de parcours en largeur, il explore en fait « à fond » les chemins un par un : pour chaque sommet, il marque le sommet actuel, et il visite les successeurs non marqués, et revient alors au sommet père.

f) Numérotations préfixe et postfixe des sommets d'un arbre

Dans une numérotation préfixe, tout sommet admet un numéro inférieur à ceux de ses fils.

Dans une numérotation postfixe, tout sommet admet un numéro supérieur à ceux de ses fils.

Exemple :

```

    graph TD
      0 --> 4
      0 --> 3
      4 --> 1
      4 --> 6
      3 --> 5
  
```

$$\left\{ \begin{array}{l} n = 7 \text{ et } G = [[4, 2, 3], [], [], [5], [1, 6], [], []] \\ \text{parent} = [0, 4, 0, 0, 0, 3, 4] \\ \text{ordre préfixe largeur} : 0, 4, 2, 3, 1, 6, 5 \\ \text{ordre préfixe profondeur} : 0, 4, 1, 6, 2, 3, 5 \\ \text{ordre postfixe profondeur} : 1, 6, 4, 2, 5, 3, 0 \end{array} \right.$$

4. Graphe inverse d'un graphe orienté

Etant un graphe orienté $G = (S, A)$, le graphe inversé est obtenu en inversant l'orientation des arêtes. Autrement dit, il s'agit du graphe $G^T = (S, B)$, où $\forall (x, y) \in S^2, (x, y) \in B \Leftrightarrow (y, x) \in A$.

La matrice d'adjacence de G^T est la transposée de la matrice d'adjacence de G .

Supposons désormais $S = \{0, 1, \dots, n-1\}$ et G codé par la liste des liste d'adjacences.

Autrement dit, $G[x]$ est la liste des sommets $y \in S$ tels que $(x, y) \in A$.

On obtient le graphe inversé en utilisant :

```
def inverse(G) :
    L = [] ; n = len(G)
    for k in range(n) : L.append([])
    for x in range(n) :
        for y in G[x] : L[y].append(x)
    return L
```

5. Parcours sur les graphes

a) Notion de bordure, notion de parcours, sommet ouvert et sommet fermé

Définition : Soit $T \subset S$. **La bordure** de T , qu'on note souvent $B(T)$ est l'ensemble des sommets du graphe n'appartenant pas à T mais admettant au moins un prédécesseur dans T .

Définition : Un parcours $L = [x_0, \dots, x_{n-1}]$ de G est une énumération des sommets de sorte que chaque sommet est un successeur de l'un des sommets précédents, s'il en existe un.

Autrement dit, si $T = \{x_0, \dots, x_{i-1}\}$, alors x_i appartient à $B(T)$ si celle-ci n'est pas vide.

Terminologie : Lorsque la bordure est vide, on dit que x_i est un point de régénération du parcours.

Définition : Soit $L = [x_0, \dots, x_{n-1}]$ un parcours de G .

On dit que x_i est **fermé** si tous ses successeurs dans G sont situés avant lui dans L .

Sinon, on dit qu'il est **ouvert** (c'est-à-dire qu'il admet au moins un successeur non parcouru).

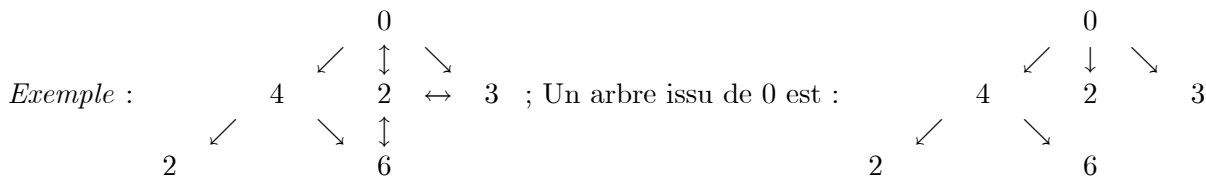
b) Parcours en largeur et en profondeur d'un graphe (*définition formelle*)

Un parcours $L = [x_0, \dots, x_{n-1}]$ est **en largeur** si chaque sommet x_i est un successeur **du premier sommet ouvert** parmi x_0, \dots, x_{i-1} (s'il existe).

Un parcours $L = [x_0, \dots, x_{n-1}]$ est **en profondeur** si chaque sommet x_i est un successeur **du dernier sommet ouvert** parmi x_0, \dots, x_{i-1} (s'il existe).

c) Parcours en largeur de l'arbre issu d'un sommet s dans un graphe

Intuitivement, un parcours en largeur consiste, en partant d'un sommet s , à explorer ses voisins non visités puis les voisins de ses voisins non déjà visités, etc ...



Cet algorithme peut être implémenté à l'aide d'une file :

Étape 1 : On crée la file contenant s comme seul élément ; et on marque s comme visité

Étape 2 : Tant que la file n'est pas vide, on considère le premier élément x de la file

- s'il est fermé, on le défile (et on l'ajoute à la liste mémorisant le parcours)

- s'il est ouvert, on ajoute à la file un de ses successeurs de x non encore visités.

Ce qui revient à remplacer l'Étape 2 par la variante plus concise :

Étape 2 : Tant que la file n'est pas vide, on défile le premier élément x de la file, on ajoute en fin de file la liste de ses successeurs non visités, et on ajoute x à la liste L mémorisant le parcours.

On suppose que le graphe est codé par les listes d'adjacence (donc G est une liste de listes).

On considère la procédure `parcours(G,s)` qui étant donné un graphe G un un sommet s du graphe renvoie la liste des sommets dans l'ordre du parcours en largeur. On utilise un tableau de marquage M pour ne visiter chaque sommet qu'une fois, une liste L donnant la liste des sommets lors du parcours en largeur de l'arbre de racine s .

```

01. def parcoursArbre(G,s) :
02.     n = len(G) ; M = [0]*n ; L = [] ; pere = [-1]*n
        # La liste M est le tableau de marquage et la liste L mémorise le parcours
03.     F = creerFileVide(n) # crée une file vide de capacité n
04.     ajouter(s,F) ; M[s] = 1
05.     while F :
06.         x = defiler(F) ; L.append(x)
07.         for y in G[x] :
08.             if M[y] == 0 :
09.                 ajouter(y,F) ; M[y] = 1
10.     return L

```

Les sommets visités doivent être marqués afin d'éviter qu'un même sommet soit parcouru plusieurs fois. Ainsi, chaque sommet est ajouté et défilé une unique fois dans la file. Donc la ligne 08 est exécutée un nombre de fois correspondant à la somme des arités des sommets visités, c'est-à-dire le nombre d'arêtes.

Interprétation en termes de distance à la racine s :

La distance $d(x)$ d'un sommet s à x est la longueur du plus petit chemin reliant s à x .

Dans un parcours en largeur de racine s , la valeur $d(x)$ croît au fur et à mesure du parcours.

Dans la file, les éléments $F = [y_0, \dots, y_{p-1}]$ vérifient : $d(y_0) \leq d(y_1) \leq \dots \leq d(y_{p-1}) \leq d(y_0) + 1$.

Ainsi, dans la liste renvoyée L , les sommets x sont donc classés par valeurs croissantes de $d(x)$.

Remarque : Une variante consiste donc à construire par récurrence sur k la liste des sommets dont la distance à s vaut k (la distance entre deux sommets est la longueur minimale d'un chemin reliant les deux sommets).

d) Parcours en profondeur de l'arbre issu d'un sommet s dans un graphe

Cet algorithme peut être implémenté à l'aide d'une pile.

Etape 1 : On crée la pile contenant s comme seul élément ; et on marque s comme visité

Etape 2 : Tant que la pile n'est pas vide, on considère l'élément de tête x :

- s'il est fermé, on l'enlève de la pile (et on l'ajoute à la liste mémorisant le parcours)

- s'il est ouvert, on ajoute à la pile un des successeurs de x non encore visités.

```
01.     def parcoursArbre(G,s) :
02.         n = len(G) ; M = [0]*n ; L = []
           # La liste M est le tableau de marquage et la liste L mémorise le parcours
02.         G = G.copy() # on utilise une copie car modifie le graphe dans la suite
03.         pile = [] ; pile.append(s) ; M[s] = 1 ; L.append(s)
04.         while F :
05.             x = tete(pile)
06.             if G[x] :      # si G[x] non vide, c'est-à-dire si x est ouvert
07.                 y = G[x].pop()
08.                 if M[y] == 0 :
09.                     pile.append(y) ; M[y] == 1 ; L.append(y)
10.             else :      # c'est-à-dire que dans ce cas, x est fermé
11.                 x = pile.pop()
```

La liste L renvoie la liste des sommets dans l'ordre du parcours en profondeur de l'arbre.

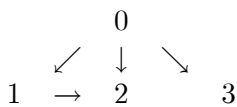
En fait, on peut donner une **version récursive très simple du parcours en profondeur** :

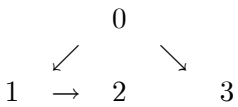
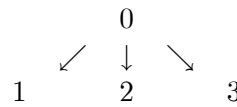
```
01.     def parcoursArbre(G,s) :
02.         n = len(G) ; n = len(G) ; M = [0]*n ; L = []
03.         def traite(x) :
04.             if M[x] == 0 :
05.                 M[x] == 1 ; L.append(x)
06.                 for y in G[x] : traite(y)
07.         traite(s) ; return L
```

Remarque : Si on définit `traite` en dehors de `parcours`, il faut prendre : `traite(x,G,M,L)`

e) Remarque

L'arbre issu d'un parcours en profondeur n'est pas le même que celui issu d'un parcours en largeur.

Par exemple, si G est le graphe  codé par $G = [[1, 2, 3], [2], [], []]$,

les arbres issus de 0 sont  (en profondeur) et  (en largeur).

f) Arbres et chemins explicites

Pour obtenir de façon effective les chemins associés, il suffit lors du parcours de construire le tableau `pere` codant l'arbre :

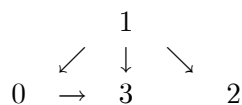
```
01. def parcoursArbre(G,s) :
02.     n = len(G) ; n = len(G) ; M = [0]*n ; pere = [-1]*n
03.     def traite(x) :
04.         if M[x] == 0 :
05.             M[x] == 1
06.             for y in G[x] : pere[y] = x ; traite(y)
07.     pere[s] = s ; traite(s) ; return pere
```

Ensuite, si on veut obtenir le chemin s à x (on suppose qu'un tel chemin existe), on utilise :

```
pere = parcoursArbre(G,s) ; L = [] ; y = x
while y != s : L.append(y) ; y = pere[y]
L.append(s) ; print(reversed(L))
```

g) Parcours en largeur ou en profondeur d'un graphe

Le principe consiste à parcourir **successivement**, pour chaque sommet, l'arbre issu de ce sommet. Les sommets visités étant marqués, chaque sommet n'est visité qu'une fois.

Considérons par exemple le graphe suivant : 

On visite successivement les arbres issus des sommets 0, 1, 2 et 3. L'arbre issu de 0 permet de visiter 0 et 3. Puis on visite les sommets non marqués de l'arbre issu de 1, c'est-à-dire 1 et 2.

Le parcours associé est donc $[0, 3, 1, 2]$. Les sommets de régénération du parcours sont 0 et 1.

Pour définir une fonction `parcours(G)`, on utilise en fait une fonction auxiliaire `parcoursArbre(x,G,M,L)` qui va parcourt l'arbre issu de x dans le graphe des sommets non encore visités (cf tableau de marquage M), met à jour M et ajoute à la liste L la liste des sommets parcourus (dans l'ordre du parcours choisi, en largeur ou en profondeur) :

```
01. def parcours(G) :
02.     n = len(G) ; M = [0]*n ; L = []
03.     for x in range(n) :
```

```

04.         parcoursArbre(x,G,M,L)
05.     return L

```

g) Numérotation postfixe d'un graphe

Considérons la fonction récursive de parcours en profondeur légèrement modifiée :

```

01.     def parcoursArbre(x,G,M,L) :
02.         if M[x] == 0 :
03.             M[x] == 1      # position 1
03.         for y in G[x] :
04.             parcoursArbre(y,G,M,L)
05.             M[x] == 2      # position 2

```

La tableau de marquage prend désormais trois valeurs possibles :

- $M[x] = 0$ signifie : x n'a pas encore été visité
- $M[x] = 1$ signifie : x a été visité mais l'arbre dont il est racine est en cours de visite
- $M[x] = 2$ signifie : x est traité (c'est-à-dire tous ses descendants sont dans L).

Lorsque $M[x]$ vaut 2, il en est de même de tous ses descendants y : en effet, $\text{parcours}(G,y,M,L)$ a de toute façon été effectué avant l'affectation $M[x] = 2$.

Il se peut d'ailleurs qu'il ait été effectué avant que x ne soit visité.

Conséquence : si on place l'instruction : $L.append(x)$ en position 2 (et non pas en position 1), on obtient une liste L dans laquelle tout sommet apparaît après ses successeurs (et donc ses descendants). On dit que la liste est une numérotation postfixe du graphe.

Par exemple $\begin{array}{ccc} & 1 & \\ & \swarrow \downarrow & \\ 0 & \rightarrow 2 & \end{array}$ donne $L = [2, \mathbf{0}, \mathbf{1}]$, où $\mathbf{0}$ et $\mathbf{1}$ sont les sommets de régénération du parcours.

Remarque : Si on place $L.append(x)$ en position 1, on obtient un parcours préfixe de chaque arbre visité, mais pas une numérotation préfixe du parcours :

Par exemple $\begin{array}{ccc} & 1 & \\ & \swarrow \downarrow & \\ 0 & \rightarrow 2 & \end{array}$ donne la liste $L = [\mathbf{0}, 2, \mathbf{1}]$, et 1 n'apparaît pas avant ses fils.

6. Existence de cycles et ordre topologique

a) Ordre topologique dans un graphe acyclique (cf document spécial sur le sujet)

Un ordre topologique dans un graphe orienté $G = (S, A)$ est une numérotation σ des sommets de S vérifiant :

$$\boxed{\forall (x, y) \in S^2, (x \rightarrow y) \Rightarrow (\sigma(x) < \sigma(y))}$$

Un graphe admet un ordre topologique ssi il est acyclique. On peut obtenir un tel ordre par un parcours avec numérotation postfixe : ainsi, tout sommet x est toujours affecté d'un numéro plus grand que tous les numéros

de ses fils, même dans le cas où un fils est traité en amont. En prenant l'ordre inverse, on obtient ainsi une numérotation σ .

La complexité de l'algorithme est linéaire, c'est-à-dire en $O(n + m)$.

b) Existence de cycles dans un graphe orienté

Le test d'existence de cycle peut se faire par l'algorithme précédent : il existe un cycle ssi on est amené à numéroter deux fois un même sommet. La complexité de l'algorithme est donc aussi en $O(n + m)$.

7. Algorithmes de plus courts chemins dans un graphe valué

Remarque : De nombreux algorithmes concernent le calcul de la distance entre deux sommets (c'est-à-dire le chemin le plus court) dans un graphe (orienté ou non). Il est important de distinguer le cas particulier des graphes non valués où la distance est définie par la longueur minimale d'un chemin (la longueur étant le nombre d'arêtes parcourues). Dans ce cas, un parcours en largeur issu d'un sommet x_0 permet de déterminer aisément la distances entre x_0 et les autres sommets : les sommets apparaissent d'ailleurs dans la file dans un ordre croissant en la distance à x_0 .

Ici, on traite le cas général des graphes valués, c'est-à-dire où chaque arête (x, y) est affectée d'un poids $\omega(x, y)$. Le poids d'un chemin est la somme des poids de ses arêtes. On cherche à déterminer les chemins reliant deux sommets donnés et de poids minimum.

Ces algorithmes de plus court chemin sont des exemples typiques de programmation dynamique, basée sur le fait (cf paragraphe b)) que tout sous-chemin d'un chemin optimal est optimal (comme plus court-chemin entre ses extrémités).

Remarque : Le cas des graphes non valués pourrait être vu comme un cas particulier des graphes valués correspondant aux cas où ω est une constante strictement positive.

a) Cycles absorbants

Def : Un cycle $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_{p-1} \rightarrow x_p = x_0$ dans un graphe valué $G = (S, A, \omega)$ est dit absorbant ssi la somme des poids de ses p arêtes est strictement négatif, c'est-à-dire $\sum_{j=1}^p \omega(x_{j-1}, x_j) < 0$.

Dans ce cas, il n'existe pas de plus court-chemin entre deux sommets du cycle, car en effectuant des tours, on peut obtenir des chemins de poids arbitrairement petits.

Remarque : En général, on suppose que les poids $\omega(x, y)$ sont positifs, donc sans cycle absorbant. Mais ce n'est pas toujours le cas. En revanche, pour qu'il existe toujours un plus court chemin (parmi les chemins existants), il faut et il suffit qu'aucun cycle ne soit absorbant. En effet, dans ce cas, en réduisant un chemin (en supprimant les cycles), on diminue son poids. On peut donc se limiter aux chemins réduits qui sont en nombre fini.

On suppose désormais que le graphe valué $G = (S, A, \omega)$ est sans cycle absorbant.

b) Propriété d'optimalité des sous-chemins

Les différents algorithmes présentés ici exploitent le fait que tout sous-chemin d'un plus court chemin est également être un plus court chemin :

Prop : Si $\Gamma : x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_m$ est un plus court chemin de x_0 à x_m , alors pour tous $0 \leq i < j \leq m$, le chemin $\Gamma' : x_i \rightarrow x_{i+1} \rightarrow \dots \rightarrow x_j$ est un plus court-chemin de x_i à x_j

Remarque : En particulier, on a donc pour tout $0 \leq i \leq m$, $d(x_0, x_m) = d(x_0, x_i) + d(x_i, x_m)$.

Preuve : Le poids de Γ s'exprime comme la somme du poids de Γ' et des autres arêtes.

S'il existait un chemin $\Gamma'' : x_i \rightarrow y_1 \rightarrow \dots \rightarrow x_j$ plus court que Γ' , alors le chemin obtenu en remplaçant dans Γ la séquence Γ' par Γ'' serait plus court que Γ , ce qui contredirait la minimalité de Γ .

Remarque : Cette propriété peut ne plus être vérifiée dès lors que des contraintes sont ajoutées. C'est le cas par exemple si on cherche les plus courts chemins empruntant au plus k arcs.

c) Algorithme de Dijkstra (cf document spécifique)

L'algorithme de Dijkstra est valable pour les graphes $G = (S, A)$ valués *positivement* et calcule tous les plus courts-chemins issus d'un sommet s fixé.

La valuation est donnée par la fonction poids $\omega : A \rightarrow \mathbb{N}$. Ainsi, on note $\omega(x, y)$ le poids de l'arête $(x, y) \in A$. Le poids d'un chemin est par définition la somme des poids des arêtes qui le constitue.

On fixe un sommet initial $s \in S$ et on cherche à calculer pour tout sommet le poids minimal d'un chemin reliant s à x (s'il en existe au moins un).

A chaque étape de l'algorithme, on dispose :

- d'une liste L de sommets pour lesquels on connaît la longueur du plus court chemin reliant s à x
- de la bordure B de L dans S
- d'un tableau d de sorte que :
 - $\forall x \in L$, $d[x]$ est le poids minimum d'un chemin de s à x
 - $\forall x \in B$, $d[x]$ est le poids minimum d'un chemin reliant s à x et **dont les sommets intermédiaires** sont dans L .

L est souvent appelée liste close (**closed list**) et B liste ouverte (**open list**).

Pour les sommets de la liste close, les valeurs de d sont définitives.

Remarque : Souvent, quitte à s'autoriser la valeur $\omega(x, y) = +\infty$, on peut considérer le graphe comme complet, et ainsi B est tout simplement le complémentaire de L dans S .

A chaque étape de l'algorithme, **on ajoute à L l'élément x de B pour lequel $d[x]$ est minimal.**

Il convient ensuite de mettre à jour la liste ouverte B en effectuant les opérations suivantes :

- supprimer x de B
- pour tout successeur y de x :
 - s'il est déjà dans L : ne rien faire

- s'il est déjà dans B : mettre à jour $d[y] = \min(d[y], d[x] + \omega(x, y))$
- sinon : on l'ajoute à B avec $d[y] = d[x] + \omega(x, y)$.

On obtient ainsi une complexité en $O(n^2)$: en effet, on utilise n calculs de minimum (en $O(n)$ opérations), n ajouts (à temps constant si on utilise un tableau de marquage), et chaque arête n'intervient qu'une fois dans la mise à jour du tableau c . D'où une complexité totale en $O(n^2 + m) = O(n^2)$, car on a toujours $m \leq n^2$.

Dans le cas de graphes peu denses ($m \ll n^2$), **il est judicieux de gérer B par un tas** (valué par d), ce qui permet un accès immédiat au sommet y minimisant $d[y]$, on obtient $O(m + n \log n)$. En effet, chacune mise-à-jour du tas a une complexité $O(\log n)$, et le nombre de mises-à-jour est exactement le nombre de sommets (pour lesquels il existe un chemin de s à x).

Remarque : L'algorithme de Dijkstra permet aussi de déterminer une arborescence des plus courts chemins. Il suffit de construire au fur et à mesure le parent de chaque élément : **pere[y] = x**.

Variante : Si on cherche seulement la distance entre deux sommets fixés s et z , il suffit d'utiliser le même algorithme (en partant de s) et d'interrompre l'algorithme dès que l'état final s est sélectionné dans S .

d) Algorithme de Floyd-Warshall

L'algorithme de Floyd est valable pour les graphes sans cycle absorbant, c'est-à-dire cycle de poids < 0 (c'est notamment le cas des graphes valués positivement), et cet algorithme calcule *tous* les plus courts-chemins (il y a donc n^2 distances à calculer). Comme tout cycle est de poids positif, les chemins de poids minimum sont des chemins réduits.

Principe : On numérote les sommets, et on détermine à la k -ième étape les distances minimales $d_k(x, y)$ des chemins reliant x à y dont tous les sommets intermédiaires appartiennent aux k premiers sommets.

D'autre part, on initialise $d_0(x, y) = \omega(x, y)$ s'il existe une arête $x \rightarrow y$, et $d_0(x, y) = +\infty$ sinon.

Si z est le k -ième sommet, on a : $\forall(x, y), \boxed{d_k(x, y) = \min(d_{k-1}(x, y), d_{k-1}(x, z) + d_{k-1}(z, y))}$.

On obtient ainsi les poids $d_n(x, y)$ avec une complexité temporelle en $O(n^3)$ et spatiale en $O(n^2)$.

En effet, on peut se contenter pour la k -ième étape de mémoriser le tableau des $d_{k-1}(x, y)$.

Remarque : On s'est contenté ici de calculer les poids, mais on pourrait aussi déterminer des chemins associés (en mémorisant à la chaque étape n^2 chemins optimaux).

Remarque : On peut aussi, en adaptant l'algorithme, calculer dans un graphe valué positivement les plus longs chemins élémentaires reliant deux sommets arbitraires. Il faut alors néanmoins s'assurer à chaque étape que les chemins considérés sont élémentaires.

Remarque : Contrairement aux algorithmes de Dijkstra et de Ford-Bellman, on ne peut pas obtenir seulement les distances d'un sommet fixé x à tous les autres (du fait des termes $d_{k-1}(z, y)$).

Variante : Algorithme de Roy-Warshall : Si on souhaite seulement connaître les accessibilités (c'est-à-dire savoir s'il existe un chemin), on utilise des booléens définis de la façon suivante :

$a_k(x, y)$ vaut **True** ssi il existe un chemin reliant x à y dont tous les sommets intermédiaires appartiennent aux k premiers sommets.

On a alors $a_k(x, y) = a_{k-1}(x, y) \vee (a_{k-1}(x, z) \wedge a_{k-1}(z, y))$, où z est le k -ième sommet.

Le calcul des $a(x, y) = a_{n-1}(x, y)$ se fait donc aussi en $O(n^3)$.

e) Algorithme de Ford-Bellman

L'algorithme de Ford-Bellman est, comme l'algorithme de Floyd, valable pour tous les graphes valués sans cycle absorbant, c'est-à-dire sans cycle de poids total < 0 , et calcule comme l'algorithme de Dijkstra tous les plus courts-chemins issus d'un sommet s fixé.

Etant fixé un sommet x , on détermine à la k -ième étape les $d_k(x)$, où $d_k(x)$ est le poids minimal d'un chemin de s à x comportant au plus k arêtes. Ainsi, on calcule, pour tout y ,

$$d_k(x) = \min(d_{k-1}(x), \min_{(z,x) \in A} (d_{k-1}(z) + \omega(z, x)))$$

La complexité de l'algorithme est en temps $O(nm)$ et en espace $O(n + m)$.

Remarque : La valeur du tableau d_k est stationnaire à partir d'un rang $\leq (n - 1)$.

Variante : Obtention du plus long chemin dans un graphe (supposé sans cycle de poids total > 0).

A chaque étape, on calcule : $m_k(x) := \max(m_{k-1}(x), \max_{z \rightarrow x} (m_{k-1}(z) + \omega(z, x)))$.

Code PYTHON : On peut en fait récupérer pour tout sommet aussi le parent (tableau **pere**) d'un des chemins de longueur minimale joignant 0 à ce sommet s .

```
d = [math.inf]**n ; d[0] = 0 ; pere = [-1]**n
for k in range (n) :
    for i in range (n) :
        for j in range (n) :
            if d[j] > d[i] + w[i,j] :
                d[j] = d[i] + w[i,j] ; pere[j] = i
```

f) Résumé

On considère dans tous les cas un graphe sans cycle absorbant (= de poids total négatif).

On note n le nombre de sommets et m le nombre d'arêtes.

	Dijkstra	Bellman-Ford	Floyd-Warshall
type de chemins	un sommet d'origine	un sommet d'origine	toutes les paires de sommets
poids négatifs possibles	non	oui	oui
temps d'exécution	$O(n \log n + m)$	$O(nm)$	$O(n^3)$
temps si $m = O(1)$	$O(n \log n)$	$O(n)$	$O(n^3)$
temps si $m = \theta(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
temps si $m = \theta(n^2)$	$O(n^2)$	$O(n^3)$	$O(n^3)$

7. Composantes connexes

a) Composantes connexes d'un graphe non orienté

On détermine les composantes connexes dans un graphe *non orienté* en utilisant un parcours en profondeur et deux tableaux de marquage (un pour la composante en cours de traitement et l'autre pour les sommets déjà visités) : La complexité est en $O(n + m) = O(\max(n, m))$, où n et m sont respectivement le nombre de sommets et d'arêtes du graphe.

b) Existence de cycle dans un graphe non orienté

On adapte l'algorithme précédent : il existe un cycle ssi au cours de l'algorithme, on explore un sommet qui est déjà présent dans la composante en cours de traitement. La complexité de l'algorithme est donc aussi en $O(n + m)$.

c) Composantes fortement connexes d'un graphe orienté

La relation $\mathcal{R} : (x \rightarrow^* y \text{ et } y \rightarrow^* x)$ est une relation d'équivalence sur G .

Les composantes fortement connexes sont les classes d'équivalence.

Remarque : Le graphe quotient G/\mathcal{R} est le graphe dont les sommets sont les classes d'équivalence. Deux classes d'équivalence sont reliées si les sommets appartenant à la première classe sont reliés dans G aux sommets de la seconde classe. Le graphe G/\mathcal{R} est acyclique.

Algorithmes de Kosaraju et de Tarjan

Ils permettent en temps linéaire $O(n + m)$ de déterminer les composantes fortement connexes.

8. Coloriages

a) Nombre chromatique et polynôme chromatique

Une coloration d'un graphe G non-orienté est une affectation de couleurs à ses sommets de sorte que deux sommets voisins soient toujours de couleurs différentes.

Le nombre minimum $\chi(G)$ de couleurs pour lequel il existe un coloriage est appelé nombre chromatique du graphe.

L'algorithme glouton de coloriage consiste à attribuer à chaque étape à un sommet non colorié une couleur différente de ses voisins coloriés. Il résulte de cet algorithme glouton que $\chi(G) \leq 1 + d(G)$, où $d(G)$ est le degré maximal des sommets de G .

Théorème de Birkhoff : On note a_k est le nombre de coloriages de G avec k couleurs.

Alors a_k est un polynôme en k de degré n à coefficients entiers, appelé polynôme chromatique de G .

Remarque : L'obtention des coefficients et la preuve se font par récurrence sur le nombre d'arêtes. En particulier, le polynôme chromatique d'une réunion disjointe de graphes est le produit de leurs polynômes chromatiques.

b) Graphes bipartis (= graphes 2-coloriables)

Un graphe $G = (S, A)$ est **biparti ssi il existe une partition $\{X, Y\}$ de S telle que toute arête de G admette une extrémité dans X et l'autre dans Y** . On peut montrer qu'un graphe est biparti ssi il ne contient pas de cycle de longueur impaire (autrement dit, ssi deux chemins ayant mêmes extrémités ont des longueurs de même parité).

La construction d'une partition se fait alors pour chaque composante connexe à l'aide d'un parcours (par exemple en largeur) à partir d'un sommet s : on partitionne les éléments selon la parité de leur distance à s .

9. Circuits eulériens et hamiltoniens

a) Circuits eulériens

Un circuit eulérien est un circuit passant une fois et une seule par chaque arête.

On dit qu'un graphe orienté vérifie la loi de Kirchoff ssi, en chaque sommet, le degré sortant est égal au degré entrant. Un graphe orienté admet un circuit eulérien (circuit passant une fois par chaque arête) ssi il est connexe et vérifie la loi de Kirchoff. **L'obtention d'un éventuel circuit eulérien se fait en temps linéaire.**

Dans le cas du graphe non orienté, il existe un circuit eulérien ssi le graphe est connexe et si chaque sommet est d'arité paire.

b) Circuits hamiltoniens

Un circuit hamiltonien est un circuit passant une fois et une seule par chaque sommet.

L'obtention (dans le cas général) d'un éventuel circuit hamiltonien est en revanche un problème NP-complet (on ne connaît pas d'algorithme de complexité polynomiale).

Exemple : Tout graphe complet est un graphe hamiltonien : il admet un circuit hamiltonien.

10. Arbres de recouvrement d'un graphe connexe non orienté

a) Arbres de recouvrement dans un graphe connexe (forêt de recouvrement d'un graphe)

Un arbre de recouvrement d'un graphe connexe G est un sous-graphe qui est un arbre couvrant tous les sommets.

On peut obtenir un arbre de recouvrement de G par un parcours en profondeur (ou en largeur).

Remarque : De façon générale, un parcours (en largeur ou en profondeur) d'un graphe permet d'obtenir une forêt d'arbres (inclus dans G) recouvrant tous les sommets.

b) Obtention d'un arbre couvrant de poids minimal dans un graphe connexe valué

On suppose que les arêtes du graphe connexe non-orienté sont valuées positivement. Il y a deux algorithmes gloutons déterminant un arbre couvrant de poids minimal (le poids d'un arbre est la somme des poids de ses arêtes) :

- **Algorithme de Kruskal** : On classe en amont les arêtes par poids croissant : à chaque étape on sélectionne l'arête de poids minimal parmi les arêtes dont l'ajout aux arêtes déjà sélectionnées ne crée pas de cycle.

- **Algorithme de Prim** : on ajoute à l'étape k l'arête de poids minimal qui relie un sommet de l'arbre déjà construit à un sommet n'appartenant pas à l'arbre (de sorte que l'ajout de cette arête donne aussi un arbre).