

Chapitre 2. Graphes.

Introduction et objectifs du chapitre

Un graphe permet de représenter les connexions d'un ensemble en indiquant les relations entre ses éléments. Ils peuvent être utiles pour décrire un réseau de communication, un réseau routier, un circuit électronique ... mais aussi des relations sociales ou des interactions entre espèces animales.

Les domaines où les graphes interviennent sont donc nombreux : chimie, biologie, sciences sociales, etc., mais c'est avant tout une branche entière des mathématiques et désormais un domaine incontournable de l'informatique.

Les questions qu'on se posera en mathématiques concernent des propriétés globales des graphes (graphes eulériens, hamiltoniens, coloration ...). Les informaticiens cherchent plutôt à concevoir des algorithmes efficaces pour résoudre des problèmes faisant intervenir des graphes (recherche de plus court chemin, problème du voyageur de commerce...)

Dans ce chapitre, on cherchera à

- comprendre la structure de graphe (orienté ou non)
- connaître les deux principales façons d'implémenter les graphes
- décrire les parcours de graphes
- utiliser les graphes pour rechercher des plus courts chemins dans des graphes pondérés

1 Définitions et exemples

1.1 Graphe non orienté

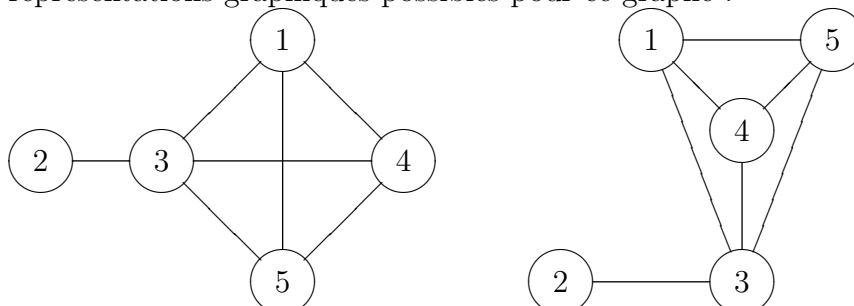
Définition 1.

- Un **graphe non orienté** est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires distinctes d'éléments de S c'est-à-dire de parties de S de cardinal 2.
- Les éléments de S sont appelés **sommets** de G (en anglais *vertices*) et les éléments de A **arêtes** de G (en anglais *edges*).
- L'ordre du graphe G est le cardinal de S (noté $|S|$) c'est-à-dire le nombre de sommets du graphe.

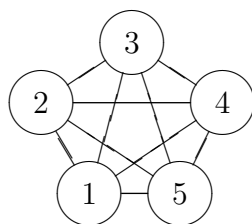
Exemple : On considère le graphe $G = (S, A)$ avec

$S = \{1, 2, 3, 4, 5\}$ et $A = \{\{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$

Voici deux représentations graphiques possibles pour ce graphe :



Exemple : Le graphe complet à n sommets (ou n -clique) est le graphe d'ordre n , noté K_n , tel que deux sommets quelconques de S sont toujours reliés par une arête.

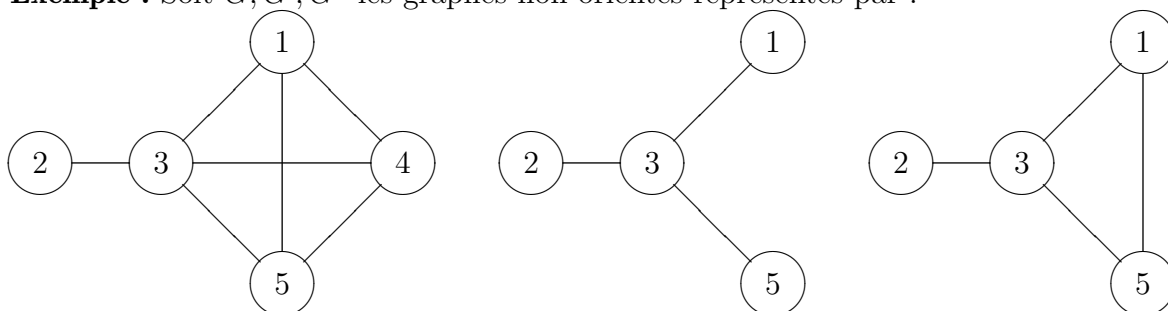


Voici une représentation de K_5 :

Définition 2. Soit $G = (S, A)$ un graphe non orienté.

- On appelle **sous-graphe** de G tout graphe non orienté $G' = (S', A')$ tel que $S' \subset S$ et $A' \subset A \cap \mathcal{P}(S')$.
- Si S' est une partie de S , on appelle **graphe induit** par G sur S' le sous-graphe $G' = (S', A \cap \mathcal{P}(S'))$.

Exemple : Soit G, G', G'' les graphes non orientés représentés par :



G' est un sous-graphe de G mais n'est pas le sous-graphe de G induit sur $S' = \{1, 2, 3, 5\}$ qui est G .

Définition 3. Soit $G = (S, A)$ un graphe non orienté.

- Deux sommets $x, y \in S$ sont dits **adjacents (ou voisins)** si $\{x, y\} \in A$.
- Une arête $a \in A$ est dite **incidente à un sommet** $x \in S$ si x est une extrémité de a , c'est-à-dire si $x \in a$. Elle est dite **incidente à une autre arête** b de A si a et b ont un unique sommet en commun, c'est-à-dire si $|a \cap b| = 1$.
- Le degré d'un sommet x , noté $d(x)$, est le nombre de sommets adjacents à x . C'est également le nombre d'arêtes incidentes à x .

Proposition : Dans un graphe $G = (S, A)$ non orienté, la somme des degrés des sommets est égal à deux fois le nombre d'arêtes soit :

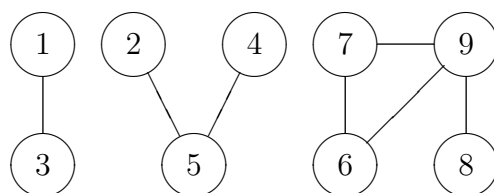
$$2|A| = \sum_{x \in S} d(x)$$

Démonstration : Quand on somme les degrés des sommets, chaque arête est prise en compte deux fois \sharp

Définition 4. Soit $G = (S, A)$ un graphe non orienté.

- Un **chemin (ou chaîne) de longueur** k reliant les sommets x et y est une suite finie $x_0 = x, x_1, \dots, x_k = y$ de sommets telle que pour tout $i \in \llbracket 0, k-1 \rrbracket$, $\{x_i, x_{i+1}\}$ soit une arête de G .
- La **distance entre deux sommets** x et y est la plus petite longueur d'un chemin reliant x à y .
- Le graphe G est dit **connexe** si pour toute paire $\{x, y\}$ de sommets de S , il existe un chemin reliant x et y .
- Une **composante connexe** de G est un sous-graphe connexe maximal de G .

Exemple : Le graphe ci-dessous possède trois composantes connexes



Citons quelques résultats.

Proposition : Tout graphe non orienté connexe d'ordre n possède au moins $n - 1$ arêtes.

Démonstration : Par récurrence sur n .

- La propriété est évidemment vraie pour $n = 1$.
- Soit $n > 1$ tel que la propriété soit vraie à l'ordre $n - 1$ et soit $G = (S, A)$ un graphe connexe d'ordre n . Distinguons deux cas :
 - ou bien G possède un sommet x de degré 1. Alors le sous-graphe G' de G obtenu en supprimant ce sommet et l'unique arête du graphe incidente à x est encore connexe donc comporte au moins $n - 2$ arêtes, donc G comporte au moins $1 + (n - 2) = n - 1$ arêtes.
 - ou bien tous les sommets de G sont de degré supérieur ou égal à 2. Donc $2|A| = \sum_{x \in S} d(x) \geq 2n$ et G possède au moins n arêtes \sharp

Définition 5. Soit $G = (S, A)$ un graphe non orienté.

- On appelle **cycle** de G tout chemin $x_0, x_1, \dots, x_{k-1}, x_0$ de longueur $k > 2$, reliant un sommet x_0 à lui-même, tel que les sommets x_0, x_1, \dots, x_{k-1} soient deux à deux distincts.

Proposition : Si dans un graphe non orienté G , tout sommet est de degré supérieur ou égal à 2, alors G possède au moins un cycle.

Démonstration : Partons d'un sommet arbitraire x_1 , et construisons une suite finie de sommets x_1, x_2, \dots, x_k de la façon suivante :

- $x_i \notin \{x_1, \dots, x_{i-1}\}$
- x_i voisin de x_{i-1}

Comme G possède un nombre fini de sommets, cette construction s'arrête au bout d'un nombre fini d'étapes et, par conséquent, il existe $j \in \llbracket 1, k - 2 \rrbracket$ tel que x_j pour est un autre voisin de x_k . Dans ces conditions, $x_j, x_{j+1}, \dots, x_k, x_j$ est un cycle de G \sharp

Proposition : Un graphe non orienté acyclique d'ordre n comporte au plus $n - 1$ arêtes.

Démonstration : Par récurrence sur n .

- La propriété est vraie pour $n = 1$ car un graphe d'ordre 1 n'a pas d'arêtes.
- Soit $n > 1$ tel que la propriété soit vraie à l'ordre $n - 1$ et soit G un graphe acyclique d'ordre n . D'après la proposition précédente, G possède au moins un sommet x de degré 0 ou 1. Supprimons ce sommet ainsi, le cas échéant que l'éventuelle arête qui lui est incidente. Le graphe obtenu est d'ordre $n - 1$, est encore acyclique donc possède au plus $n - 2$ arêtes. Ainsi, G possède au plus $n - 1$ arêtes \sharp

Définition 6. On appelle **arbre** tout graphe connexe acyclique.

Remarque : D'après les résultats précédents, un arbre d'ordre n possède exactement $n - 1$ arêtes. Plus précisément, pour un graphe d'ordre n , on peut montrer l'équivalence entre les trois propriétés suivantes :

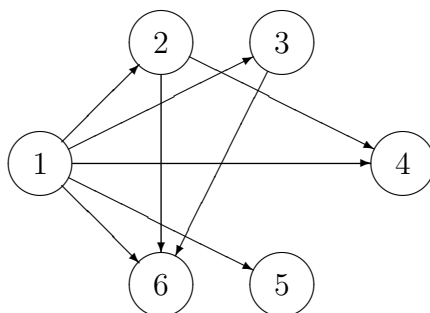
- G est un arbre
- G est un graphe connexe à $n - 1$ arêtes
- G est un graphe acyclique à $n - 1$ arêtes

1.2 Graphe orienté

Définition 7.

- Un **graphe orienté** est un couple $G = (S, A)$ où S est un ensemble fini et A un ensemble fini de couples distincts d'éléments distincts de S (donc tel que $A \subset \{(x, y) \in S \times S / x \neq y\}$).
- Les éléments de S sont appelés **sommets** et ceux de A sont appelés **arcs** (ou arêtes s'il n'y a pas d'ambiguïté sur le fait que G est orienté).
- Si $a = (x, y) \in A$, on dit que a est l'arc d'**origine** x et de **destination** y .

Exemple : On peut considérer sur $S = \llbracket 1, n \rrbracket$ le graphe associé à la relation divise : si $x \neq y$, (x, y) est un arc du graphe si x divise y . Pour $n = 6$, il peut être représenté par :



Définition 8. Soit $G = (S, A)$ un graphe orienté et $x \in S$.

- Un sommet y de S est **voisin de** x (ou **adjacent à** x) si $(x, y) \in A$.
- On appelle **degré sortant** de x le nombre de voisins de x et **degré entrant** de x , le nombre de sommets dont il est voisin

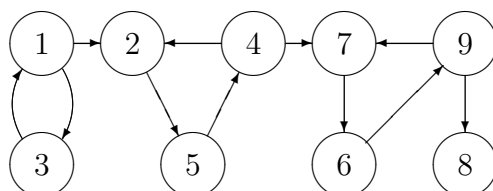
Exemple : Dans le graphe précédent pour $n = 6$, on a le tableau des degrés entrants et sortants :

x	1	2	3	4	5	6
$d_s(x)$	5	2	1	0	0	0
$d_e(x)$	0	1	1	2	1	3

Définition 9. Soit $G = (S, A)$ un graphe orienté.

- Un **chemin de longueur** k reliant les sommets x et y est une suite finie $x_0 = x, x_1, \dots, x_k = y$ de sommets telle que pour tout $i \in \llbracket 0, k-1 \rrbracket$, (x_i, x_{i+1}) soit une arête de G .
- Un **circuit** (ou cycle) de G est un chemin de longueur $k > 1$ de la forme $x_0, x_1, \dots, x_{k-1}, x_0$ reliant un sommet à lui-même et tel que x_0, x_1, \dots, x_{k-1} soient deux à deux distincts.
- La **distance entre deux sommets** x et y est la plus petite longueur d'un chemin reliant x à y .
- Le graphe G est dit **fortement connexe** si pour toute paire $\{x, y\}$ de sommets de S , il existe un chemin reliant x et y et un chemin reliant y à x .
- Une **composante fortement connexe** de G est un sous-graphe de G fortement connexe maximal.

Exemple : dénombrer et représenter les composantes fortement connexes du graphe ci-dessous :



Arbre enraciné

Revenons à la définition générale des arbres que nous avons donnée dans le paragraphe précédent, c'est-à-dire un graphe non orienté, connexe et acyclique et montrons qu'on peut l'enraciner et l'orienter pour obtenir un arbre au sens usuel.

Proposition Si x et y sont deux sommets distincts d'un arbre G , il existe un unique chemin dans G reliant x et y .

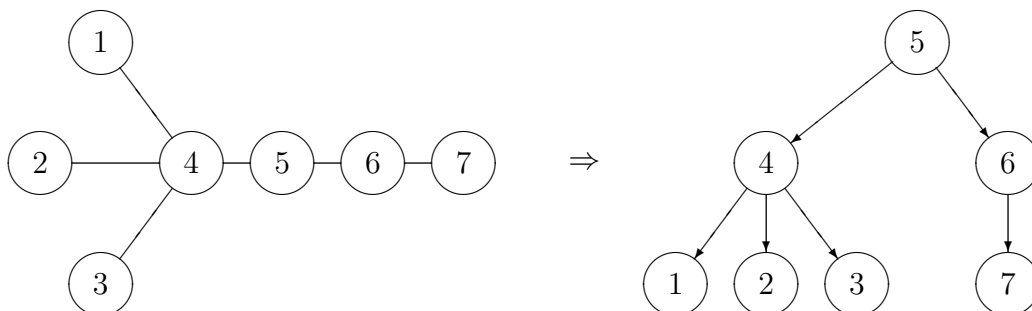
Démonstration : G est connexe d'où l'existence d'un tel chemin. S'il en existait un deuxième, on pourrait former un cycle dans G en empruntant le premier chemin entre x et y , puis le second entre y et x . Le caractère acyclique de G serait alors contredit. $\#$

Une conséquence de ce résultat est qu'il est possible de choisir arbitrairement un sommet r d'un arbre G puis d'orienter les arêtes de ce graphe de sorte qu'il existe un chemin reliant r à tous les autres sommets. On obtient alors un arbre *enraciné* correspondant au sens qu'on lui donne usuellement en informatique.

Démonstration : Raisonnons par récurrence sur n .

- Si $n = 1$, il y a un seul sommet et aucune arête à orienter.
- Soit $n > 1$ tel que la propriété soit vraie à l'ordre $n - 1$. Soit G un arbre d'ordre n et r l'un de ses sommets. On sait que la somme des degrés de G vaut $2n - 2$ donc il y a au moins deux sommets de degré 1 donc au moins un sommet x différent de r et de degré 1. Si on supprime x de G ainsi que l'arête qui le relie à l'arbre, on obtient un arbre d'ordre $n - 1$ auquel on peut appliquer l'hypothèse de récurrence pour l'enraciner en r . Il ne reste plus qu'à orienter l'arête supprimée en direction de x pour enraciner G en r .

Exemple : Ci-dessous on montre un enracinement de l'arbre à gauche en $r = 5$



2 Implémentations

Il existe principalement deux méthodes pour représenter un graphe en machine :

- à l'aide de *listes d'adjacence* (à chaque sommet on associe la liste de ses voisins) ;
- à l'aide d'une *matrice d'adjacence* (la valeur du coefficient d'indices (i, j) indiquera l'existence ou non d'une liaison entre les sommets i et j).

2.1 Par listes d'adjacence

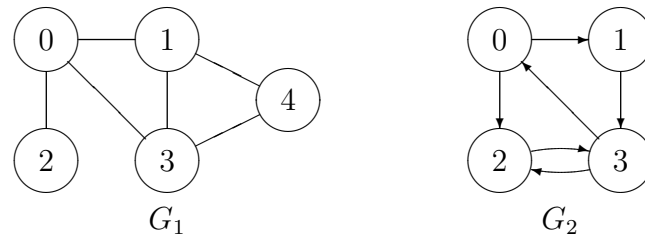
• **Première version** Dans le cas où on travaille sur des graphes dont le nombre de sommets n'a pas vocation à être modifié, on peut utiliser le type suivant pour représenter des graphes :

```
type graphe = int list array
```

Si g est un graphe de ce type, ses sommets sont les entiers de 0 à `Array.length g - 1` et $g.(i)$ est la liste des voisins du sommet i .

On peut, si on le souhaite, et c'est ce que nous ferons ici, s'arranger pour que les listes de voisins soient toujours triées en ordre croissant.

Exemple : Voici deux graphes et leur représentation informatique avec le type précédent



```
let g1 = [| [1;2;3] ; [0;3;4] ; [0] ; [0;1;4] ; [1;3] |];;
let g2 = [| [1;2] ; [3] ; [3] ; [0;2] |];;
```

Écrivons les fonctions donnant la liste des voisins d'un sommet, et celles ajoutant ou supprimant une arête. On donne la version pour un graphe orienté, les lignes à rajouter dans le cas d'un graphe non orienté étant indiquées en commentaires.

```
let voisins g i = g.(i);;

let rec insere x l = match l with
| []          -> [x]
| t::_ when x < t -> x::l
| t::_ when x = t -> l
| t::q         -> t::(insere x q);;

let ajoute_arete g i j =
  g.(i) <- insere j g.(i);
  (* g.(j) <- insere i g.(j) *);;

let rec supprime x l = match l with
| []          -> []
| t::_ when x < t -> l
| t::q when x = t -> q
| t::q         -> t::(supprime x q);;

let supprime_arete g i j =
  supprime j g.(i);
  (* supprime i g.(j) *);;
```

On peut également écrire une fonction convertissant un graphe orienté en un graphe non orienté : par chaque arc d'origine i et de destination j , il faut rajouter i aux voisins de j .

```
let desorienter g =
  let rec aux i li = match li with
  | [] -> ()
  | j::q -> g.(j) <- insere i g.(j); aux i q in
  for i = 0 to Array.length g - 1 do aux i g.(i);;
```

- **Seconde version** L'implémentation précédente présente l'inconvénient de ne pas pouvoir ajouter de sommet aux graphes, la taille d'un tableau n'étant pas modifiable. Pour y remédier, on peut définir un graphe comme une liste de sommets : plus précisément on définit un sommet comme un enregistrement constitué d'un identifiant et de la liste des identifiants de ses voisins.

```
type 'a sommet = {id : 'a ; voisins : 'a list};;
type 'a graph = 'a sommet list;;
```

Cette représentation est plus souple, mais présente l'inconvénient de ne pas permettre un accès rapide à un sommet ou à une arête particulière.

Exemple : Avec cette représentation, le graphe G_2 de l'exemple précédent serait défini par :

```
let g2 = [ {id = 0 ; voisins = [1;2]};
           {id = 1 ; voisins = [3]};
           {id = 2 ; voisins = [3]};
           {id = 3 ; voisins = [0;2]} ];;
```

L'ajout et la suppression d'un sommet, pour le type `'a graph`, se réalisent de la manière suivante :

```
let rec ajoute_sommet g a = match g with
| []          -> [ {id = a ; voisins = [] } ]
| s::q when s.id = a -> g
| s::q        -> s::(ajoute_sommet q a);;

let rec supprime_sommet g a =
  let rec suppr l = match l with
  | []          -> []
  | t::q when t = a -> q
  | t::q        -> t::(suppr q)
  in match g with
  | []          -> []
  | s::q when s.id = a -> supprime_sommet q a
  | s::q        -> let ns = {id = s.id ; voisins = suppr s.voisins}
                    in ns::(supprime_sommet q a);;
```

2.2 Par matrice d'adjacence

Si on veut accéder aux différentes arêtes d'un graphe $G = (S, A)$ en temps constant, il faut procéder autrement : les sommets étant numérotés de 0 à $n - 1$, on introduit la matrice carrée M de taille n dont le terme général $m_{i,j}$ vaut 1 si $(i, j) \in A$ et 0 sinon. On notera que pour un graphe non orienté, la matrice M sera symétrique. La matrice M est appelée matrice d'adjacence du graphe G .

```
type graphe = int array array;;
```

Cette représentation a des avantages : l'ajout ou la suppression d'une arête a un coût constant mais elle a l'inconvénient d'occuper beaucoup plus d'espace mémoire, en

particulier lorsque le nombre d'arêtes est réduit par rapport au nombre de sommets : si $n = |S|$ et $p = |A|$, le coût spatial de la représentation par matrice d'adjacence est un $\Theta(n^2)$ contre un $\Theta(n + p)$ pour une représentation par listes d'adjacence.

Exemple : Le graphe G_2 donné en exemple en 2.1. est représenté par :

```
let g2= [| [|0;1;1;0|];
           [|0;0;0;1|];
           [|0;0;0;1|];
           [|1;0;1;0|] |];;
```

Les fonctions vues précédemment deviennent (on donne la version pour les graphes orientés) :

```
let voisins g i =
  let vi = ref [] in
  for j = Array.length g - 1 downto 0 do
    if g.(i).(j)=1 then vi := j :: !vi
  done;
  !vi;;
```

```
let ajoute_arete g i j = g.(i).(j) <- 1;;
```

```
let supprime_arete g i j = g.(i).(j) <- 0;;
```

Désorienter un graphe orienté revient à rendre sa matrice d'adjacence symétrique ce qu'on peut coder par exemple ainsi :

```
let desorienter g =
  let n = Array.length g in
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      if g.(j).(i) = 1 then g.(i).(j) <- 1
      else if g.(i).(j) = 1 then g.(j).(i) <-1
    done;
  done;;
```

2.3 Passage d'une représentation à l'autre

Il peut être utile de passer de l'un des modes de représentation à l'autre. Le fonction donnant le tableau des listes d'adjacence à partir de la matrice d'adjacence est facile à écrire :

```
let listes_de_mat m =
  let n = Array.length m in
  let g = Array.make n [] in
  for i = 0 to n-1 do
    for j = n-1 downto 0 do
      if m.(i).(j) = 1 then g.(i) <- j::g.(i)
    done;
  done;
  g;;
```


Pour obtenir la matrice d'adjacence d'un graphe donné par listes d'adjacence, nous écrivons tout d'abord une fonction `faire_pour_liste` qui étant donné une liste $l=[l_1; \dots; l_p]$ et une fonction f exécute la succession d'instructions $f(l_1); \dots; f(l_p)$;

```

let rec faire_pour_liste l f = match l with
| []   -> ()
| t::q -> f t; faire_pour_liste q f;;

let mat_de_listes g =
  let n = Array.length g in
  let m = Array.make_matrix n n 0 in
  for i = 0 to n-1 do
    faire_pour_liste g.(i) (fun j -> m.(i).(j) <- 1)
  done;
  m;;

```

Exemple : En prenant toujours l'exemple du graphe G_1 on obtient

```

# let g1 = [| [1;2;3] ; [0;3;4] ; [0] ; [0;1;4] ; [1;3] |];;
val g1 : int list array = [| [1; 2; 3]; [0; 3; 4]; [0]; [0; 1; 4]; [1; 3] |]
# mat_de_listes g1;;
- : int array array =
| [| [0; 1; 1; 1; 0] |]; [| [1; 0; 0; 1; 1] |]; [| [1; 0; 0; 0; 0] |]; [| [1; 1; 0; 0; 1] |];
| [| [0; 1; 0; 1; 0] |] |]

```

3 Parcours d'un graphe

Parcourir un graphe, c'est énumérer l'ensemble des sommets accessibles par un chemin à partir d'un sommet donné, pour leur faire subir un certain traitement. Différentes solutions sont possibles, mais en général, celles-ci tiennent à jour deux listes : la liste des sommets visités (« déjàVus ») et la liste des sommets en cours de traitement (« àTraiter ») et ces méthodes vont différer par la façon dont sont insérés puis retirés les sommets dans cette structure de données.

```

procedure PARCOURS(sommet  $s_0$ )
  àTraiter  $\leftarrow s_0$ 
  déjàVus  $\leftarrow s_0$ 
  while àTraiter  $\neq \emptyset$  do
    àTraiter  $\rightarrow s$ 
    traiter( $s$ )
    for  $t \in \text{voisins}(s)$  do
      if  $t \notin \text{déjàVus}$  then
        Rajouter  $t$  à àTraiter et à déjàVus

```

Arborescence associée à un parcours

À chaque parcours débutant en s_0 peut être associé un arbre enraciné en s_0 : on débute avec le graphe $(\{s_0\}, \emptyset)$ puis à chaque insertion d'un nouveau sommet t dans la liste « ÀTraiter » on ajoute le sommet t et l'arête (s, t) . On construit ainsi un graphe connexe ayant k sommets et $k - 1$ arêtes c'est-à-dire un arbre.

Coût du parcours

Lors d'un parcours, chaque sommet entre au plus une fois dans la liste des sommets à traiter, et n'en sort donc qu'au plus une fois. Si ces opérations d'entrée et de sortie dans la liste ont un coût constant (ce sera le cas dans la suite), le coût total des manipulations de la liste « àTraiter » est un $O(n)$ où $n = |S|$.

Chaque liste d'adjacence est parcourue au plus une fois donc le temps passé à manipuler les listes d'adjacence est un $O(p)$ où $p = |A|$, à la condition toutefois de déterminer si un sommet a déjà été visité avec un coût constant.

Dans ces conditions, le coût total est un $O(n + p)$.

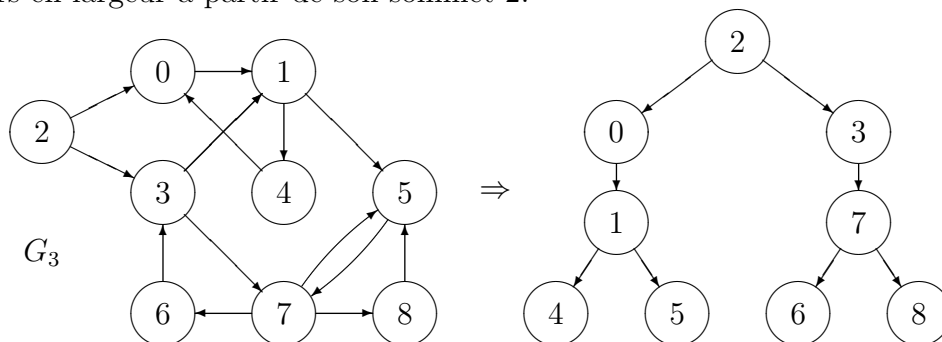
Pour assurer la condition précédente, on utilisera un tableau de booléens pour représenter « déjàVus » ce qui permettra de marquer chaque sommet au moment où il entre dans la liste « àTraiter ».

Nous allons nous intéresser à deux types de parcours, qui diffèrent seulement par la façon d'extraire les sommets de la liste en cours de traitement : les parcours en largeur et en profondeur. Pour le codage, les graphes seront définis par listes d'adjacence et on supposera donnée une fonction `traitement` à appliquer aux sommets parcourus.

3.1 Parcours en largeur

L'algorithme de parcours en largeur (appelé BFS, pour Breadth First Search), consiste à utiliser une file d'attente¹ pour stocker les sommets à traiter : tous les voisins sont traités avant de parcourir le reste du graphe. On traite ainsi successivement tous les sommets à une distance égale à 1 du sommet initial, puis tous ceux à une distance égale à 2, etc. Ce type de parcours est donc idéal pour trouver la plus courte distance entre deux sommets du graphe.

Exemple : On a représenté ci-dessous le graphe G_3 et l'arborescence associée à son parcours en largeur à partir de son sommet 2.



Consignons dans un tableau les états successifs de « déjàVus » et de « àTraiter » au cours de l'exécution de l'algorithme

déjàVus	àTraiter
	2
2	0,3
2,0	3,1
2,0,3	1,7
2,0,3,1	7,4,5
2,0,3,1,7	4,5,6,8
2,0,3,1,7,4	5,6,8
2,0,3,1,7,4,5	6,8
2,0,3,1,7,4,5,6	8
2,0,3,1,7,4,5,6,8	

1. FIFO pour First In, First Out cf. cours de première année

Codage

Voici une fonction de parcours en largeur avec l'implémentation par listes d'adjacence. Le tableau de booléens `t` permet de savoir en temps constant si un sommet a déjà été visité.

```
let bfs g traitement s0 =
  let t = Array.make (Array.length g) false in
  t.(s0) <- true;
  let rec traiter li = match li with
    | [] -> ()
    | x::q -> traitement x; traiter (q @ voisins g.(x))
  and voisins li = match li with
    | [] -> []
    | y::q when t.(y) -> voisins q
    | y::q -> t.(y) <- true; y :: voisins q
  in traiter [s0];;
```

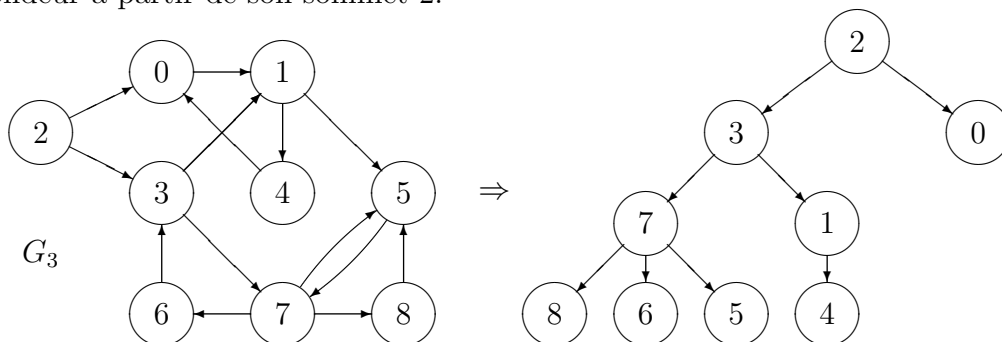
Exemple : Sur l'exemple précédent on obtient :

```
# let g3 = [| [1]; [4;5]; [0;3]; [1;7]; [0]; [7]; [3]; [5;6;8]; [5] |];;
val g : int list array =
  [| [1]; [4; 5]; [0; 3]; [1; 7]; [0]; [7]; [3]; [5; 6; 8]; [5] |]
# bfs g (print_int) 2;;
203174568- : unit = ()
```

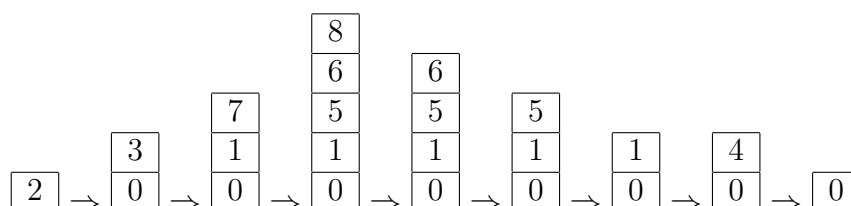
3.2 Parcours en profondeur

L'algorithme de parcours en profondeur (appelé DFS pour Depth First Search) consiste à utiliser une pile² pour stocker les éléments à traiter. On explore ainsi chaque chemin jusqu'au bout avant de passer au chemin suivant.

Exemple : Appliquons ce parcours en profondeur au même exemple que pour le parcours en largeur. On représente à droite l'arborescence associée à son parcours en profondeur à partir de son sommet 2.



Voici les états successifs de la pile « àTraiter » :



2. LIFO pour last in, first out

Le parcours en profondeur de G_3 à partir de 2 est donc 2, 3, 7, 8, 6, 5, 1, 4, 0. On constate qu'il correspond au parcours préfixe dans l'arborescence associée.

Codage :

Pour coder l'algorithme de parcours en profondeur, nous utiliserons le module `Stack` d'OCaml et en particulier ses fonctions `create` (`create()` crée une pile vide), `push` (`push x p` ajoute l'élément `x` au sommet de la pile `p`), `pop` (`pop p` enlève l'élément qui est au sommet de la pile et le renvoie ou retourne `Stack.Empty` si la pile est vide) et enfin `is_empty` (`is_empty p` retourne un booléen indiquant si la pile est vide ou non).

```
open Stack;;

let dfs g traitement s =
  let dejavu = Array.make (Array.length g) false
  and atraiter = create() in
  push s atraiter ; dejavu.(s) <- true ;
  let rec ajoute_voisin l = match l with
    | [] -> ()
    | t::q when dejavu.(t) -> ajoute_voisin q
    | t::q -> push t atraiter ; dejavu.(t) <- true ;
              ajoute_voisin q
  in
  while not (is_empty atraiter) do
    let s = pop atraiter in
      traitement s ;
      ajoute_voisin g.(s)
  done;;
```

Exemples : Exécutons la fonction précédente à partir de plusieurs sommets de G_3 :

```
# let g3 = [| [1]; [4;5]; [0;3]; [1;7]; [0]; [7]; [3]; [5;6;8]; [5] |];;
val g3 : int list array =
  [| [1]; [4; 5]; [0; 3]; [1; 7]; [0]; [7]; [3]; [5; 6; 8]; [5] |]
# dfs g3 (print_int) 2;;
237865140- : unit = ()
# dfs g3 (print_int) 3;;
37865140- : unit = ()
# dfs g3 (print_int) 1;;
15786340- : unit = ()
```

3.3 Application : détermination des composantes connexes

Si l'on effectue un parcours (en largeur ou en profondeur) d'un graphe non orienté à partir d'un sommet `s`, les sommets obtenus correspondent à la composante connexe du sommet `s`.

Donc pour tester si un graphe non orienté est connexe, il suffit de lancer un parcours et de vérifier si l'on obtient bien tous les sommets du graphe.

De même, pour obtenir les composantes connexes d'un graphe non orienté, on choisit un sommet et on effectue un parcours à partir de ce sommet, puis on recommence

tant qu'il reste des sommets non obtenus.

Codage

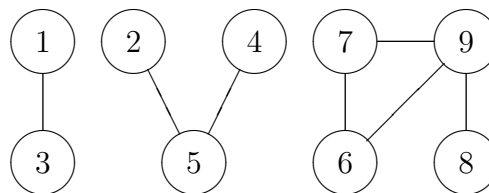
On utilise une variante du parcours en largeur pour obtenir la liste des sommets dans l'ordre où ils sont parcourus.

```

let composantes g =
  let n = Array.length g in
  let t = Array.make n false in
  let rec bfs li = match li with
    | [] -> []
    | x :: q when t.(x) -> bfs q
    | x :: q -> t.(x) <- true; x :: bfs (q @ g.(x))
  in
  let rec aux acc x = match x with
    | _ when x = n -> acc
    | _ when t.(x) -> aux acc (x+1)
    | _ -> aux (bfs [x] :: acc) (x+1) in
  aux [] 0;;

```

Exemple : Appliquons la fonction `composantes` au graphe suivant :



```

# let g = [| [6;7;8]; [3]; [5]; [1]; [5]; [2;4]; [0;7]; [0;6]; [0] |];;
val g : int list array =
  [| [6; 7; 8]; [3]; [5]; [1]; [5]; [2; 4]; [0; 7]; [0; 6]; [0] |]
# composantes g;;
- : int list list = [[2; 5; 4]; [1; 3]; [0; 6; 7; 8]]

```

4 Plus court chemin

4.1 Principes généraux

Pour déterminer la distance entre deux sommets d'un graphe non pondéré, il suffit d'effectuer un parcours en largeur à partir de l'un des sommets jusqu'à trouver l'autre.

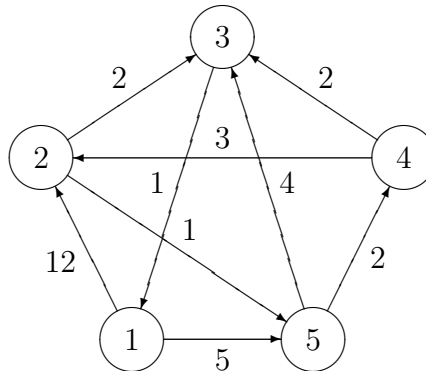
Cependant, connaître le nombre minimal d'arêtes à parcourir entre deux sommets n'est pas toujours suffisant : de nombreux problèmes ajoutent une pondération à chaque arête.

Définition 10.

- Un graphe pondéré (orienté ou non) est un triplet $G = (S, A, f)$ tel que (S, A) est un graphe et f une fonction de A dans \mathbb{R} . Si $a \in A$, $f(a)$ est appelé poids de l'arête a .
- Soit $N \in \mathbb{N}$ et $c = x_0, x_1, \dots, x_N$ un chemin dans le graphe (S, A) . Alors le poids du chemin c est $\sum_{i=0}^{N-1} f(x_i, x_{i+1})$

- La distance entre deux sommets x et y de G est le poids minimal d'un chemin de x à y . Un plus court chemin de x à y dans G est un chemin de x à y de poids égal à la distance entre x et y .

Exemple : Dans le graphe suivant :



la distance entre les sommets 1 et 2 est égale à 10 et 1,5,4,2 est un plus court chemin entre ces deux sommets.

Remarque : Pour pouvoir assurer l'existence d'un plus court chemin, il faut qu'il n'y ait pas de cycle de poids strictement négatif (sinon, on peut obtenir un chemin de poids arbitrairement petit en bouclant sur ce cycle). Dans la suite, on supposera que le graphe considéré est connexe et ne possède pas de cycle de poids négatif.

Problèmes de plus courts chemins

Il existe trois problèmes de plus courts chemins :

- (i) calculer le chemin de poids minimal entre une origine a et une destination b ;
- (ii) calculer les chemins de poids minimal entre une origine a et tous les autres sommets du graphe ;
- (iii) calculer tous les chemins de poids minimal entre deux sommets quelconques du graphe.

Le troisième problème est le plus simple à résoudre : l'algorithme de FLOYD-WARSHALL nous en donnera une solution. En revanche, et de manière surprenante, il n'existe pas à l'heure actuelle d'algorithme qui donne la solution du premier problème sans résoudre le second. Nous verrons une façon de résoudre le second problème à l'aide de l'algorithme de DIJKSTRA.

Principe de sous-optimalité

Les algorithmes que nous allons étudier utilisent tous le résultat suivant :

Proposition Si $a \rightsquigarrow b$ est un plus court chemin qui passe par c , alors ses sous-chemins $a \rightsquigarrow c$ et $c \rightsquigarrow b$ sont eux aussi des plus courts chemins.

Démonstration : En effet, s'il existait un chemin plus court entre, par exemple a et c , il suffirait de l'emprunter lors du trajet entre a et b pour contredire le caractère minimal du chemin $a \rightsquigarrow b$ \sharp

4.2 Algorithme de FLOYD-WARSHALL

Pour prendre en compte le fait qu'un graphe soit pondéré, on peut modifier la matrice d'adjacence M de ce graphe (S, A, f) en convenant que désormais $m_{i,j} = f(s_i, s_j)$ si $i \neq j$ et qu'il existe un arc d'origine v_i et de destination v_j et $m_{i,j} = \infty$ sinon. En outre, on pose $m_{i,i} = 0$.

Exemple : Voici la nouvelle matrice d'adjacence du graphe pondéré précédent

$$M = \begin{pmatrix} 0 & 12 & \infty & \infty & 5 \\ \infty & 0 & 2 & \infty & 1 \\ 1 & \infty & 0 & \infty & \infty \\ \infty & 3 & 2 & 0 & \infty \\ \infty & \infty & 4 & 2 & 0 \end{pmatrix}$$

Principe de l'algorithme

Si n désigne l'ordre du graphe pondéré G , et M sa matrice d'adjacence, l'algorithme de FLOYD-WARSHALL consiste à calculer la suite finie de matrices $M^{(k)}$ pour $0 \leq k \leq n$ avec $M^{(0)} = M$ et :

$$\forall k \in \llbracket 0, n-1 \rrbracket, \forall (i, j), m_{i,j}^{(k+1)} = \min(m_{i,j}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)})$$

Proposition Avec les notations précédentes, $\forall k \in \llbracket 0, n \rrbracket$ et $\forall (i, j)$, $m_{i,j}^{(k)}$ est égal au poids du chemin minimal reliant s_i à s_j et ne passant que par les sommets de $\{s_1, s_2, \dots, s_k\}$.

En particulier, $\forall (i, j)$, $m_{i,j}^{(n)}$ est le poids minimal d'un chemin reliant s_i à s_j .

Démonstration : On raisonne par récurrence finie sur k .

- La propriété est vraie pour $k = 0$ car $m_{i,j}^{(0)} = m_{i,j}$ est bien le poids d'un chemin minimal reliant s_i à s_j sans passer par aucun autre sommet.
- Soit $k \in \llbracket 0, n-1 \rrbracket$ tel que la propriété soit vraie à l'ordre k . Considérons un chemin $s_i \rightsquigarrow s_j$ ne passant que par les sommets s_1, \dots, s_{k+1} et de poids minimal.

Si ce chemin ne passe pas par s_{k+1} , son poids est par hypothèse de récurrence égal à $m_{i,j}^{(k)}$.

Sinon, il passe une unique fois par s_{k+1} et, par principe de sous-optimalité, ses sous-chemins $s_i \rightsquigarrow s_{k+1}$ et $s_{k+1} \rightsquigarrow s_j$ sont de poids minimal parmi les chemins ne passant que par s_1, \dots, s_k donc par hypothèse de récurrence, $s_i \rightsquigarrow s_j$ est de poids total $m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}$.

On peut donc en conclure que $\min(m_{i,j}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}) = m_{i,j}^{(k+1)}$ est le poids minimal d'un plus court chemin reliant s_i à s_j en ne passant que par les sommets s_1, \dots, s_{k+1} \sharp

Exemple : Sur notre exemple la mise en œuvre de l'algorithme de Floyd-Warshall donne par exemple les matrices

$$M^{(1)} = \begin{pmatrix} 0 & 12 & \infty & \infty & 5 \\ \infty & 0 & 2 & \infty & 1 \\ 1 & \mathbf{13} & 0 & \infty & \mathbf{6} \\ \infty & 3 & 2 & 0 & \infty \\ \infty & \infty & 4 & 2 & 0 \end{pmatrix} \quad \text{et} \quad M^{(2)} = \begin{pmatrix} 0 & 12 & \mathbf{14} & \infty & 5 \\ \infty & 0 & 2 & \infty & 1 \\ 1 & 13 & 0 & \infty & 6 \\ \infty & 3 & 2 & 0 & \mathbf{4} \\ \infty & \infty & 4 & 2 & 0 \end{pmatrix}$$

Implémentation

Pour gérer la valeur ∞ , on peut définir le type somme suivant :

```
type poids = Inf | P of int;;
```

ce qui nous permet de représenter un graphe pondéré par le type `poids array array`.

On a besoin de définir quelques fonctions de deux objets de type `poids`.

```

let som p1 p2 = match p1, p2 with
  | Inf , _    -> Inf
  | _ , Inf    -> Inf
  | P a , P b -> P (a + b);;

let mini p1 p2 = match p1, p2 with
  | Inf , x    -> x
  | x , Inf    -> x
  | P a , P b -> P (min a b);;

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

```

Remarquons qu'il est possible de calculer les différentes valeurs de la suite $(M^{(k)})$ en utilisant une seule matrice car la formule

$$m_{i,j}^{(k+1)} = \min(m_{i,j}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)})$$

reste valable si on remplace $m_{i,k+1}^{(k)}$ par $m_{i,k+1}^{(k+1)}$ étant donné qu'ils sont égaux puisqu'on a supposé qu'il n'y avait pas de cycle de poids négatif. De même, on a $m_{k+1,j}^{(k)} = m_{k+1,j}^{(k+1)}$.

```

let floydWarshall g =
  let n = Array.length g in
  let m = Array.make_matrix n n Inf in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- g.(i).(j)
    done;
  done;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- mini m.(i).(j) (som m.(i).(k) m.(k).(j))
      done;
    done;
  done;
  m;;

```

Exemple : Sur notre exemple, la fonction retourne la matrice

```

# floydWarshall g;;
- : poids array array =
[[|P 0; P 10; P 9; P 7; P 5||]; [|P 3; P 0; P 2; P 3; P 1||];
 [|P 1; P 11; P 0; P 8; P 6||]; [|P 3; P 3; P 2; P 0; P 4||];
 [|P 5; P 5; P 4; P 2; P 0||]]

```

Complexité

En raison de la présence des trois boucles `for` la fonction `floydWarshall` a une complexité temporelle en $O(n^3)$ et une complexité spatiale en $O(n^2)$.

Détermination des plus courts chemins

L'algorithme précédent se contente de calculer le poids des plus courts chemins mais ne garde pas la trace de ces chemins. Ces derniers peuvent être stockés dans une matrice annexe, ce qui conduit à la version suivante :

```
let plusCourtsChemins g =
  let n = Array.length g in
  let m = Array.make_matrix n n Inf
  and c = Array.make_matrix n n [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- g.(i).(j)
    done;
  done;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        let l = som m.(i).(k) m.(k).(j) in
        if inferieur l m.(i).(j) then
          (m.(i).(j) <- l ; c.(i).(j) <- c.(i).(k)@c.(k).(j))
        done;
      done;
    done;
  done;
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i <> j && m.(i).(j) <> Inf then c.(i).(j) <- [i+1]@c.(i).(j)@[j+1]
    done;
  done;
  c;;
```

Exemple : Sur notre exemple, on obtient la matrice suivante :

```
# plusCourtsChemins g;;
- : int list array array =
[[[[]; [1; 5; 4; 2]; [1; 5; 3]; [1; 5; 4]; [1; 5]]];
 [[2; 3; 1]; []; [2; 3]; [2; 5; 4]; [2; 5]]];
 [[3; 1]; [3; 1; 5; 4; 2]; []; [3; 1; 5; 4]; [3; 1; 5]]];
 [[4; 3; 1]; [4; 2]; [4; 3]; []; [4; 2; 5]]];
 [[5; 3; 1]; [5; 4; 2]; [5; 3]; [5; 4]; []]]]
```

On remarque que le plus court chemin allant de 3 à 2 passe par les 5 sommets du graphe et a un poids égal à 11 (inférieur au poids de 3,1,2 égal à 13).

Application au calcul de la fermeture transitive d'un graphe

On considère un graphe *non pondéré*, orienté ou non. Le problème de la *fermeture transitive* consiste à déterminer si deux sommets s et t peuvent être reliés par un chemin allant de s à t . Pour le résoudre, on peut utiliser la matrice d'adjacence associée à ce graphe (remplie de booléens indiquant la présence ou non d'une arête entre deux sommets) et on remplace dans l'algorithme de FLOYD-WARSHALL la relation de récurrence sur les coefficients des matrices $M^{(k)}$ par :

$$m_{i,j}^{(k+1)} = m_{i,j}^{(k)} \text{ ou } (m_{i,k+1}^{(k)} \text{ et } m_{k+1,j}^{(k)})$$

Il est facile de prouver que le booléen $m_{i,j}^{(k)}$ indique l'existence ou non d'un chemin reliant les sommets s_i et s_j en ne passant que par les sommets s_1, s_2, \dots, s_k et que par conséquent $M^{(n)}$ résout le problème de la fermeture transitive. L'algorithme ainsi modifié est connu sous le nom d'algorithme de WARSHALL ou de ROY-WARSHALL.

```

let Warshall g =
  let n = Array.length g in
  let m = Array.make_matrix n n false in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- g.(i).(j)
    done;
  done;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- m.(i).(j) || (m.(i).(k) && m.(k).(j))
      done;
    done;
  done;
  m;;

```

4.3 Algorithme de DIJKSTRA

Le problème que l'on cherche ici à résoudre est de déterminer les poids des plus courts chemins depuis un sommet s_0 fixé vers l'ensemble des autres sommets du graphe. L'algorithme que nous allons étudier nécessite de supposer que **les poids des arêtes sont tous positifs**. On notera $\delta(s, s')$ le poids minimum d'un chemin de s à s' appelé encore distance de s à s' .

Le principe est de calculer, à chaque étape, le sommet suivant le plus proche de s_0 : c'est un **algorithme glouton** (*greedy* en anglais) qui progresse en cherchant un optimum local et qui permet à la fin d'obtenir un optimum global.

On commence par la recherche du sommet le plus proche de s_0 : c'est forcément un voisin de s_0 car si s non voisin de s_0 , pour aller de s_0 à s , il faudra passer par un voisin s' de s_0 et alors $\delta(s_0, s) \geq f(s_0, s')$ (f fonction poids).

Le plus proche sommet de s_0 est donc s_1 tel que $f(s_0, s_1) = \min\{f(s_0, s), s \text{ voisin de } s_0\}$

On généralise ce raisonnement pour les sommets suivants.

- On suppose déterminés les $p - 1$ sommets les plus proches de s_0 : s_1, \dots, s_{p-1} . On cherche le suivant : s_p .
- On note $T = \{s_0, s_1, \dots, s_{p-1}\}$ et $S' = S \setminus T$: s_p doit être un sommet de S' tel que $\delta(s_0, s_p) = \min\{\delta(s_0, s'), s' \in S'\}$.
- Un chemin de poids minimum de s_0 à s_p ne passe, en dehors de s_p que par des sommets de T car s'il existait un point de S' dans ce chemin, celui-ci serait à une distance de s_0 inférieure à celle de s_p ce qui est exclu.
- Si l'avant-dernier sommet du chemin de poids minimum de s_0 à s_p est s_k , on peut écrire : $\delta(s_0, s_p) = \delta(s_0, s_k) + f(s_k, s_p)$.
- Ainsi, s_p est un sommet de S' , voisin d'un sommet de T et qui réalise le minimum de $\{\delta(s_0, s) + f(s, s'); s \in T \text{ et } s' \in S', s' \text{ voisin de } s\}$

L'algorithme de DIJKSTRA se déroule donc de la manière suivante :

```

function DIJKSTRA(sommet  $s_0$ )
   $T \leftarrow s_0$ 
  for all  $s \in S$  do
     $d_s \leftarrow f(s_0, s)$ 
  while  $S \setminus T \neq \emptyset$  do
    déterminer  $u \in S \setminus T$  tel que  $d_u = \min\{d_{s'}; s' \in S \setminus T\}$ 
     $T \leftarrow T \cup \{u\}$ 
    for  $s' \in S \setminus T$  do
       $d_{s'} \leftarrow \min(d_{s'}, d_u + f(u, s'))$ 
  return  $d$ 

```

En vertu des explications précédentes on a la proposition :

Proposition

Au cours de l'exécution de l'algorithme de DIJKSTRA :

- ◇ Pour tout $s \in T$, le nombre d_s est le poids d'un chemin minimal de s_0 à s .
- ◇ Pour tout $s \in S \setminus T$, le nombre d_s est le poids d'un chemin minimal de s_0 à s ne passant que par des sommets de T .

Exemple : Reprenons l'exemple qui nous a déjà servi à illustrer l'algorithme de FLOYD-WARSHALL et observons l'évolution du tableau d quand on choisi le sommet 4 comme origine :

T	1	2	3	4	5
$\{4\}$	∞	3	2	<u>0</u>	∞
$\{4, 3\}$	3	3	<u>2</u>	<u>0</u>	∞
$\{4, 3, 1\}$	<u>3</u>	3	<u>2</u>	<u>0</u>	8
$\{4, 3, 1, 2\}$	<u>3</u>	<u>3</u>	<u>2</u>	<u>0</u>	4
$\{4, 3, 1, 2, 5\}$	<u>3</u>	<u>3</u>	<u>2</u>	<u>0</u>	<u>4</u>

On observe que le résultat obtenu est l'avant dernière ligne de la matrice $A^{(5)}$ obtenue par l'algorithme de FLOYD-WARSHALL.

Étude de la complexité

L'étude de la complexité de l'algorithme de DIJKSTRA est un peu délicate car elle dépend du choix de la représentation des structures de données. En gros, si n désigne le nombre de sommets du graphe, on effectue $n - 1$ transferts de S' vers T en ayant à chaque fois à déterminer le plus petit élément d'une partie du tableau d puis en modifiant certaines des valeurs de ce tableau en conséquence. Tout ceci ayant un coût linéaire, la complexité totale est *a priori* en $O(n^2)$.

D'ailleurs, pour un graphe dense (c'est-à-dire ayant beaucoup d'arêtes, le pire des cas étant le cas d'un graphe complet donc avec $\binom{n}{2}$ arêtes), on a de toute façon un coût au moins quadratique.

En revanche, pour les graphes dits « creux », il est possible d'améliorer cette complexité en utilisant une file de priorité (autrement dit un tas et ici un tas-min) pour représenter $S' = S \setminus T$.

On sait qu'un tas-min permet la récupération de l'élément de priorité minimale (ici de d_s minimum) pour un coût en $O(\log n)$ (correspondant à la reformation du tas) donc le coût total des transferts de S' à T est un $O(n \log n)$. Par ailleurs, chaque mise à jour d'une valeur du tableau d a lui aussi un coût en $O(\log n)$ (là encore pour reformer le tas S') mais chaque arête n'interviendra au plus qu'une fois dans ces modifications de d , donc si p désigne le nombre total d'arêtes du graphe, le coût total de ces modifications est un $O(p \log n)$.

Au final, ceci permet d'avoir une complexité un coût total en $O((n + p) \log n)$ donc en $O(p \log n)$ si le graphe est connexe car alors $p \geq n - 1$.

Pour que l'utilisation de tas n'augmente pas la complexité, il faut donc que $p \log n = O(n^2)$ soit que $p = O(\frac{n^2}{\log n})$.

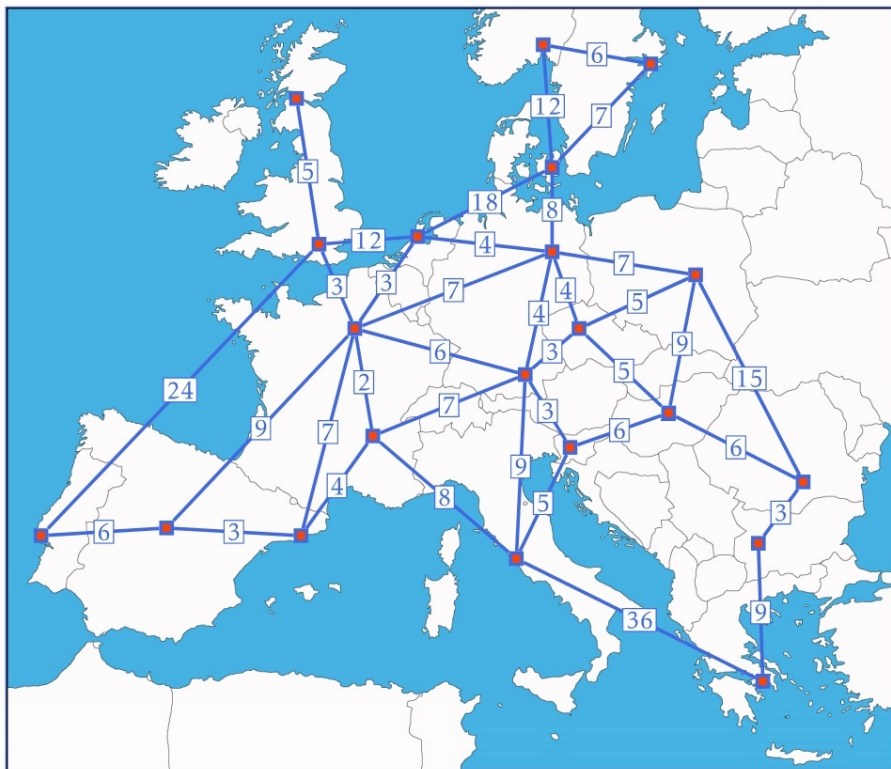
Détermination des chemins de poids minimum

Si l'on souhaite garder trace des chemins de poids minimum entre s_0 et les différents sommets du graphe, il suffit d'utiliser un tableau c que l'on modifie en même temps que d : en convenant de noter les sommets $0, 1, \dots, n - 1$, lorsque d_j est remplacé par $d_i + f(i, j)$, on mémorise i dans c_j . Ainsi, à la fin de l'algorithme, c_j contient le sommet précédant le sommet j dans un chemin minimal allant de s_0 à j , ce qui permet de reconstituer un chemin de poids minimum de s_0 vers j une fois l'algorithme terminé.

5 Arbres couvrant minimal

5.1 Généralités

Pour présenter le problème auquel nous allons nous intéresser dans cette section, considérons l'exemple d'un réseau de transport maritime et ferroviaire représenté sous forme d'un graphe dont les sommets sont les centres de transit, les arêtes représentent les liaisons existantes et leur poids donne la durée du trajet entre les deux extrémités de l'arête.



Imaginons que, pour des raisons d'économie, la société qui gère ce réseau souhaite supprimer le plus de liaisons possible, tout en gardant la possibilité de relier entre eux tous les centres de transit. En outre, elle souhaite que son réseau reste le plus performant possible, autrement dit que la somme des durées des liaisons conservées soit la plus petite possible : c'est le problème de la recherche d'un arbre couvrant de poids minimum (*minimal spanning tree*) dont une solution pour cet exemple est présentée ci-dessous :



Définition 11. Soit $G = (S, A)$ un graphe non orienté connexe muni d'une pondération $f : \rightarrow \mathbb{R}$ à valeurs strictement positives.

Un sous-graphe $G' = (S', A')$ de G est dit **couvrant** s'il est également connexe et que $S' = S$.

Un sous-graphe couvrant G' de G est dit **minimal** si la somme des poids de ses arêtes est minimale.

Proposition Avec les notation précédentes, si la fonction poids est à valeurs dans \mathbb{R}_+ , G possède un sous-graphe couvrant minimal, et tout sous-graphe couvrant minimal de G est un arbre (c'est-à-dire un graphe connexe acyclique)

Démonstration : L'ensemble des sous-graphes couvrants de G est non vide car il contient G et il est fini, donc il existe un sous-graphe couvrant minimal de G .

Soit G' un sous-graphe couvrant minimal de G . Alors, par définition G' est connexe donc s'il n'était pas un arbre, il contiendrait un cycle. En supprimant une arête de ce cycle, le sous-graphe obtenu serait toujours couvrant mais de poids strictement inférieur, ce qui est absurde. G' est donc un arbre \sharp

Nous allons étudier deux algorithmes permettant de construire un arbre couvrant minimal pour un graphe G . Chacun d'entre eux utilise une des caractérisations des arbres : le premier, l'algorithme de PRIM, utilise le fait qu'un graphe à n sommets est un arbre si et seulement si il est connexe à $n - 1$ arêtes ; le second, l'algorithme de KRUSKAL, qu'un graphe à n sommets est un arbre si et seulement si il est acyclique à $n - 1$ arêtes.

5.2 Algorithme de PRIM

Principe

- On choisit un sommet s_0 arbitrairement
- On ajoute les arêtes une à une en choisissant à chaque étape une arête qui sort de l'arbre déjà construit et qui est de poids minimum

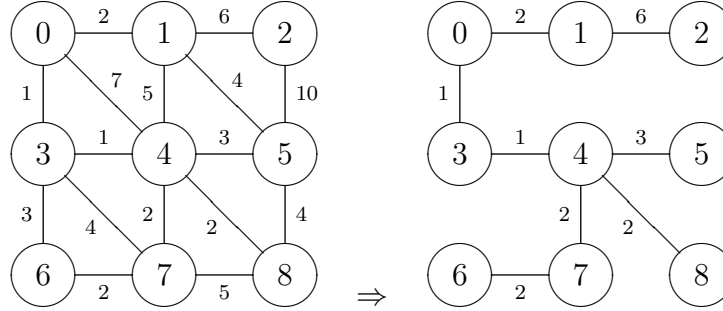
Algorithme

```

function PRIM(graphe connexe :  $G = (S, A)$ )
   $S' \leftarrow \{s_0\}$  (* un sommet choisi arbitrairement *)
   $A' = \emptyset$ 
  while  $S' \neq S$  do
    Soit  $(a, b) \in A$  tel que  $(a, b) \in S' \times (S \setminus S')$  et  $f(a, b)$  est minimal
     $S' \leftarrow S' \cup \{b\}$ 
     $A' \leftarrow A' \cup \{(a, b)\}$ 
  return  $(S', A')$ 

```

Exemple : Pour le graphe ci-dessous, on a représenté l'arbre couvrant minimal obtenu en démarrant avec le sommet 0.



Proposition L'algorithme de PRIM calcule un arbre couvrant minimal.

Démonstration : Il est clair que le graphe construit par l'algorithme précédent est un graphe connexe couvrant. De plus, il possède par construction n sommets et $n - 1$ arêtes donc il s'agit bien d'un arbre. Il reste à montrer qu'il est bien de poids minimal. On montre par récurrence sur $|S'|$ qu'à chaque étape de l'algorithme, il existe un arbre couvrant de poids minimal dont (S', A') est un sous-graphe, ce qui montrera le résultat souhaité.

- Le résultat est vrai lorsque $|S'| = 1$ car alors $|A'| = \emptyset$.
- Soit $k \in \mathbb{N}^*$ tel que le résultat soit vrai quand $|S'| = k$: il existe donc un arbre couvrant minimal T dont (S', A') est un sous-graphe. Notons (a, b) l'arête que l'algorithme ajoute à A' .

Si (a, b) appartient à T , le résultat est acquis. Dans le cas contraire, ajoutons cette arête à T . Cet ajout a pour effet de créer un cycle. Ce cycle parcourt à la fois des éléments de S' (et, parmi eux a) et des éléments de $S \setminus S'$ (et parmi eux b). Il existe donc nécessairement une autre arête $(a', b') \neq (a, b)$ de ce cycle telle que $a' \in S'$ et $b' \in S \setminus S'$. Considérons alors l'arbre T' obtenu en supprimant l'arête (a', b') . Il s'agit à nouveau d'un arbre couvrant et il est de poids minimal car par choix de (a, b) on a $f(a, b) \leq f(a', b')$ $\#$

Étude de la complexité

Si p est le nombre d'arêtes du graphe G , la recherche naïve de l'arête de poids minimal (a, b) est un $O(p)$ et le coût total un $O(np)$. On peut néanmoins faire mieux en procédant à un prétraitement des sommets consistant à déterminer pour chacun d'eux l'arête incidente de poids minimal qui le relie à un sommet de S' . Dès lors, le coût de la recherche de l'arête de poids minimal devient un $O(n)$, et une fois le nouveau sommet ajouté à S' , il suffit de mettre à jour les voisins de ce dernier. Sachant que le coût du pré-traitement est un $O(p) = O(n^2)$, le coût total de l'algorithme est un $O(n^2)$.

En outre, on peut montrer que l'utilisation d'un tas pour stocker les différents sommets n'appartenant pas à S' permet de réduire le coût, qui devient alors un $O(p \log n)$.

5.3 Algorithme de KRUSKAL

L'idée de cet algorithme est de maintenir à chaque étape un graphe partiel acyclique (autrement dit une forêt) jusqu'à obtenir une unique composante connexe (donc un arbre).

Principe

- On part du graphe comportant tous les sommets de $G = (S, A)$ et aucune arête
- Tant que c'est possible, on ajoute une arête de poids minimum permettant de réunir deux composantes connexes distinctes

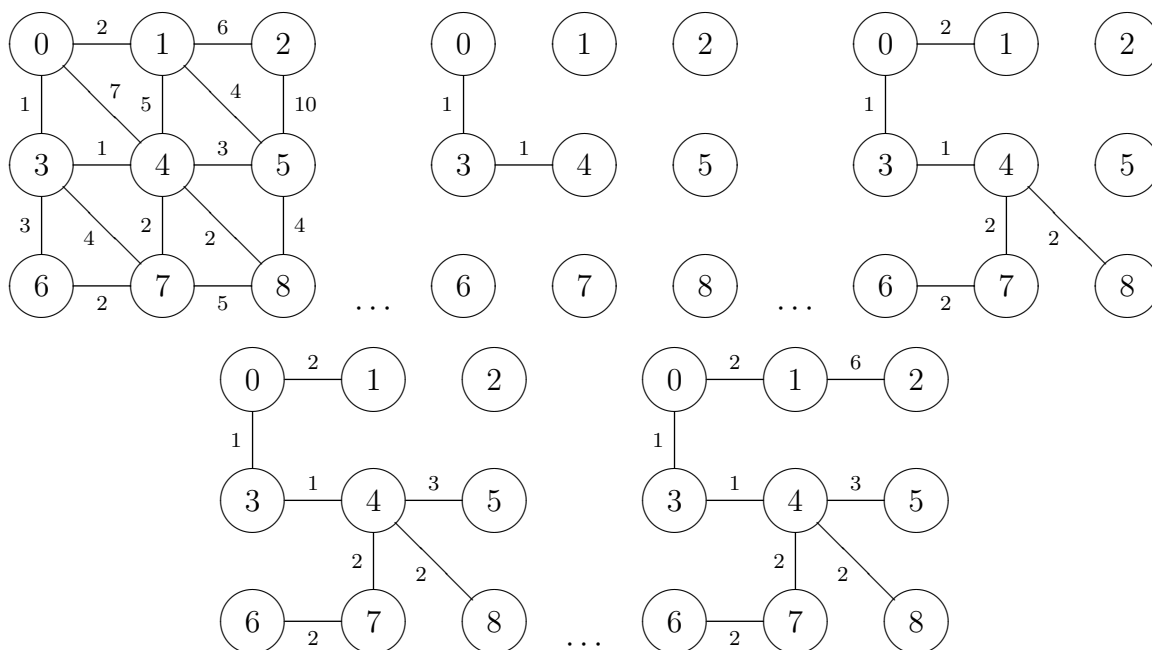
Algorithme

```

function KRUSKAL(graphe :  $G = (S, A)$ )
   $A' = \emptyset$ 
   $A_t = \text{tri\_croissant}(A)$ 
  for  $(a, b) \in A_t$  do
    if  $A' \cup \{(a, b)\}$  est acyclique then
       $A' \leftarrow A' \cup \{(a, b)\}$ 
  return  $(S, A')$ 

```

Exemple : On applique l'algorithme de KRUSKAL au même exemple que celui choisi pour illustrer l'algorithme de PRIM en représentant trois étapes intermédiaires en plus du résultat final.



Remarques :

- Rien n'empêche d'appliquer l'algorithme de KRUSKAL à un graphe G non connexe : dans ce cas, il fournira une forêt couvrante de poids minimal de G , c'est-à-dire une forêt dont chaque composante connexe est un arbre couvrant minimal d'une des composantes connexes de G
- Si on applique l'algorithme de KRUSKAL à un graphe G dont on sait qu'il est connexe, on pourra s'arrêter après avoir ajouté $|S| - 1$ arêtes pour obtenir un arbre couvrant minimal.

Pour prouver la validité de l'algorithme de KRUSKAL nous établissons tout d'abord un résultat préliminaire

Lemme Toutes les forêts couvrantes d'un graphe G ont le même nombre d'arêtes

Démonstration : Notons $G = (S, A)$ et $G_1 = (S_1, A_1), \dots, G_p = (S_p, A_p)$ les composantes connexes de G . Puisqu'un arbre a un nombre d'arêtes égal à son nombre de sommets -1 , le nombre d'arêtes d'une forêt couvrante de G est égale à $\sum_{i=1}^p (|S_i| - 1) = |S| - p$ indépendant de la forêt couvrante considérée. $\#$

Théorème Appliqué à un graphe G l'algorithme de KRUSKAL détermine une forêt couvrante de poids minimal de G .

Démonstration : Le graphe construit par l'algorithme de KRUSKAL est bien une forêt couvrante puisqu'on ajoute des arêtes sans jamais créer de cycle.

Il reste à prouver que cette forêt est de poids minimal. Pour cela, notons $A' = (a'_1, \dots, a'_k)$ les arêtes choisies par l'algorithme de KRUSKAL, rangées par poids croissants, et soit une autre forêt couvrante $F = (S, A'')$ d'arêtes (a''_1, \dots, a''_k) (également rangées par poids croissants). Nous allons montrer que $\forall i \in \llbracket 1, k \rrbracket, f(a'_i) \leq f(a''_i)$ (où f est la fonction poids) ce qui donnera le résultat souhaité.

On raisonne par l'absurde en supposant qu'il existe un entier $i \in \llbracket 1, r \rrbracket$ tel que $f(a'_i) > f(a''_i)$. Considérons le graphe $G' = (S, E)$ avec $E = \{a \in A \mid f(a) \leq f(a''_i)\}$. Appliqué à G' , l'algorithme de KRUSKAL se déroule comme sur G et retourne un ensemble d'arêtes inclus dans $\{a'_1, \dots, a'_{i-1}\}$, autrement dit une forêt couvrante de G' comportant au plus $i - 1$ arêtes. Or la forêt, $F' = (S, E')$ avec $E' = (a''_1, \dots, a''_i)$ est une forêt couvrante de G' qui comporte i arêtes, ce qui contredit le résultat du lemme $\#$

Complexité.

En utilisant un tas pour trier les arêtes par poids croissants et les énumérer, puis une structure de données³ permettant de gérer efficacement une partition d'objets (ici l'évolution des composantes connexes), on montre qu'on a une complexité pour l'algorithme de KRUSKAL comparable à celle de l'algorithme de PRIM à savoir en $O(n \log p)$ où n est le nombre de sommets du graphe et p son nombre d'arêtes.

3. en l'occurrence cette structure de données s'appelle *Union-Find*