# La structure de graphe

## I - Généralités

### 1) Les graphes orientés

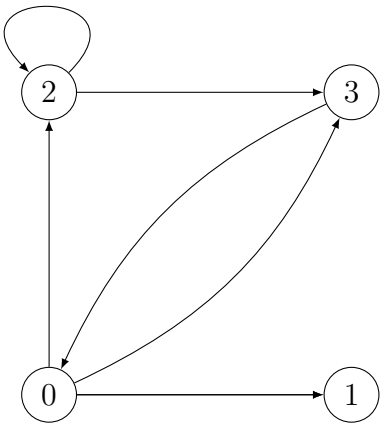
**Définition :** un graphe orienté est un couple $G = (S, A)$ où $S$ est un ensemble et $A$ est une relation binaire sur $S$, i.e. une partie de $S \times S$. Les éléments de $S$ sont appelés les *sommets* ou les *nœuds* du graphe $G$ et les éléments de $A$ sont ses *arcs*. Quelquefois, il est nécessaire de permettre à plusieurs arcs de relier les mêmes sommets, mais cette situation ne semble pas avoir été retenue par le programme officiel.

Nous travaillerons exclusivement sur des graphes finis, i.e. sur des graphes possédant un nombre fini de sommets. L'ordre d'un tel graphe est le nombre de ses sommets. Le programme officiel nous demande aussi d'exclure les `boucles`, i.e. les arêtes de la forme $(s, s)$. Nous considérerons donc la plupart du temps que $A$ ne contient aucun point de la diagonale.

Un graphe orienté peut être représenté par un graphique où les sommets sont des points et où chaque arc $(s, t)$ est une flèche reliant le point associé à $s$ à celui associé à $t$. Ainsi, le graphe :

$$G_1 = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2), (0, 3), (2, 2), (2, 3), (3, 0)\})$$

peut être représenté par le schéma :




**Définitions :** soit $G = (S, A)$ un graphe orienté.

- Si $a = (s, t)$ est un arc de $G$, certains auteurs disent que les sommets $s$ et $t$ sont les *extrémités* de $a$, $s$ étant l'extrémité *initiale* et $t$ l'extrémité *finale*. D'autres disent que $s$ est l'*origine* de $a$ et $t$ son *extrémité* ou sa *destination*. Il faudra donc bien lire les énoncés des épreuves de concours pour éviter les malentendus ! Dans ce cours, je parlerai d'origine et de destination.

- Deux arcs sont adjacents si l'origine de l'un est la destination de l'autre.
- Si $s$ est un sommet du graphe, le *degré entrant* de $s$ est le nombre $d_G^-(s)$ d'arcs qui arrivent en $s$, i.e. le nombre d'arcs de la forme $(t, s)$ ; le *degré sortant* de $s$ est le nombre $d_G^+(s)$ d'arcs qui partent de $s$, i.e. le nombre d'arcs de la forme $(s, t)$.
- Un *chemin* $c$ dans $G$ est une suite de sommets $(s_0, s_1, \ldots, s_k)$ telle que $(s_0, s_1), \ldots, (s_{k-1}, s_k) \in A$. $k$ est la longueur du chemin, $s_0$ son origine et $s_k$ son extrémité. On dit que $c$ est un chemin de $s_0$ à $s_k$, et on peut noter en raccourci $s_0 \xrightarrow{c} s_k$. Par convention, il existe un chemin de longueur nulle d'un sommet $s$ à lui-même, qu'il ne faut pas confondre avec les éventuels chemins $(s, s, \ldots, s)$, si le graphe contient la boucle $(s, s)$. Si les sommets $s_i$ sont deux à deux distincts, on dit que $c$ est un chemin *simple* ou *élémentaire*.
- Le graphe $G$ est dit *fortement connexe* si pour tout couple $(s, t) \in S^2$, il existe un chemin de $s$ à $t$.
- Une *chaîne* dans $G$ est une suite de sommets $(s_0, s_1, \ldots s_k)$ telle que pour tout $i \in [\![1, k]\!]$, $(s_{i-1}, s_i)$ ou $(s_i, s_{i-1})$ est un arc. $k$ est la longueur de la chaîne, $s_0$ son origine et $s_k$ son extrémité. On dit que $c$ est une chaîne de $s_0$ à $s_k$. Par convention, il existe un chaîne de longueur nulle de tout sommet $s$ à lui-même.
- Le graphe $G$ est dit *connexe* si pour tout couple $(s, t) \in S^2$, il existe une chaîne (éventuellement vide si $s = t$) de $s$ à $t$. Le graphe $G_1$ est connexe, mais n'est pas fortement connexe, puisqu'il n'existe pas de chemin de 1 à 0.
- Un *circuit* ou un *cycle* dans $G$ est un chemin $(s_0, s_1, \ldots, s_k)$ non vide tel que $s_0 = s_k$. Un circuit est dit *simple* ou *élémentaire* si $s_1, s_2, \ldots, s_k$ sont distincts.
- Un graphe orienté est dit *acyclique* s'il ne contient pas de cycle.

Si un graphe n'est pas connexe, il est possible de le séparer en "sous-graphes indépendants". Cela se fait par le biais d'une relation d'équivalence :

**Définition-Propriété :** on définit un relation d'équivalence $\sim$ sur $S$ par :

$$\forall s, t \in S,\ s \sim t \iff \text{il existe une chaîne de } s \text{ à } t$$

Les classes d'équivalence définies par cette relation sont les *composantes connexes* du graphe. La classe d'un sommet $s$ pour $\sim$ est la plus grande partie $S'$ de $S$ contenant $s$ et telle que la restriction de $G$ à $S'$ soit connexe.

De la même façon, on peut travailler sur la connexité forte :

**Définition-Propriété :** on définit un relation d'équivalence $\equiv$ sur $S$ par :

$$\forall s, t \in S,\ s \equiv t \iff \text{il existe un chemin de } s \text{ à } t \text{ et un chemin de } t \text{ à } s$$

Les classes d'équivalence définies par cette relation sont les *composantes fortement connexes* du graphe. La classe d'un sommet $s$ pour $\equiv$ est la plus grande partie $S'$ de $S$ contenant $s$ et telle que la restriction de $G$ à $S'$ soit fortement connexe. Le graphe $G_1$ possède 2 composantes fortement connexes : $\{0, 2, 3\}$ et $\{1\}$.

## 2) Représentation informatique des graphes orientés

Il existe deux façons naturelles de représenter en machine les graphes orientés. Pour chacune, nous commencerons par numéroter les sommets pour identifier $S$ à $\{0, 1, \ldots, N-1\}$ où $N$ est l'ordre du graphe.

- Représentation par *listes d'adjacence* : on stocke dans un vecteur de taille $N$ les listes des successeurs de chaque sommet. Ainsi, le graphe $G_1$ est représenté par le vecteur Caml :

  $$G_1 = [|[1; 2; 3]; []; [2, 3]; [0]|]$$

  Cette représentation est efficace du point de vue de l'encombrement mémoire, puisque l'espace utilisé est de l'ordre de grandeur de $|S| + |A|$. Elle peut être coûteuse en temps de calcul, puisqu'il faut parcourir les listes d'adjacence pour obtenir les successeurs d'un nœud (et c'est encore pire si on cherche les prédécesseurs d'un nœud).

- Représentation par *matrice d'adjacence* : on stocke dans une matrice carrée $G$ de taille $N$ des booléens, avec la convention que la case $(i, j)$ contient `true` si et seulement si $(i, j)$ est un arc du graphe. En notant 0 et 1 au lieu de `false` et `true`, le graphe $G_1$ sera représenté par la matrice d'adjacence :

  $$G_1 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

  On répond en temps constant à la question "$(i, j)$ est-il un arc du graphe ?" mais on utilise un espace mémoire de l'ordre de $|S|^2$ pour représenter le graphe.

Ces deux représentations ont leurs avantages et leurs inconvénients, et on privilégiera l'une ou l'autre suivant la nature du graphe (un graphe "creux", i.e. avec peu d'arcs, demandera moins d'espace mémoire avec les listes d'adjacence, mais un graphe "dense" pour lequel $|A|$ est de l'ordre de $|S|^2$ sera avantageusement représenté par sa matrice d'adjacence) et suivant les requêtes courantes que les méthodes utilisées demanderont d'effectuer.

Ainsi, si un graphe orienté est représenté par une matrice d'adjacence $G$ telle que $G[i, j] = 1$ s'il y a un arc de $i$ à $j$ et 0 sinon, la matrice $G^k$, pour $k \in \mathbb{N}$, permet de calculer le nombre de chemin de longueur $k$ permettant de relier deux sommets quelconques du graphe. Ce résultat se prouve facilement par récurrence sur $k$, en notant $n_k(i, j)$ le nombre de chemin de longueur $k$ allant de $i$ à $j$ :

- si $k = 0$, $G^k[i, j]$ vaut 1si $i = j$ et 0 sinon : il y a bien un unique chemin de longueur nulle reliant chaque sommet à lui-même, et aucun chemin de longueur nulle reliant deux sommets différents.

- soit $k \geq 0$ et supposons que pour tout $(i, j)$, $G^k[i, j] = n_k(i, j)$. Pour dénombrer tous les chemins de longueur $k + 1$ reliant $i$ à $j$, on peut regarder chaque successeur $p$ de $i$

et compter les chemins de longueur $k$ allant de $p$ à $j$. Nous aurons donc :

$$n_{k+1}(i,j) = \sum_{p \ t.q. \ (i,p)\in A} n_k(p,j) = \sum_{p=0}^{N-1} G[i,p] n_k(p,j) = \sum_{p=0}^{N-1} G[i,p] G^k[p,j] = G^{k+1}[i,j].$$

La représentation par matrice d'adjacence est également intéressante quand on souhaite manipuler des graphes valués, i.e. des graphes dont les arcs ont des poids. On utilisera alors une matrice $G$ dont l'entrée d'indice $(i,j)$ contient le poids de l'arc $(i,j)$ et $(i,j) \in A$, et une valeur conventionnelle (adaptée au problème étudié) si $(i,j) \notin A$. Dans le cas où les poids représentent un coût et que l'on cherche à trouver un chemin de coût minimal reliant un sommet à un autre sommet, il sera naturel de poser $G[i,j] = +\infty$ si $(i,j) \notin A$.

Certains spécialistes de la théorie des graphes utilisent une représentation par matrice d'incidence *sommets-arcs* : si $G = (S, A)$ est un graphe orienté, sa matrice d'incidence sommets-arcs est une matrice $I$ de taille $|S| \times |A|$ : chaque ligne correspond à un sommet et chaque colonne à une arête, avec la convention :

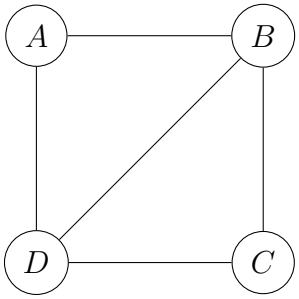
$$\forall\, s \in S,\ \forall a \in A,\ I[s,a] = \begin{cases} 1 & \text{si } i \text{ est l'origine de l'arc } a \\ -1 & \text{si } i \text{ est l'extrémité de l'arc } a \\ 0 & \text{sinon.} \end{cases}$$

## 3) Les graphes non orientés

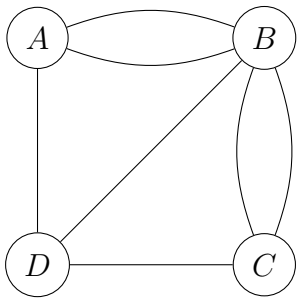
Il arrive parfois que les arcs d'un graphe n'aient pas besoin d'être orientés. Par exemple, dans le problème du voyageur de commerce, on se donne un ensemble de villes $(V_i)_{1\leq i\leq n}$ et le problème consiste à trouver le plus court circuit passant une et une seule fois par chaque $V_i$, à partir de la matrice des distances séparant les villes deux à deux. Chaque arête $(V_i, V_j)$ peut donc être empruntée indifféremment dans le sens $V_i \longrightarrow V_j$ ou $V_j \longrightarrow V_i$.

Un graphe non orienté est ainsi un couple $G = (S, A)$ où $S$ est un ensemble fini et $A$ un ensemble de paires d'éléments de $S$. Les éléments de $S$ sont les sommets du graphe et les éléments de $A$ sont ses *arêtes*. Pour des raisons pratiques, on écrit souvent $(s,t)$ pour représenter la paire $\{s,t\}$, ce qui oblige à confondre $(s,t)$ et $(t,s)$.

On utilise aussi une représentation graphique pour définir des graphes simples, comme dans l'exemple ci-dessous :



Quelquefois, on peut avoir des boucles (i.e. des arêtes reliant un sommet à lui-même) ou plusieurs arêtes reliant deux même sommets : certains auteurs parlent alors de *multigraphes*. C'est le cas du graphe associé au fameux problème des ponts de Kœnigsberg :




On retrouve des définitions à peu près identiques à celles vues dans le cadre des graphes orientés :

**Définitions :** soit $G = (S, A)$ un graphe non orienté.

- Si $a = \{s, t\}$ est une arête de $G$, les sommets $s$ et $t$ sont les *extrémités* de $a$.
- Deux arêtes sont adjacentes si elles ont une extrémité commune.
- Si $s$ est un sommet du graphe, le *degré* de $s$ est le nombre d'arêtes $d_G(s)$qui admettent $s$ pour extrémité.
- Un *chemin* $c$ dans $G$ est une suite de sommets $(s_0, s_1, \ldots, s_k)$ telle que $\{s_0, s_1\}, \ldots, \{s_{k-1}, s_k\} \in A$. $k$ est la longueur du chemin, $s_0$ son origine et $s_k$ son extrémité. On dit que $c$ est un chemin de $s_0$ à $s_k$, et on peut noter en raccourci $s_0 \xrightarrow{c} s_k$. Par convention, il existe un chemin de longueur nulle d'un sommet $s$ à lui-même, qu'il ne faut pas confondre avec les éventuels chemins $(s, s, \ldots, s)$, si le graphe contient la boucle $\{s\}$.
- Le graphe $G$ est dit *connexe* si pour tout couple $(s, t) \in S^2$, il existe un chemin de $s$ à $t$.
- Un *cycle* dans $G$ est un chemin $(s_0, s_1, \ldots, s_k)$ avec $k \neq 0$ et $s_0 = s_k$. Un tel cycle est dit *simple* ou *élémentaire* si les sommets $s_1, s_2, \ldots, s_k$ sont deux à deux distincts.
- Un graphe est dit *acyclique* s'il ne contient pas de cycle.

Si un graphe non orienté n'est pas connexe, on le sépare également en ses composantes connexes :

**Définition-Propriété :** on définit un relation d'équivalence $\sim$ sur $S$ par :

$$\forall s, t \in S,\ s \sim t \iff \text{il existe un chemin de } s \text{ à } t$$

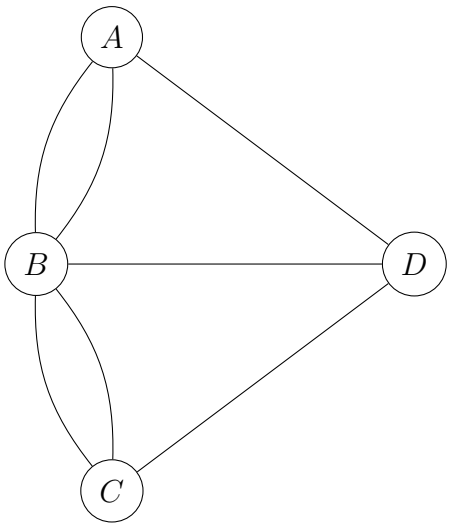
Les classes d'équivalence définies par cette relation sont les *composantes connexes* du graphe. La classe d'un sommet $s$ pour $\sim$ est la plus grande partie $S'$ de $S$ contenant $s$ et telle que la restriction de $G$ à $S'$ soit connexe.

**Remarques :** a) un graphe non orienté peut être modélisé par un graphe orienté dont les arcs sont les couples de la forme $(s,t)$ où $\{s,t\}$ décrit l'ensemble des arêtes du graphe. On peut donc représenter un graphe non orienté par une matrice d'adjacence symétrique.

b) Réciproquement, si $G$ est un graphe orienté, on peut lui associé à un graphe non orienté $G'$ (chaque arc $(s,t)$ donne une arête $\{s,t\}$) : les chaînes de $G$ sont alors les chemins de $G'$ et les notions de connexités dans $G$ et dans $G'$ coïncident.

## 4) Graphes eulériens

Le problème des ponts de Kœnigsberg est sans doute le premier problème traité à l'aide de la notion de graphe : est-il possible pour un promeneur de faire une boucle dans la ville de Kœnigsberg en empruntant une et une seule fois chacun des 7 ponts de la ville ? C'est Euler qui a démontré le premier que le problème n'avait pas de solution. On peut représenter la ville de Kœnigsberg par le graphe




où $A$, $B$, $C$ et $D$ représentent les 4 parties de la ville séparée par la rivière ($A$ est la rive nord, $C$ la rive sud et $B$ et $D$ sont les deux iles), chaque arête représentant un pont. Le problème revient alors à trouver un cycle dans ce graphe qui emprunte une et une seule fois chaque arête, ce qui conduit aux définitions :

**Définition :** soit $G = (S, A)$ un multigraphe (orienté ou non orienté). Un *chemin eulérien* (resp. un *cycle eulérien*) dans $G$ est un chemin (resp. un cycle) qui passe une et une seule fois par chaque arête du graphe. Un graphe possédant un cycle eulérien est appelé *graphe eulérien*. Un graphe est dit *semi-eulérien* s'il n'est pas eulérien, mais s'il possède un chemin eulérien.

Il est assez facile de décrire les graphes eulériens :

**Théorème :**

a) un graphe non orienté connexe est eulérien si et seulement si tous ses sommets sont de degré pairs.

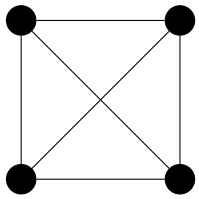
b) un graphe orienté fortement connexe est eulérien si et seulement si pour tout sommet $s$, $d^-(s) = d^+(s)$.

c) un graphe non orienté connexe est semi-eulérien si et seulement s'il possède exactement deux sommets de degrés impairs.

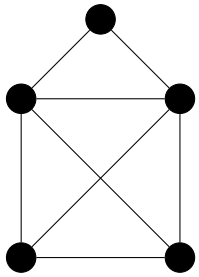
d) un graphe orienté fortement connexe est semi-eulérien si et seulement si $d^-(s) = d^+(s)$ pour tout sommet $s$, sauf pour deux sommets $s_1$ et $s_2$ pour lesquels $d^-(s_1) = d^+(s_1) + 1$ et $d^-(s_2) = d^+(s_2) - 1$ .

On fait la preuve du sens réciproque pour les graphes orientés (la preuve s'adapte facilement aux graphes non orientés), pour le cas eulérien (le cas semi-eulérien est alors une conséquence directe). On travaille par récurrence sur le nombre d'arête $n$, la propriété étant évidente si $n = 0$. Si $n \geq 1$ et si la propriété est démontrée pour tous les graphes orientés possédant strictement moins de $n$ arêtes, alors pour un graphe orienté $G$ connexe à $n$ arêtes tel que $d^-_G(s) = d^+_G(s)$ pour tout sommet $s$, on commence par construire un cycle non vide $C$ (existence facile à prouver) ; si toutes les arêtes n'ont pas été parcourues, on note $G'$ le graphe obtenu en supprimant de $G$ les arêtes utilisées par $C$. On choisit un sommet $s$ de $G'$ qui a été visité par $C$ et tel que $d^+_{G'}(s) \neq 0$ (un tel sommet existe nécessairement) et on complète en un cycle $C'$ de $G'$. La réunion de $C$ et de $C'$ est un cycle de $G$ : on itère ce procédé tant que toutes les arêtes n'ont pas été utilisées.

Ainsi, le graphe de Kœnigsberg n'est pas eulérien puisque les 4 sommets sont de degrés impairs. De même, il n'est pas possible de tracer sans lever le crayon et sans repasser deux fois sur la même arête le dessin :



car les trois sommets du graphe sont une nouvelle fois de degrés impairs. Par contre, on peut tracer "l'enveloppe" sans lever le crayon et sans passer deux fois sur la même arête :



Il faut commencer par un des deux sommets de longueur impaire, c'est-à-dire par un des sommets du bas ... à terminer à la main, avec un papier et un stylo !

**Programmation de la recherche d'une circuit eulérien dans un graphe orienté** : on suppose qu'un graphe orienté $G$ est représenté par un tableau `g` de listes d'adjacence et on cherche à construire un circuit eulérien (en supposant qu'il en existe un). La première étape consiste, à partir d'un graphe $G$ et d'un sommet $s$, à construire un chemin à partir de $s$ en suivant, tant que l'on peut le faire, des arcs non encore utilisés. La fonction `chemin` fait ce travail récursivement, en supprimant les arcs utilisés.

```
let rec chemin g s = match g.(s) with
  | [] -> [s]
  | t::q -> g.(s) <- q; s :: (chemin g t);;
```

Si on suppose que chaque sommet du graphe a même degré entrant que degré sortant, la fonction `chemin` va renvoyer un cycle. En effet, si ce n'était pas le cas, en notant

$$s = s_0 \to s_1 \to \cdots \to s_k$$

ce chemin, il y aurait $N$ arcs sortant de $s_k$ et $N+1$ arcs entrant en $s_k$ : comme il n'existe plus d'arcs sortant de $s_k$, on aurait $d^+(s_k) = N < N+1 \leq d^-(s_k)$. Pour construire un circuit eulérien, on construit un premier cycle `c` à partir d'un sommet quelconque, par exemple 0 ; on applique à à `c` la fonction récursive `optimise`, qui va le compléter en un cycle eulérien ; cette fonction utilise l'analyse suivante :

- si `c` est vide, on renvoie la liste vide (cas terminal) ;
- si `c = s::q` et qu'il n'y a plus d'arc sortant de $s$, on renvoie `s :: (optimise q)` ;
- si `c = s::q` et s'il y a encore un arc sortant de $s$, on applique la fonction `chemin` au sommet `s`, on concatène le chemin obtenu à `q` et on applique la fonction `optimise` : on renvoie donc `optimise ((chemin g s) @ q)`.

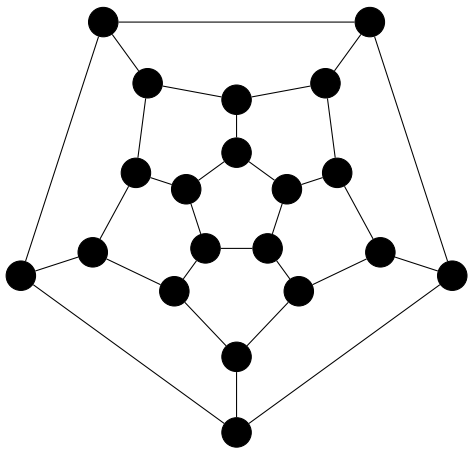
Pour ne pas modifier le graphe, on commence par faire une copie de `g`. Cela donne :

```
let circuit_eulerien g =
  let n = Array.length g in
  let g1 = Array.make n [] in
  for i = 0 to n-1 do
   g1.(i) <- g.(i)
  done;
  let rec optimise l = match l with
   | [] -> []
   | s::q when g1.(s) = [] -> s :: (optimise q)
   | s::q -> optimise ((chemin g1 s) @ q) in
  optimise (chemin g1 0);;
```

## 5) Graphes hamiltoniens

Le problème dual est la recherche d'un cycle simple passant une et une seule fois par chaque sommet du graphe. L'exemple historique a été décrit par Hamilton en 1859, sous la forme d'un

jeu : on considère 20 villes réparties sur le globe terrestre, que l'on assimile pour simplifier au 20 sommets d'un dodécaèdre régulier, et l'on cherche à passer une et une seule fois par chaque ville en utilisant uniquement les arêtes du dodécaèdre, et en revenant à son point de départ. Le graphe des villes est le suivant :



**Définition :** soit $G = (S, A)$ un graphe non orienté. Un *cycle hamiltonien* dans $G$ est un cycle simple passant par chaque sommet de $G$. Un *graphe hamiltonien* est un graphe possédant un cycle hamiltonien.

Le graphe décrit par Hamilton est hamiltonien (il suffit de trouver un cycle hamiltonien pour le démontrer ... à vos crayons !), mais le problème général de l'existence de cycle hamiltonien est beaucoup plus complexe que celui de la recherche de chemin ou de cycle eulérien : c'est un problème NP que l'on ne sait pas résoudre en temps polynomial (et qui n'a peut-être pas de solution en temps polynomial).

# II - Parcours d'un graphe

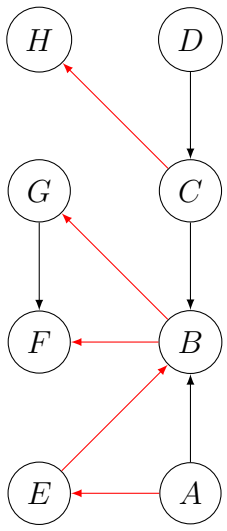
Comme dans le cas des arbres, la résolution d'un problème modélisé par un graphe $G$ nécessitera souvent de parcourir les sommets du graphe. Les parcours que nous allons considérer consistent à suivre les arêtes du graphe pour faire évoluer une partition de $S$ en 3 parties :

- $E$ : ensemble des sommets déjà explorés ;
- $D$ : ensemble des sommets découverts mais pas encore explorés ;
- $I$ : ensemble des sommets qui n'ont pas encore été découvert.

Ces ensembles évoluent de façons différentes suivant les types de parcours, mais le schéma général est le suivant. Au début du calcul, $I = S$ et $E = D = \emptyset$. On utilise également une variable $\mathcal{F}$ qui permet de stocker l'ensemble des arêtes utilisées lors du parcours : $\mathcal{F} = \emptyset$ au début du calcul. On effectue ensuite en boucle, tant que $D$ et $I$ ne sont pas vides, l'opération suivante :

- si $D$ est non vide, on choisit un élément $s$ dans $D$ ; si tous les successeurs de $s$ ont déjà été découverts, on fait passer $s$ de $D$ à $E$. Sinon, on choisit un successeur $t$ de $s$ qui n'a pas été encore découvert : on fait passer $t$ de $I$ à $D$ et on ajoute $(s, t)$ à $\mathcal{F}$.
- si $D$ est vide, on choisit $s$ dans $I$ et on le fait passer de $I$ à $D$.

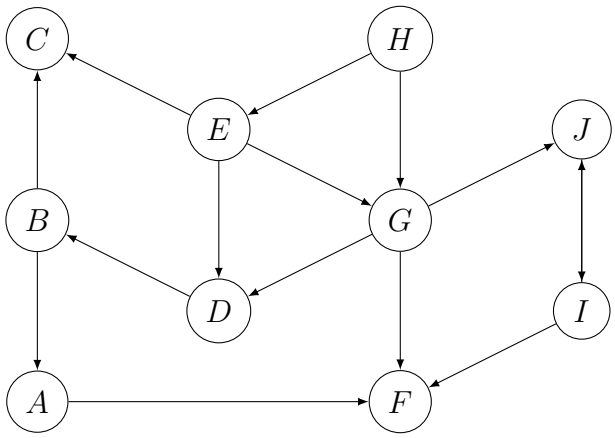
L'algorithme de parcours se termine donc quand $E = S$ et permet d'ordonner les sommets du graphe en une suite $s_1, s_2, \ldots, s_n$ par ordre croissant de leur date de découverte. De plus, le graphe $(S, \mathcal{F})$ est un sous-graphe de $G$ formé d'arbres disjoints ; on dit que c'est une *forêt couvrante* du graphe $G$. Cette forêt est "compatible" avec l'ordre des sommets, dans le sens suivant : si $(s_i, s_j) \in \mathcal{F}$, on a nécessairement $i < j$ (dans le parcours, on découvre un nouveau sommet $t$ en parcourant un arc $(s, t)$ à un moment où $s$ a déjà été découvert). Ainsi, voici un exemple de graphe et de parcours possible, les arcs utilisés pour "découvrir" de nouveaux sommets étant tracés en rouge :




| $E$ | $D$ | $s$ | $t$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $A$ | |
| $\emptyset$ | $\{A\}$ | $A$ | $E$ |
| $\emptyset$ | $\{A, E\}$ | $E$ | $B$ |
| $\emptyset$ | $\{A, E, B\}$ | $A$ | |
| $\{A\}$ | $\{E, B\}$ | $B$ | $F$ |
| $\{A\}$ | $\{E, B, F\}$ | $B$ | $G$ |
| $\{A\}$ | $\{E, B, F, G\}$ | $E$ | |
| $\{A, E\}$ | $\{B, F, G\}$ | $B$ | |
| $\{A, E, B\}$ | $\{F, G\}$ | $F$ | |
| $\{A, E, B, F\}$ | $\{G\}$ | $G$ | |
| $\{A, E, B, F, G\}$ | $\emptyset$ | $C$ | |
| $\{A, E, B, F, G\}$ | $\{C\}$ | $C$ | $H$ |
| $\{A, E, B, F, G\}$ | $\{C, H\}$ | $C$ | |
| $\{A, E, B, F, G, C\}$ | $\{H\}$ | $H$ | |
| $\{A, E, B, F, G, C, H\}$ | $\emptyset$ | $D$ | |
| $\{A, E, B, F, G, C, H\}$ | $\{D\}$ | $D$ | |
| $\{A, E, B, F, G, C, H, D\}$ | $\emptyset$ | | |

## 1) Parcours en largeur (Breadth First Search)

Le parcours en largeur consiste à partir d'un sommet $s_0$ de $G$, à explorer tous les successeurs $s_1, s_2, \ldots, s_k$ de $s_0$, puis les successeurs non encore rencontrés de ces successeurs, et ainsi de suite, jusqu'à ce qu'on ait atteint tous les descendants de $s_0$. Considérons par exemple le graphe :




Un parcours en largeur depuis le nœud $E$ peut par exemple (cela dépend de l'ordre donnés aux fils) parcourir les nœuds dans l'ordre $(E, C, D, G, B, F, J, A, I)$. Ce parcours permet par exemple de calculer la distance de chaque nœud du graphe à un sommet fixé $s_0$, où la distance d'un nœud $s$ à un nœud $t$ est la longueur d'un plus court chemin (s'il en existe) de $s$ à $t$. Il permet aussi de définir une *arborescence de parcours en largeur* de racine $s_0$, i.e. un arbre $\mathcal{T}$ extrait du graphe $G$ tel que pour tout nœud $t$, la profondeur de $t$ dans $\mathcal{T}$ soit la distance de $s_0$ à $t$. Cet arbre $\mathcal{T}$ est représenté, par exemple, par le *tableau des pères* noté $\pi$ : $\pi.(j)$ contient le père de $j$ dans $\mathcal{T}$ s'il existe, et la valeur $-1$ sinon.

Étudions la mise en place de cet algorithme pour un graphe orienté sur l'ensemble $S = \{0, 1, \ldots, n-1\}$ défini par listes d'adjacence : $G$ est donc un vecteur de listes. Le but est, étant donné un sommet fixé $s_0$, de construire les vecteurs $d = [|d(s_0, 0); d(s_0, 1); \ldots, d(s_0, n-1)|]$ (avec pour convention $d(s_0, i) = -1$ s'il n'existe pas de chemin de $s_0$ à $i$) et $\pi$. Une fois initialisé les vecteurs $\pi$ et $d$ de taille $n$ ne contenant que des $-1$, le remplissage peut se faire de deux façons :

- si on dispose d'une structure de file, on crée une file $F$ contenant le sommet $s_0$ et on affecte à la case $d.(s_0)$ la valeur 0. Tant que la file n'est pas vide, on sort l'élément $i$ qui est à sa tête ; on parcourt alors la liste $G.(i)$ des successeurs de $i$ : pour chacun de ces éléments $j$, si $d.(j) = -1$, on affecte à $d.(j)$ la valeur $d.(i) + 1$ et on insère $j$ dans la file $F$ ; sinon, on ne fait rien. Une fois que la file est vide, tous les descendants de $s_0$ ont été étudiés, il suffit de renvoyer $d$. Cela donne :

```
type 'a ldc = Vide | Cellule of 'a cell
    and 'a cell = {mutable avant: 'a ldc; mutable valeur: 'a;
       mutable apres: 'a ldc} ;;

type 'a file = {mutable tete: 'a ldc; mutable queue : 'a ldc} ;;

let cell = function
 | Vide -> failwith "la liste est vide"
 | Cellule(c) -> c;;

let creer_file_vide () = {tete = Vide; queue = Vide} ;;
```

```
let est_vide f = (f.tete = Vide);;

let inserer a f =
  match est_vide f with
   | true -> let l = Cellule {avant = Vide; valeur = a; apres =
      Vide} in
          f.tete <- l;
          f.queue <- l
   | _ -> let l1 = f.queue in
       let l = Cellule {avant = l1; valeur = a; apres = Vide}
          in
        (cell l1).apres <- l;
        f.queue <- l;;

let defiler f =
  match est_vide f with
   | true -> failwith "file␣vide"
   | _ ->  let c = cell(f.tete) in
         match c.apres with
          | Vide -> f.tete <- Vide;
               f.queue <- Vide;
               c.valeur
          | g -> f.tete <- g;
              (cell g).avant <- Vide;
              c.valeur;;

let parcours_largeur g s =
        let d = Array.make (Array.length g) (-1) in
        let pi = Array.make (Array.length g) (-1) in
        let f = creer_file_vide() in
          inserer s f;
          d.(s) <- 0;
          while not(est_vide f) do
           let i = defiler f in
           let l = ref g.(i) in
             while (!l) <> [] do
              let j = List.hd !l in
                l := List.tl !l;
                if d.(j) = (-1) then
                 begin
                 d.(j) <- d.(i)+1;
                 pi.(j) <- i;
                 inserer j f
                 end;
             done;
          done;
          d,pi;;
```

- si on veut éviter l'utilisation d'une file, on travaille uniquement sur des listes : la liste $L$ contient la liste des nœuds qui se trouvent à la distance $k$ de $s_0$ ($k$ est incrémenté tant que de nouveaux nœuds sont découverts). Ainsi, au début du calcul, nous avons $k = 0$ et $L = [s_0]$. Tant que $L$ n'est pas vide, on calcule les successeurs $j$ des nœuds de $L$ qui n'ont pas encore été rencontrés : leur distance à $s_0$ vaut $k + 1$ et ils constituent la nouvelle liste $L$. Cela donne :

```
let parcours_largeur g s =
  let d = Array.make (Array.length g) (-1) in
  let pi = Array.make (Array.length g) (-1) in
  let l = ref [s] and k = ref 0 in  (* d(s,s) = 0 *)
  d.(s) <- 0;
  while (!l) <> [] do
   incr k;
   let nouveaul = ref [] in
   while !l <> [] do  (* tant que la pile L est non vide *)
     let i = List.hd !l in  (* on dépile la tête i de L *)
     l := List.tl !l;
     let fils = ref g.(i) in  (* successeurs de i *)
     while !fils <> [] do
      let j = List.hd !fils in
      fils := List.tl !fils;
      if d.(j) = (-1) then  (* si j n'a pas été rencontré *)
        begin
        d.(j) <- (!k);  (* j est à la distance k de s *)
        pi.(j) <- i;  (* le père de j sera i dans l'arborescence
            finale *)
        nouveaul := j::(!nouveaul)  (* on ajoute j à la "
           nouvelle pile" *)
        end;
     done;
   done;
   l := !nouveaul;  (* L pointe vers la nouvelle pile *)
  done;
  d,pi;;
```

Pour les amateurs de récursivité, on peux supprimer les boucles en utilisant une fonction auxiliaire `ajouter` qui, quand on l'applique à une liste `liste` de nœuds et à une seconde liste de nœuds $[j_1; j_2; \ldots; j_m]$, met à jour le vecteur $d$ pour les $j_i$ non encore rencontrés, et les ajoutent à `liste`. La fonction `nouveau` calcule ensuite la nouvelle liste $L$ quand on l'applique à la liste vide et à l'ancienne liste $L$.

```
let parcours_largeur g s =
  let d = Array.make (Array.length g) (-1) in
  let l = ref [s] and k = ref 0 in
  d.(s) <- 0;
  while (!l) <> [] do
   incr k;
   let rec ajouter liste = function
```

```
      |[] -> liste
      |j::q -> if d.(j) = (-1) then
             begin
             d.(j) <- (!k);
             ajouter (j::liste) q
             end
            else
             ajouter liste q in
     let rec nouveau liste = function
        | [] -> liste
        | i::q -> nouveau (f liste g.(i)) q in
     l := nouveau [] (!l)
    done;
    d;;
```

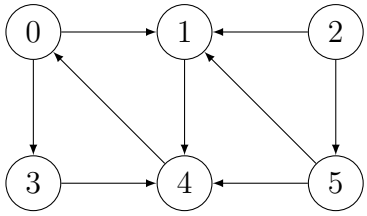
## 2) Parcours en profondeur (Depth First Search)

La seconde stratégie de parcours d'un graphe consiste à descendre le plus profondément possible. On peut faire ce travail à partir d'un nœud particulier $s_0$ (en parcourant uniquement l'ensemble des descendants de $s_0$), ou bien parcourir le graphe entier, que l'on séparera en sous-arbres disjoints (on aura ainsi défini une *forêt couvrante*). Nous détaillons ici cette seconde approche. La méthode est la suivante : on initialise un vecteur $\pi$ qui va contenir à la fin du calcul le tableau des pères de la forêt cherchée. Nous convenons ici de noter $\pi.(i) = -2$ si $i$ n'a pas encore été découvert et $\pi.(i) = -1$ si le nœud $i$ est une racine d'un arbre. Tant que tous les nœuds n'ont pas été "découverts", on choisit un nœud $s_0$, on met à jour le champ $\pi.(s_0)$ et on lui applique la procédure récursive `visiter`. Cette procédure, appliquée à un nœud $i$ qui vient d'être découvert, étudie chaque successeur $j$ de $i$ : si $j$ n'a pas encore été découvert, on associe à $\pi.(j)$ la valeur $i$ ($i$ sera le père de $j$) et on lui applique la procédure `visiter`. Cela donne :

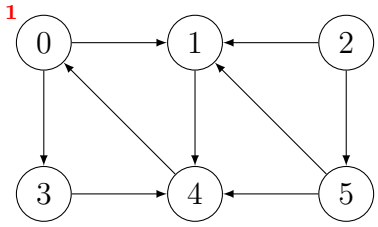
```
let parcours_profondeur g =
  let n = Array.length g in
  let pi = Array.make n (-2) in
  let rec visiter i = let l = ref g.(i) in
   while !l <>[] do
     let j = List.hd !l in
     l := List.tl !l;
     if pi.(j) = (-2) then
      begin
      pi.(j)<-i;
      visiter j
      end
   done in
  for i = 0 to (n-1) do
   if pi.(i) = (-2) then
     begin
```

```
    pi.(i) <- (-1);
    visiter i
    end;
done;
pi;;
```

Dans la pratique, il est intéressant de retenir les instants de découverte et de fin de traitement de chaque nœud. Le temps est discret : l'algorithme commence à l'instant 0 et chaque fois qu'un nouveau nœud est découvert ou que la visite d'un nœud est terminée, on incrémente le temps d'une unité. Précisons ceci dans le cas particulier du graphe :

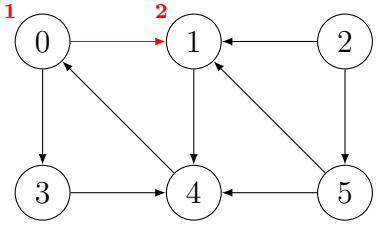



Nous noterons en rouge les dates de découverte des nœuds et en bleue les dates de fin de visite. Les arcs qui constituent la forêt finale sont également représentés en rouge, au fur et à mesure du parcours du graphe.

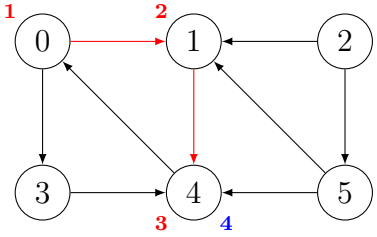



Le nœud 0 est découvert à l'instant 1

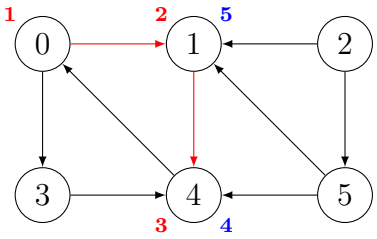
On étudie ensuite les fils de 0, dans l'ordre 1 puis 3, et ainsi de suite :



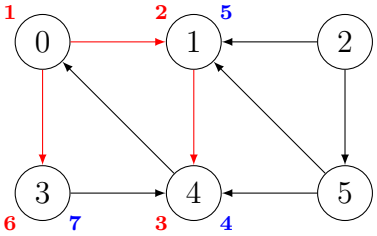

Le nœud 1 est découvert à l'instant 2



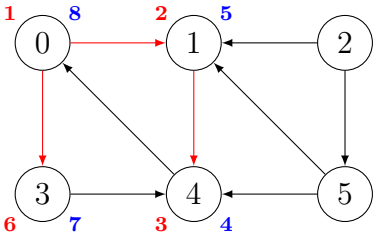

Le nœud 4 est découvert à l'instant 3
et son traitement est terminé à l'instant 4



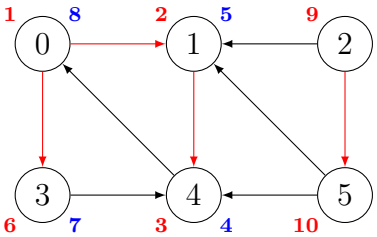

Le traitement du nœud 1 est terminé à l'instant 5
(tous ses fils ont été découverts)



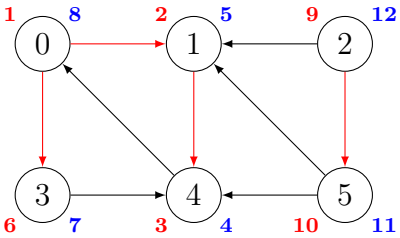

Le nœud 3 est découvert à l'instant 6
et son traitement est terminé à l'instant 7




Le traitement du nœud 1 est terminé à l'instant 8




Le nœud 2 est découvert à l'instant 9
et le nœud 5 à l'instant 10




On termine les traitements des nœuds 5 et 2
aux instants 11 et 12

Ainsi, chaque nœud est associé à deux dates : la date du début de son traitement (moment où le nœud est découvert) et la date de la fin de son traitement (moment où on remonte vers son père). Il est facile de modifier la procédure précédente pour qu'elle renvoie, en plus du vecteur $\pi$, deux vecteurs $d$ et $f$ tels que pour tout $i$, $d.(i)$ et $f.(i)$ contiennent respectivement les dates de début et de fin de traitement du nœud $i$.

Les vecteurs $d$ et $f$ vont jouer un rôle important dans les applications du parcours en profondeur. Si $s$ et $t$ sont deux sommets distincts du graphe, trois cas peuvent se produire :

- les intervalles $[d(s), f(s)]$ et $[d(t), s(t)]$ sont disjoints : les arbres enracinés en $s$ et $t$ et disjoints ($s$ peuvent être dans un même arbre ou dans deux arbres différents de la forêt $\mathcal{F}$ créée par le parcours en profondeur) ;

- $d(s) < d(t) < f(t) < f(s)$ : $t$ est un descendant de $s$ dans l'arborescence $\mathcal{F}$ ;
- $d(t) < d(s) < f(s) < f(t)$ : $s$ est un descendant de $t$ dans l'arborescence $\mathcal{F}$.

On utilise pour cela deux propriétés presque évidentes :

- pour chaque sommet $i$, on effectue une et une seule fois l'appel `visiter i`, à la date $d.(i)$ ;
- pour chaque sommet $i$, l'appel `visiter i` va mettre à jour les champs $pi.(j)$, $d.(j)$ et $f.(j)$ de tous les descendants de $i$ (dans le graphe $G$) qui n'ont pas encore été découverts à l'instant $d.(i)$.

Le parcours en profondeur permet aussi de classifier les arcs du graphe :

- les *arcs de liaison* sont les arcs de la forêt $\mathcal{F}$ ;
- les *arcs arrières* sont les arcs $(s,t)$ où $t$ est un ancêtre de $s$ dans $\mathcal{F}$ ;
- les *arcs avants* sont les arcs $(s,t)$ qui ne sont pas de liaison et où $t$ est un descendant de $s$ dans $\mathcal{F}$ ;
- les *arcs transverses* sont tous les autres arcs (ils peuvent relier deux sommets qui n'appartiennent pas au même arbre de $\mathcal{F}$, ou bien deux sommets d'un même arbre qui ne sont pas descendant l'un de l'autre).

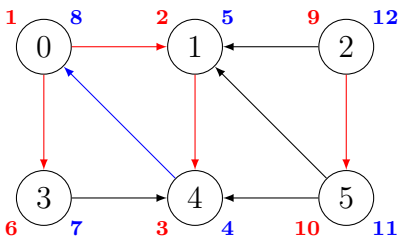
Un arc $(s,t)$ est

- de liaison ou avant si et seulement si $d(s) < d(t) < f(s) < f(t)$ ;
- arrière si et seulement si $d(t) < d(s) < f(s) < f(t)$ ;
- transverse si et seulement si $d(t) < f(t) < d(s) < d(t)$.

```
let parcours_profondeur g =
  let n = Array.length g in
  let pi = Array.make n (-2) in
  let d = Array.make n 0 in
  let f = Array.make n 0 in
  let date = ref 1 in
  let rec visiter i = let l = ref g.(i) in
   d.(i) <- (!date); (* on commence le traitement de i *)
   incr date;
   while(!l)<>[] do
     let j = List.hd !l in
     l := List.tl !l;
     if pi.(j) = (-2) then
      begin
      pi.(j) <- i;
```

```
      visiter j
      end
  done ;
  f.(i)<-(!date); (* on a terminé le traitement de i *)
  incr date; in
for i = 0 to (n-1) do
  if pi.(i) = (-2) then
    begin
    pi.(i) <- (-1);
    visiter i;
    end;
done;
pi,d,f;;
```

Dans l'exemple précédent, nous obtenons le schéma suivant, où les arcs de liaisons sont rouges, l'unique arc arrière est bleu et les arcs transverses sont noirs (il n'y a pas d'arc avant) :




Cette classification des arcs du graphe permet de tester le caractère acyclique d'un graphe : en effectuant un parcours en profondeur d'un graphe, on calcule les vecteurs $d$ et $f$ : le graphe est alors acyclique si et seulement s'il ne contient pas d'arc arrière, i.e. si et seulement si pour tout $(s,t) \in A$, on n'a pas $d(t) < d(s) < f(s) < f(t)$, ce qui revient à dire que pour $(s,t) \in A$, on a soit $d(s) < d(t)$, soit $f(t) < f(s)$. Ceci permet de détecter l'acyclicité au cours du parcours en profondeur dès que l'on détecte un problème ; au moment où on examine un successeur $j$ de $i$, trois cas peuvent se produire :

- (a) $j$ n'a pas encore été découvert : on poursuit le parcours en profondeur ;
- (b) $j$ a déjà été découvert et $d.(i) < d.(j)$ (cela arrive si $j$ est un descendant d'un autre successeur de $i$, qui a été étudié avant $j$) : on poursuit le parcours en profondeur ;
- (c) $j$ a déjà été découvert, $d.(j) < d.(i)$ et $f.(j) = 0$ : le traitement du sommet $j$ n'est pas terminé et celui de $i$ se terminera avant alors que celui de $i$ ne l'est pas encore : on aura $f.(i) < f.(j)$ à la fin du calcul et l'arc $(i,j)$ ne sera pas un arc arrière : on poursuit le parcours en profondeur ;
- (d) $j$ a déjà été découvert, $d.(j) < d.(i)$ et $f.(j)$ n'est pas nul : le traitement du sommet $j$ est terminé alors que celui de $i$ ne l'est pas encore : on aura $d.(j) < d.(i) < f.(i) < f.(j)$ à la fin du calcul et le graphe n'est pas acyclique : on peut arrêter le calcul et renvoyer le booléen `false`.

Pour mettre en place cette sortie un peu sauvage du parcours en profondeur, nous utiliserons une exception de type `Failure`. On supprime également le vecteur `pi` qui n'est plus utile et on obtient :

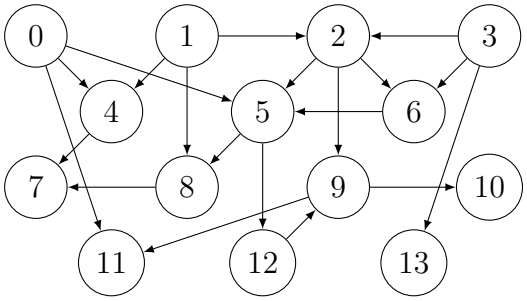
```
let est_acyclique g =
  let n = Array.length g in
  let d = Array.make n 0 in
  let f = Array.make n 0 in
  let date = ref 1 in
  let rec visiter i = let l = ref g.(i) in
   while(!l)<>[] do
     let j = List.hd(!l) in
     l := List.tl(!l);
     match d.(j),f.(j) with
       | 0,_ -> d.(j) <- (!date);
             incr date;
             visiter j
       | k,0 when k < d.(i) -> failwith "pb"   (* on a un cycle *)
       | _ -> ()
   done;
   f.(i) <- (!date);
   incr date; in
  try
   for i = 0 to (n-1) do
     if d.(i) = 0 then
      begin
      d.(i) <- (!date);
      incr date;
      visiter i;
      end;
   done;
   true (* si le calcul termine, le graphe est acyclique *)
  with Failure "pb" -> false;; (* sinon, on récupère l'exception *)
```

## 3) Tri topologique d'un graphe orienté sans circuit

Supposons que $G = (S, A)$ soit un graphe orienté sans circuit. Trier topologiquement $G$ consiste à calculer un ordre total strict $<$ sur l'ensemble $S$ de sorte que pour tout arc $(s, t) \in A$, on ait $s < t$. Un tel tri est par exemple intéressant quand $G$ est un *graphe de tâches* : les éléments de $S$ sont les tâches à accomplir et les éléments de $A$ représentent les contraintes de préséance, i.e. que pour deux tâches $s$ et $t$, $(s, t) \in A$ si et seulement si la tâche $s$ doit être effectuée avant la tâche $t$. Le tri topologique de $G$ permet de calculer un bon ordonnancement des tâches, i.e. un ordre dans lequel on pourra effectuer l'ensemble de toutes les tâches en respectant les contraintes de préséance. Le calcul d'un tel ordre est très facile : il suffit d'appliquer la fonction `parcours_profondeur` et de ranger les sommets de $S$ dans l'ordre décroissant de dates de fin de traitement. Cela se fait à la volée : on crée une

liste initialement vide, dans laquelle on insère chaque sommet à la fin de son traitement. à la fin du calcul, cette liste est de la forme $[s_1; s_2; \ldots; s_n]$ avec $f(s_1) > f(s_2) > \cdots > f(s_n)$ : ceci assure qu'il n'existe pas de chemin de $s_i$ à $s_j$ avec $i < j$ (si $s_j$ est accessible depuis $s_i$, $d(s_i) < d(s_j) < f(s_j) < f(s_i)$).

Considérons le graphe $G_1$ :




L'algorithme de parcours en profondeur donne les tableaux suivants :

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi.(i)$ | -1 | -1 | 1 | -1 | 0 | 0 | 2 | 4 | 5 | 12 | 9 | 9 | 5 | 3 |
| $d.(i)$ | 1 | 19 | 20 | 25 | 2 | 6 | 21 | 3 | 7 | 10 | 11 | 13 | 9 | 26 |
| $f.(i)$ | 18 | 24 | 23 | 28 | 5 | 17 | 22 | 4 | 8 | 15 | 12 | 14 | 16 | 27 |

On peut donc choisir l'ordre $(3, 13, 1, 2, 6, 0, 5, 12, 9, 11, 10, 8, 4, 7)$ pour ordonnancer les tâches représentées par le graphe $G_1$.

Cela donne la procédure réduite suivante, dans laquelle on ne calcule que la liste finale :

```
let tri_topologique g =
  let n = Array.length g in
  let jamais_vu = Array.make n true in
  let pile = ref [] in
  let rec visiter i =
   if jamais_vu.(i) then
     begin
      jamais_vu.(i)<-false;
      List.iter visiter g.(i);
      pile := i::(!pile)
     end in
  for i = 0 to (n-1) do
   visiter i;
  done;
  !pile;;
```

```
# let g1 = [|[4;5;11];[2;4;8];[5;6;9];[2;6;13];[7];[8;12];[5];[];[7];
              [10;11];[];[];[9];[]|];;
```
val g1 : int list vect =
  [|[4 ; 5 ; 11] ; [2 ; 4 ; 8] ; [5 ; 6 ; 9] ; [2 ; 6 ; 13] ; [7] ; [8 ; 12] ; [5] ; [] ; [7] ; [10 ; 11] ; [] ; [] ; [9] ; []|]

```
tri_topologique g1;;
```
- : int list = [3 ; 13 ; 1 ; 2 ; 6 ; 0 ; 5 ; 12 ; 9 ; 11 ; 10 ; 8 ; 4 ; 7]

## 4) Calcul des composantes fortement connexes d'un graphe orienté

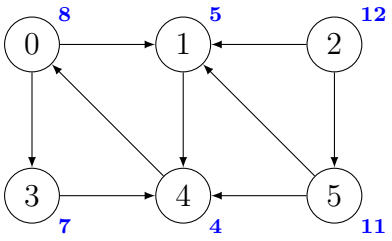
Soit $G = (S, A)$ un graphe orienté ; deux sommets $s$ et $t$ de $G$ sont dits équivalents s'il existe un chemin dans $G$ de $s$ à $t$ et un chemin de $t$ à $s$. Cette relation est une relation d'équivalence, dont les classes d'équivalences sont les *composantes fortement connexes* de $G$. Pour calculer les composantes fortement connexes de $G$, nous pouvons utiliser l'algorithme suivant :

(a) On fait un premier parcours en profondeur du graphe $G$ et on renvoie la liste $p$ des sommets dans l'ordre décroissant des dates de fin de traitement (comme pour le calcul d'un ordre topologique quand le graphe est acyclique) ;

(b) On calcule $G' = (S, E')$, graphe inverse de $G$ (on a un arc de $s$ à $t$ dans $G'$ si et seulement si on a un arc de $t$ à $s$ dans $G$) ;

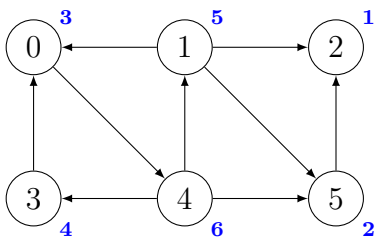
(c) On fait un parcours en profondeur de $G'$ en parcourant $S$ dans l'ordre de $p$ : quand on traite un nouvel élément $s$, les sommets rencontrés pendant le traitement de $s$ sont exactement les sommets de la composante fortement connexe de $s$.

En reprenant l'exemple traité au paragraphe 2 (les nombres bleus sont les instants de fin de traitement) :




$p = [2; 5; 0; 3; 1; 4]$

On inverse le sens des arêtes et on fait un parcours en profondeur :



On commence le parcours en le sommet 2 qui n'a pas de successeur : la première composante connexe est $\{2\}$ ; on poursuit à partir du sommet 5 qui donne également une composante connexe réduite à $\{5\}$, puis on passe au sommet 0, qui permet d'atteindre les sommets 4, 3 et 1 : la dernière composante connexe est $\{0, 1, 3, 4\}$. Les trois derniers sommets de $p$ ont déjà été rencontrés : le parcours en profondeur est terminé.

On peut donc calculer les composantes fortement connexes par le biais des trois fonctions suivantes :

- la fonction `tri` est exactement celle utilisée pour le tri topologique ; la forme a été un peu simplifiée, en utilisant la fonction `do_list` qui permet d'appliquer une fonction à tous les éléments d'une liste ;
- la fonction `inverse` calcule le graphe inversé : on initialise la table des listes d'adjacence du nouveau graphe (au début du calcul, toutes les listes sont vides) puis on parcourt les arêtes du graphe initial en ajoutant, pour chaque arc $(i, j)$, un arc $(j, i)$ ; ce calcul est fait récursivement par la fonction `traitement_succ`, appliquée à un sommet $i$ et la liste de ses successeurs ;
- la fonction `calcul_composantes` fait ensuite le parcours en profondeur du graphe inversé, en traitant les sommets dans l'ordre de $p$ ; on utilise une table `jamais_vu` comme pour la fonction `tri` et une référence `c` initialisée à la liste vide : cette référence va pointer tout au long du calcul sur la liste des composantes connexes déjà construites ; la fonction `composante`, appliquée à un sommet $i$ non encore rencontré, va initialiser une référence `comp` à la valeur $[i]$, puis va ajouter tous les sucesseurs de $i$ à cette pile au moyen de la fonction auxiliaire récursive `parcours` ; une fois tous les successeurs étudiés, `comp` contient une liste des sommets qui forment la composante fortement connexe de $i$ : on ajoute cette liste à `c` ; il suffit ensuite d'appliquer la fonction `composante` aux éléments de la liste $p$ et de renvoyer la liste `!c` qui contient toutes les composantes fortement connexes du graphe.

```
let tri g =
  let n = Array.length g in
  let jamais_vu = Array.make n true in
  let pile = ref [] in
  let rec visiter i =
   if jamais_vu.(i) then
      begin
      jamais_vu.(i) <- false;
      List.iter visiter g.(i);
      pile := i::(!pile)
      end in
  for i = 0 to (n-1) do
   visiter i;
  done;
  !pile;;
```

```
let inverse g =
  let n = Array.length g in
  let g1 = Array.make n [] in
  let rec traitement_succ i liste =
   match liste with
     | [] -> ()
     | j::q -> g1.(j) <- i::g1.(j); traitement_succ i q in
  for i = 0 to n-1 do
   traitement_succ i g.(i)
  done;
  g1;;

let calcul_composantes g =
 let n = Array.length g in
 let g1 = inverse g in
 let jamais_vu = Array.make n true and c = ref [] in
 let composante i =
   if jamais_vu.(i) then
    begin
    jamais_vu.(i) <- false;
    let comp = ref [i] in
    let rec parcours j =
        if jamais_vu.(j) then
        begin
        jamais_vu.(j) <- false;
        comp := j::(!comp);
        List.iter parcours g1.(j)
        end in
                List.iter parcours g1.(i);
                c := (!comp)::(!c)
                end in
  List.iter composante (tri g);
  !c;;
```

### 5) 2-SAT par calcul des composantes fortement connexes : voir X-ENS 2016.

# III - Calcul de chemins de poids minimum

Un *graphe orienté pondéré* (resp. un graphe non orienté pondéré) est un triplet $(S, A, P)$ où $(S, A)$ est un graphe orienté (resp. non orienté) et où $P$ est une application de $A$ dans $\mathbb{R}$ ; si $a$ est une arc (resp. une arête) du graphe, $P(a)$ est appelé *le poids* de l'arc (resp. de l'arête). Nous prolongeons cette fonction aux chemins : si $c = (s_0, s_1, \ldots, s_k)$ est un chemin dans $G$, $P(c) = P(s_0, s_1) + P(s_1, s_2) + \cdots + P(s_{k-1}, s_k)$.

Les graphes pondérés permettent par exemple de modéliser un réseau routier :

- les sommets du graphe sont les lieux reliés par le réseau routier ;
- deux sommets $s$ et $t$ sont reliés par une arête s'il existe une route entre $s$ et $t$ ; le poids de cette arête peut être la longueur, le temps de parcours ou le coût de construction de cette route ;
- on peut aussi imaginer que les poids puissent être négatif (par exemple, si le graphe est orienté et si le trajet se fait avec un véhicule électrique qui se recharge dans les descentes).

Dans la suite de ce chapitre, nous considérons un graphe orienté pondéré $G = (S, A, P)$, avec $S = \{0, 1, \ldots, n-1\}$. Ce graphe sera représenté soit par la « matrice des poids » $P$ :

$$\forall s, t \in S,\ w_{s,t} = \begin{cases} +\infty & \text{si } s \neq t \text{ et } (s,t) \notin A \\ 0 & \text{si } s = t \\ P(s,t) & \text{si } (s,t) \in A \end{cases}$$

soit par listes d'adjacence, i.e. par une table *adj* de taille $n$ telle que pour tout $s \in S$, $adj.(s)$ est la liste $[(t_1, p_1); (t_2, p_2); \ldots; (t_k; p_k)]$ qui représentent les arcs issus de $s$ ($p_i$ étant le poids de l'arc $(s, t_i)$).

Nous nous intéressons exclusivement à la recherche de chemins de poids minimum.

Pour deux sommets $s$, $t$ du graphe, $k \in \mathbb{N}$ et $u \in \{0, 1, \ldots, n\}$, nous noterons :

- $\mathcal{C}_{s \rightsquigarrow t}$ l'ensemble des chemins de $s$ à $t$ ;
- $\mathcal{C}^k_{s \rightsquigarrow t}$ l'ensemble des chemins de $s$ à $t$ de longueur au plus $k$ ;
- $\mathcal{C}^{(u)}_{s \rightsquigarrow t}$ l'ensemble des chemins $c = (s_0, s_1, \ldots, s_k)$ de $s = s_0$ à $t = s_k$ dont les états intermédiaires $s_1, \ldots, s_{k-1}$ appartiennent à $\{0, 1, \ldots, u-1\}$.

Nous notons ensuite, avec la convention $\inf \emptyset = +\infty$ :

- $d^k_{s,t} = \inf\{P(c),\ c \in \mathcal{C}^k_{s \rightsquigarrow t}\}$ ;
- $d^{(u)}_{s,t} = \inf\{P(c),\ c \in \mathcal{C}^{(u)}_{s \rightsquigarrow t}\}$ ;
- $d_{s,t} = \inf\{P(c),\ c \in \mathcal{C}_{s \rightsquigarrow t}\} = d^{(n)}_{s,t}$.

Si le graphe contient un *circuit absorbant*, c'est-à-dire un cycle $c$ de poids strictement négatif, on peut, pour certains $s$ et $t$, trouver des chemins de $s$ à $t$ de poids arbitrairement petits (il suffit de suivre autant de fois que nécessaire le cycle $c$). L'existence d'un circuit absorbant est caractérisé par l'existence d'un sommet $s$ et d'un entier $k \leq n$ tel que $d^k_{s,s} < 0$. En effet, s'il existe un circuit absorbant, il existe alors un circuit absorbant $c$ de longueur minimale $m$. Le

circuit $c$ est simple, car s'il contenait une boucle $c'$, la minimalité de la longueur de $c$ serait contredite ou par $c'$, ou par le circuit $c''$ obtenu en supprimant $c'$ de $c$. Plus précisément, on aurait (avec $0 \leq i < j \leq m$, $j - i < m$, $s_0 = s_m$ et $s_i = s_j$) :

$$c = s_0 \to s_1 \to \cdots \to \underbrace{s_i \to \cdots \to s_j}_{=c'} \to \cdots \to s_m$$

$$c'' = s_0 \to s_1 \to \cdots \to s_i \to s_{j+1} \to \cdots \to s_m$$

et $0 > P(c) = P(c') + P(c'')$, donc $P(c') < 0$ ou $P(c'') < 0$ : absurde par minimalité de $m$.

Si le graphe ne contient pas de circuit absorbant, on montre facilement que, pour tout couple de sommets $s$ et $t$, ou bien il n'existe pas de chemin de $s$ à $t$, ou bien il existe un chemin de $s$ à $t$ de poids minimum, puisqu'on peut restreindre la recherche du minimum à l'ensemble fini des chemins simples de $s$ à $t$, qui sont de longueur au plus $n-1$ (si $(s_0, s_1, \ldots, s_N)$ est un chemin de longueur $N \geq n$, deux des $N+1$ sommets $s_i$ sont égaux et $c$ contient un cycle). Autrement-dit, on a $d_{s,t} = d_{s,t}^{n-1}$ pour tout couple de sommets $(s, t)$.

## 1) Plus court chemin pour une origine $s$ fixée et une extrémité $t$ quelconque

Deux sommets $s$ et $t_0$ étant fixés, on peut montrer que le temps de calcul d'un plus court chemin de $s$ à $t_0$ a une complexité asymptotique du même ordre de grandeur que le calcul de plus courts chemins de $s$ à tous les sommets $t$ du graphe. Nous allons donc décrire dans ce paragraphe des méthodes de calcul d'une arborescence de plus courts chemins depuis une origine $s$ fixée. Plus précisément, nous souhaitons calculer deux tables $d$ et $\pi$ telles que pour tout $t \in S$, $d.(t) = d_{s,t}$ et, quand $s \neq t$ et $d_{s,t} \neq +\infty$, $\pi.(t)$ est le père de $t$ dans un plus court chemin de $s$ à $t$.

Tous les algorithmes classiques utilisent la même approche ; on commence par initialiser les tables $d$ et $\pi$ :

$$\forall t \in S,\ d.(t) = \begin{cases} 0 & \text{si } t = s \\ \\ +\infty & \text{sinon} \end{cases} \qquad \text{et} \qquad \pi.(t) = \begin{cases} -1 & \text{si } t = s \\ \\ -2 & \text{sinon} \end{cases}$$

Tout au long du calcul, on aura toujours :

- pour tout $t$, $d_{s,t} \leq d.(t)$ ;
- $\pi$ définit un sous-arbre de $G$, de racine $s$ : les sommets $t$ tels que $\pi.(t) = -2$ n'appartiennent pas à l'arborescence et si $\pi.(t) \geq 0$, $t$ appartient à l'arborescence et son père est le sommet $\pi.(t)$ ;
- pour chaque sommet $t$, $d.(t)$ est fini si et seulement si $t$ appartient à l'arbre codé par $\pi$, le chemin (dans l'arbre) qui relie $s$ à $t$ étant alors un chemin de $G$ de poids $d.(t)$.

La mise en place en Caml peut se faire en créant un type `poids` (ici, les poids sont des entiers, mais on peut si besoin les remplacer par des flottants) :

```
type poids = N of int | Infini;;

let inferieur p1 p2 =
  match p1,p2 with
   | Infini, _ -> false
   | _, Infini -> true
   | N a, N b -> a<b;;

let somme p1 p2 =
  match p1,p2 with
   | Infini, _ -> Infini
   | _, Infini -> Infini
   | N a, N b -> N(a+b);;
```

La fonction `initialisation` permet d'initialiser $d$ et $\pi$ :

```
let initialiser g s =
  let n = Array.length g in
        let d = Array.make n Infini and pi = Array.make n (-2) in
        d.(s) <- N 0;
        pi.(s) <- (-1);
        d, pi;;
```

On améliore ensuite le couple $(d, \pi)$ en *relâchant* successivement les arêtes du graphe. L'opération de relâchement de l'arête $(t_1, t_2)$ consiste à regarder si l'utilisation de l'arête $(t_1, t_2)$ permet d'améliorer le chemin pour accéder à $t_2$ : si $d.(t_2) > d.(t_1) + P(t_1, t_2)$, on modifie $d.(t_2)$ en $d.(t_1) + P(t_1, t_2)$ et $\pi.(t_2)$ en $t_1$ ; sinon, on ne change rien. L'arête est définie par $t_1$ et $(t_2, P(t_1, t_2))$, cela donne la fonction :

```
let relacher d pi t1 (t2,p) =
  match (somme d.(t1) (N p)) with
          | a when inferieur a d.(t2) -> d.(t2) <- a; pi.(t2) <- t1
          | _ -> ();;
```

### a) Algorithme de Bellman-Ford

Cet algorithme consiste à relâcher successivement tous les arcs, et ceci $n - 1$ fois ; le code ci-dessous met en place l'algorithme de Bellman-Ford.

```
let bellman_ford g s =
  let n = Array.length g and d,pi = initialiser g s in
  for i = 1 to n-1 do
   for t = 0 to n-1 do
     List.iter (relacher d pi t) g.(t)
   done;
```

```
done;
let test d pi t1 (t2,p) = match (somme d.(t1) (N p)) with
 | a when inferieur a d.(t2) -> failwith "circuit negatif detecte"
 | _ -> () in
for t = 0 to n-1 do
 List.iter (test d pi t) g.(t)
done;
let rec chemin t l = match pi.(t) with
 | -1 -> t::l
 | pere -> chemin pere (t::l) in
let f t = match d.(t) with
 | Infini -> failwith "sommet non accessible"
 | N(a) -> a, chemin t [] in
f;;
```

à la fin de ce calcul, deux cas peuvent se produire :

(a) si le graphe ne possède pas de circuit de poids strictement négatif atteignable depuis $s$ , $d$ et $\pi$ répondent au problème posé : $d.(t) = d_{s,t}$ pour tout $t$ et $\pi$ définit une arborescence de plus courts chemins pour l'origine $s$ ;

(b) sinon, il existe au moins un arc $(t_1, t_2)$ tel que $d.(t_2) > d.(t_1) + P(t_1, t_2)$.

Une fois les $n-1$ tours de relâchements effectués, la fonction `test`, appliquée à chaque arc, permet de vérifier si l'on est le cas (b). Si c'est le cas, on renvoie un message d'erreur. Sinon, on renvoie une fonction $f$ permettant de calculer un chemin minimal ; plus précisément, $f$, appliquée à un sommet $t$, renvoie :

- un message d'erreur si $t$ n'est pas accessible depuis $s$ ;
- un couple $(d_{s,t}, c)$ où $c$ est un chemin de poids minimal de $s$ à $t$.

**Preuve :** supposons dans un premier temps que le graphe n'a pas de circuit absorbant atteignable depuis $s$. Pour tout sommet $t$ et tout $i \in \{0, 1, \ldots, n-1\}$, notons $d^i.(t)$ la valeur contenue dans $d.(t)$ après que l'on a effectué $i$ séries de relâchement. On montre alors que l'on a l'invariant de boucle :

$$\mathcal{P}_i : \ \forall t \in S, \ d^i.(t) \leq d^i_{s,t}.$$

- $\mathcal{P}_0$ est vraie, puisque $d^0_{s,t} = \begin{cases} 0 & \text{si } t = s, \\ +\infty & \text{sinon.} \end{cases}$
- Soit $i \in \{0, \ldots, n-2\}$ et supposons $\mathcal{P}_i$ vérifiée ; soit $t \in S$ distinct de $s$ (la propriété est évidente si $t = s$). Si $d^{i+1}_{s,t} = d^i_{s,t}$, on a $d^{i+1}.(t) \leq d^i.(t) \leq d^i_{s,t} = d^{i+1}_{s,t}$. Sinon, il existe un chemin $c$ de longueur au plus $i+1$ dont le poids est égal à $d^{i+1}_{s,t}$. On peut décomposer $c$

en $c'$, chemin de longueur au plus $i$ de $s$ à $t'$, concaténé avec l'arc $(t', t)$ et nous avons $d^{i+1}_{s,t} = p(c) = p(c') + p(t', t)$. Quand l'arc $(t', t)$ est traité lors du $(i+1)$-ème passage dans la boucle, la valeur $d.(t')$ est inférieure ou égale à $d^i.(t')$ (à chaque relâchement, les valeurs stockées dans $d$ ne peuvent que diminuer), et donc $d.(t') \leq d^i_{s,t'} \leq p(c')$. Après le traitement de cet arc, on aura donc

$$d.(t) \leq d.(t') + p(t', t) \leq p(c') + p(t', t) = p(c) = d^{i+1}_{s,t}.$$

Les calculs restant dans la $(i+1)$-ème boucle ne faisant que diminuer $d.(t)$, nous aurons donc $d^{i+1}.(t) \leq d^{i+1}_{s,t}$, ce qui achève la preuve par récurrence.

Ainsi, nous avons à la fin du calcul :

$$\forall t \in S,\ d_{s,t} \leq d.(t) = d^{n-1}.(t) \leq d^{n-1}_{s,t} = d_{s,t}$$

et $\pi$ contient une arborescence de plus courts chemins depuis $s$. Ceci prouve également que pour tout arc $(t_1, t_2)$, on a $d.(t_2) \leq d.(t_1) + p(t_1, t_2)$ : le test final ne va donc pas engendrer de message d'erreur et la fonction va renvoyer la fonction voulue $f$.

S'il existe un circuit absorbant atteignable depuis $s$, le test final va nécessairement renvoyer un message d'erreur. En effet, dans le cas contraire, on pourrait effectuer un nombre quelconque de boucles sur $i$ et le vecteur $d$ ne serait plus modifié à partir de l'étape $n-1$ ; l'invariant de boucle donnerait toujours $d^i.(t) \leq d^i_{s,t}$ pour tout $i \in \mathbb{N}$ et pour tout $t \in S$, ce qui est absurde car s'il existe un circuit absorbant de $t_1$ à $t_2$ avec $s$ atteignable depuis $s$, $d^i_{s,t_2}$ tend vers $-\infty$ quand $i$ tend vers $+\infty$.

**Complexité :** le temps du relâchement étant constant, le calcul se fait en un temps de l'ordre de $|S| \times |A|$, qui est de l'ordre de $n^3$ pour les graphes denses ou de l'ordre de $n^2$ pour les graphes peu denses.

**Remarque :** on peut évidemment arrêter la boucle si aucune modification de distance n'a été faite lors d'un passage de relâchement des arêtes (on pourra même dans ce cas se passer du test final). Cela donne :

```
let relacher_bis d pi t1 b (t2,p) =
  match (somme d.(t1) (N p)) with
          | a when inferieur a d.(t2) -> d.(t2) <- a; pi.(t2) <- t1;
              b := false;
          | _ -> ();;

let bellman_ford_bis g s =
  let n = Array.length g and d,pi = initialiser g s and b = ref true
      and i = ref 1 in
  while !i < = n && !b do
   for t = 0 to n-1 do
     List.iter (relacher_bis d pi t b) g.(t)
   done;
  done;
```

```
  if !b then
   failwith "circuit␣negatif␣detecte" ;
  let rec chemin t l = match pi.(t) with
   | -1 -> t::l
   | pere -> chemin pere (t::l) in
  let f t = match d.(t) with
   | Infini -> failwith "sommet␣non␣accessible"
   | N(a) -> a, chemin t [] in
  f;;
```

### b) Cas des graphes sans circuit atteignable depuis $s$

Si le graphe est sans circuit depuis $s$, on peut se contenter de relâcher une fois chaque arc partant des sommets atteignables depuis $s$, à condition de traiter les sommets dans un ordre topologique. Cela donne :

```
let tri_topologique g s =
  let n = Array.length g in
         let jamais_vu = Array.make n true in
         let pile = ref [] in
         let rec visiter (i,p) =
         if jamais_vu.(i) then
                        begin
                        jamais_vu.(i)<-false;
                        List.iter visiter g.(i);
                        pile := i::(!pile)
                        end in
         visiter (s,0);
         !pile;;

let plus_courts_chemins_sans_circuits g s =
        let d,pi = initialiser g s in
        let l = tri_topologique g s in
        let traiter t = List.iter (relacher d pi t) g.(t) in
        List.iter traiter l;
        let rec chemin t l = match pi.(t) with
                | -1 -> t::l  (* l sert d'accumulateur pour
                    construire le chemin de s à t *)
                | pere -> chemin pere (t::l) in
        let f t = match d.(t) with
                | Infini -> failwith "sommet␣non␣accessible"
                | N(a) -> a, chemin t [] in
        f;;
```

**Complexité :** comme le tri topologique, ce calcul demande un temps de l'ordre de $|S| + |A|$

### c) Algorithme de Dijkstra

Quand les masses sont toutes positives, il suffit de relâcher chaque arc une seule fois, en parcourant les sommets dans un ordre bien choisi. Pour cela, nous allons partitionner dynamiquement $S$ en deux parties :

- une partie $S_1$ constituée des sommets pour lequel un chemin optimal a été calculé ; quand, à un moment du calcul, $t$ est élément de $S_1$, alors $d.(t) = d_{s,t}$ et $\pi$ permet de reconstruire un plus court chemin de $s$ à $t$ (tous les sommets par lesquels passe ce chemin sont dans $S_1$) ;
- $S_2$ est le complémentaire de $S_1$ ; pour $t \in S_2$, on a deux possibilités :
  (a) $d.(t) = \infty$ et $\pi.(t) = -2$ : il n'existe aucun arc d'un sommet de $S_1$ à $t$ ; autrement-dit, aucun chemin n'a encore été détecté permettant de relier $s$ à $t$ ;
  (b) $d.(t) \neq \infty$ et $\pi.(t) = t_1$ : $t_1 \in S_1$ et si $c$ est un chemin minimal de $s$ à $t_1$, de poids $d_{s,t_1}$, alors le chemin obtenu à ajoutant à $c$ l'arc $(t, t_1)$ est un chemin de poids minimal de $s$ à $t$ parmi tous les chemins de la forme $(s_0, s_1, \ldots, s_m)$ avec $s_0 = s$, $s_0, s_1, \ldots, s_{m-1} \in S_1$ et $s_m = t$.

Si, à un moment du calcul, ces propriétés sont vérifiées et si $S_2$ est non vide, alors on calcule $t \in S_2$ tel que $d.(t)$ soit minimal : on est alors certain que $d.(t) = d_{s,t}$ et que $\pi$ permet de construire un chemin optimal de $s$ à $t$ ; en faisant passer $t$ de $S_2$ à $S_1$, la propriété (a) sera préservée. Il suffit ensuite de relâcher tous les arcs qui partent de $t$ pour que (b) soit également préservée. On peut remarquer que l'on peut aussi relâcher un arc $(t_1, t)$ avec $t_1 \in S_1$, puisque ce relâchement ne produira aucun effet.

Une mise en place élémentaire donne :

```
let dijkstra g s =
  let n = Array.length g in
  let s1 = Array.make n false in
  let d,pi = initialiser g s in
  let minimum () =
   let t = ref (-1) and m = ref Infini in
   for i = 0 to n-1 do
     if (not s1.(i)) && inferieur d.(i) !m then
      begin
      t := i;
      m := d.(i)
      end
   done;
   !t in
  let t = ref (minimum()) in
  while !t <> -1 do
   s1.(!t) <- true;
   List.iter (relacher d pi !t) g.(!t);
   t := minimum()
```

```
done;
let rec chemin t l = match pi.(t) with
 | -1 -> t::l  (* l sert d'accumulateur pour construire le chemin
    de s à t *)
 | pere -> chemin pere (t::l) in
let f t = match d.(t) with
 | Infini -> failwith "sommet non accessible"
 | N(a) -> a, chemin t [] in
f;;
```

Cette mise en place donne un temps de calcul de l'ordre de $|S|(m+1)+m+|A_1|$, où $m$ est le cardinal final de $S_1$ et $A_1$ l'ensemble des arcs $(t_1, t_2)$ avec $t_1 \in S_1$ (on applique $m+1$ fois la fonction `minimum` qui est de coût $\Theta(n)$ et les arcs de $A_1$ sont relâchés une et une seule fois). Cela donne donc un temps $O(|S|^2+|A|)$, i.e. $O(|S|^2)$. Pour améliorer ce temps (pour les graphes peu denses), il faut utiliser une structure de file d'attente, qui permet de récupérer et de supprimer le minimum en temps constant $O(\ln n)$, puis de mettre à jour la file d'attente à chaque relâchement en temps $O(\ln n)$. Ainsi, nous aurons un temps de calcul en $O(\ln|S|\times|S|+\ln|S|\times|A|)$, soit $O(|A|\times\ln|S|)$ (en supposant que le nombre d'arc est au moins de l'ordre du nombre de sommet). On peut ensuite, en utilisant un tas de Fibonacci, faire le calcul en un temps $O(|A|+|S|\ln|S|)$ (voir Cormen-Leiserson-Rivest-Stein).

## 2) Plus court chemin pour tous couples de sommets.

Considérons maintenant un graphe orienté pondéré sans cycle absorbant, de sommets $S = \{0, 1, \ldots, n-1\}$ représenté par sa matrice la "matrice des poids" $W$ dont le coefficient d'indice $(s,t) \in S^2$ est défini par :

$$W[s,t] = \begin{cases} \infty & \text{si } s \neq t \text{ et } (s,t) \notin A \\ 0 & \text{si } s = t \\ P(s,t) & \text{si } (s,t) \in A \end{cases}$$

Pour calculer la *matrice des distances* $D = (d_{s,t})_{s,t\in,S}$ et, éventuellement, des chemins minimaux entre tous couples de sommets, nous pouvons appliquer les algorithmes précédents en faisant varier $s \in S$. Il est cependant possible de diminuer le coût de calcul, en utilisant l'une des deux méthodes générales suivantes.

### a) Calcul matriciel

Pour tout $m \in \mathbb{N}$, notons $W^k$ la matrice dont le coefficient d'indice $(s,t)$ vaut $d^k_{s,t}$. Ainsi, $D = W^{n-1}$, puisque $d_{s,t} = d^{n-1}_{s,t}$ pour tout $(s,t) \in S^2$ (quand le graphe est sans cycle

absorbant). Le calcul de $W^m$ peut alors se faire par récurrence, puisque :

$$W_0 = \begin{pmatrix} 0 & \infty & \dots & \infty \\ \infty & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty \\ \infty & \dots & \infty & 0 \end{pmatrix} \text{ et } \forall s,t \in S,\ \forall k \in \mathbb{N},\ W^{k+1}[s,t] = \min_{u \in S} \left( W^k[s,u] + W[u,t] \right) \quad (1)$$

On reconnaît ici la définition d'un produit matriciel un peu particulier : si $A$ et $B$ sont deux matrices carrées de taille $n$ dont les coefficients sont des nombres (on considère que $\infty$ est un nombre), on note $A \star B$ la matrice carrée de taille $n$ définie par :

$$\forall s,t \in S,\ A \star B[s,t] = \min_{u \in S} \left( A[s,u] + B[u,t] \right).$$

Ce produit est associatif et admet la matrice $W_0$ pour élément neutre : la notation $W^k$ prend tout son sens, puisque $W^k$ est bien le produit de $W$ $k$ fois avec lui-même. Le calcul de $A \star B$ se faisant en un temps de l'ordre de $n^3$ (il faut un temps de l'ordre de $n$ pour calculer chaque coefficient), nous obtenons donc $D = W^{n-1}$ en un temps de l'ordre de $n^4$.

Dans le cas des graphes peu denses, on peut améliorer le calcul de $D$ en remarquant que le minimum calculé en appliquant la formule (1) peut s'obtenir en ne parcourant que les prédécesseurs du sommet $t$. Le calcul de $W^k \star W$ demande alors un temps de l'ordre de $|S| \times (|A| + 1)$ (pour chaque $s$ et $t$, le temps de calcul de $d^{k+1}_{s,t}$ est de l'ordre de $d^-(t) + 1$), ce qui permet d'obtenir $D$ en un temps de l'ordre de $|S|^2|A|$ (quand $A$ est non vide).

Il est possible de calculer en même temps les chemins minimaux, par le biais du calcul de matrice $\pi^k$, telle que pour tout $(s,t)$, $\pi^k[s,t]$ est le père de $t$ dans un chemin minimal de longueur au plus $k$ (et parmi les chemins minimaux de longueur $\leq k$, le chemin choisi est de longueur minimal). On utilise une unique matrice $\pi$, qui contient $\pi^k$ à l'étape $k$ ; pour ne pas utiliser trop d'espace mémoire, on utilise 2 matrices : l'une contient $W^k$ et la seconde est utilisée pour calculer $W^{k+1}$. On utilise des références sur ces matrices pour pouvoir ensuite "remplacer" $W^k$ par $W^{k+1}$. Cela donne :

### (i) Calcul de la matrice des distances

La fonction `multiplie` calcule le produit de `!a1` et de `b`, stocke le résultat dans `!a2`, puis échange les deux références `a1` et `a2`. Nous l'appliquerons avec `b` $= W$ et `!a1` $= W^k$ pour qu'à la fin du calcul, `a1` pointe sur $W^{k+1}$.

```
let multiplie a1 b a2 =
  let n = Array.length !a1 in
  for s = 0 to n-1 do for t = 0 to n-1 do
   let m = ref Infini in
   for u = 0 to n-1 do
     match somme !a1.(s).(u) b.(u).(t) with
     | p when inferieur p (!m) -> m := p
     | _ -> ()
   done;
```

```
    !a2.(s).(t) <- !m
  done; done;
  let ancien_a1 = !a1 in
   a1 := !a2;
   a2 := ancien_a1;;
```

On suppose ensuite que le graphe est donné pas listes d'adjacence : on définit la matrice `w` (on utilise la fonction auxiliaire `f` qui modifie les poids dans la matrice `w`), les références `a1` et `a2` et on applique $n-1$ fois la fonction `multiplie` :

```
let calcul_distances g =
  let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 0 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  let a1 = ref (Array.make_matrix n n Infini) in
  let a2 = ref (Array.make_matrix n n Infini) in
  for s = 0 to n-1 do
   (!a1).(s).(s) <- N 0;
  done;
  for k = 1 to n-1 do
   multiplie a1 w a2
  done;
  !a1;;
```

### (ii) Calcul des plus courts chemins

La fonction `multiplie_bis` prend en plus le paramètre `pi` qu'elle met à jour. On remarquera qu'il faut initialiser `m` à la valeur `!a1.(s).(t)`, et pas à la valeur `Infini`, pour ne pas modifier `pi.(s).(s)`, qui doit toujours prendre la valeur $-1$.

```
let multiplie_bis a1 b a2 pi  =
  let n = Array.length !a1 in
  for s = 0 to n-1 do for t = 0 to n-1 do
   let m = ref !a1.(s).(t) in
   for u = 0 to n-1 do
     match somme !a1.(s).(u) b.(u).(t) with
     | p when inferieur p (!m) -> pi.(s).(t) <- u ; m := p
     | _ -> ()
   done;
   !a2.(s).(t) <- !m;
  done; done;
  let ancien_a1 = !a1 in
   a1 := !a2;
   a2 := ancien_a1;;
```

On calcule ensuite les plus courts chemins en appliquant $n-1$ fois la fonction `multiplie_bis` et en renvoyant la fonction $f$ qui calcule, pour deux sommets quelconques $s$ et $t$, le couple $(d, L)$ où $d = d_{s,t}$ et $L$ est un chemin de $s$ à $t$ de poids minimal (un message d'erreur est renvoyé si $s$ et $t$ ne sont pas connectés).

```
let plus_courts_chemins g =
  let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 1 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  let a1 = ref (Array.make_matrix n n Infini) in
  let a2 = ref (Array.make_matrix n n Infini) in
  let pi = Array.make_matrix n n (-2) in
  for s = 0 to n-1 do
   (!a1).(s).(s) <- N 0;
    pi.(s).(s) <- (-1) ;
  done;
  for k = 1 to n-1 do
   multiplie_bis a1 w a2 pi;
  done;
  let f s t =
   let rec g s t l = match s = t with
     | true -> t::l
     | _ -> g s pi.(s).(t) (t::l) in
   match !a1.(s).(t) with
     | Infini -> failwith "les sommets ne sont pas reliés"
     | N p -> p, g s t []in
  f;;
```

### (iii) Détection de l'existence d'un cycle absorbant

Le calcul des $W^k$ permet également de détecter l'existence d'un cycle absorbant, puisqu'un tel cycle existe si et seulement si $W^n$ a au moins un terme diagonal strictement négatif (ou, ce qui est équivalent, si $W^n \neq W^{n-1}$). Voici une fonction qui teste si le graphe passé en argument possède un circuit absorbant ; la fonction `multiplie_ter` renvoie le booléen `true` si la nouvelle matrice $W^{k+1}$ a un terme diagonal strictement négatif.

```
let multiplie_ter a1 b a2 =
  let n = Array.length !a1 and bool = ref false in
  for s = 0 to n-1 do for t = 0 to n-1 do
   let m = ref Infini in
   for u = 0 to n-1 do
     match somme !a1.(s).(u) b.(u).(t) with
     | p when inferieur p (!m) -> m := p
     | _ -> ()
```

```
    done;
    !a2.(s).(t) <- !m;
    bool := !bool or (s = t && inferieur !m (N 0));
  done; done;
  let ancien_a1 = !a1 in
   a1 := !a2;
   a2 := ancien_a1;
   !bool;;

let detection_circuit_absorbant g =
  let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 0 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  let a1 = ref (Array.make_matrix n n Infini) in
  let a2 = ref (Array.make_matrix n n Infini) in
  for s = 0 to n-1 do
   (!a1).(s).(s) <- N 0;
  done;
  let k = ref 0 in
  while (!k < n) && not(multiplie_ter a1 w a2) do
   incr k
  done;
  !k <> n;;
```

### (iv) Exponentiation rapide

Le produit $\star$ étant associatif, on peut calculer $W^n$ en effectuant de l'ordre de $\ln n$ multiplications, ce qui permet de calculer $D$ en un temps $O(n^3 \ln n)$. On utilise deux références `a1` et `a2`, qui pointent sur des matrices, et une référence entière `k` ; au cours du calcul, si `!k` $= k$, on a `!a1` $= W^k$ et $k$ est doublé à chaque étape (le passage de $W^k$ à $W^{2k}$ se fait par le biais de la fonction `carre`). Le calcul s'arête quand $k \geq n$ : il reste à vérifier que la matrice `! a1` n'a pas de terme strictement négatif sur la diagonale. Cela donne :

```
let carre a1 a2 =
  let n = Array.length !a1 in
  for s = 0 to n-1 do for t = 0 to n-1 do
   let m = ref Infini in
   for u = 0 to n-1 do
     match somme !a1.(s).(u) !a1.(u).(t) with
     | p when inferieur p (!m) -> m := p
     | _ -> ()
   done;
   !a2.(s).(t) <- !m
```

```
  done; done;
  let ancien_a1 = !a1 in
   a1 := !a2;
   a2 := ancien_a1;;

let calcul_distances_rapide g =
  let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 0 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  let a1 = ref w in
  let a2 = ref (Array.make_matrix n n Infini) in
  let k = ref 1 in (* a1 pointe sur W^k *)
  while !k < n do
   carre a1 a2;
   k := 2*(!k)
  done;
  for s = 0 to n-1 do
   if inferieur !a1.(s).(s) (N 0) then failwith "le graphe contient
      un circuit absorbant"
  done;
  !a1;;
```

**Remarque :** il est possible, une fois connue la matrice $D$, de calculer un plus court chemin entre deux sommets du graphe.

### b) Algorithme de Floyd-Warshall

Cet algorithme utilise une approche de type "programmation dynamique", en calculant les $d_{s,t}^{(u)}$ grâce aux propriétés, valables pour un graphe sans circuit absorbant :

$$\forall s,t,u \in S,\ d_{s,t}^{(0)} = W[s,t] \text{ et } d_{s,t}^{(u+1)} = \min\left(d_{s,t}^{(u)}, d_{s,u}^{(u)} + d_{u,t}^{(u)}\right).$$

En notant $D^{(u)}$ la matrice de terme général $d_{s,t}^{(u)}$, le calcul de $D^{(u+1)}$ à partir de $D^{(u)}$ se fait en temps quadratique, et on obtient $D = D^{(n)}$ en un temps de l'ordre de $|S|^3$.

### (i) Calcul de la matrice des distances

Comme dans le paragraphe précédent, on utilise deux références sur des matrices pour minimiser l'espace mémoire nécessaire au calcul des différents $D^{(u)}$. Si `a1` pointe sur la matrice $D^{(u)}$ (avec $0 \leq u < n$), alors après l'appel `calcul a1 a2 u`, `a1` pointera sur $D^{(u+1)}$.

```
let calcul a1 a2 u =
  let n = Array.length !a1 in
```

```
  for s = 0 to n-1 do for t = 0 to n-1 do
   match !a1.(s).(t),somme !a1.(s).(u) !a1.(u).(t) with
     | x,y when inferieur x y -> !a2.(s).(t) <- x
     | _,y -> !a2.(s).(t) <- y
  done; done;
  let ancien_a1 = !a1 in
   a1 := !a2;
   a2 := ancien_a1;;

let floyd_warshall g =
         let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 0 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  let a1 = ref w in
  let a2 = ref (Array.make_matrix n n Infini) in
  for u = 0 to n-1 do
   calcul a1 a2 u
  done;
  !a1;;
```

En fait, on peut calculer $D$ en modifiant une unique matrice :

```
let floyd_warshall_bis g =
  let n = Array.length g in
  let w = Array.make_matrix n n Infini in
  let f s (t,p) = w.(s).(t) <- N p in
  for s = 0 to n-1 do
   w.(s).(s)<- N 0;
   List.iter (f s) g.(s)
  done;
  for u = 0 to n-1 do
   for s = 0 to n-1 do for t = 0 to n-1 do
     match w.(s).(t),somme w.(s).(u) w.(u).(t) with
     | x,y when inferieur x y -> w.(s).(t) <- x
     | _,y -> w.(s).(t) <- y
   done; done;
  done;
  w;;
```