Chapitre 2

option informatique

Graphes

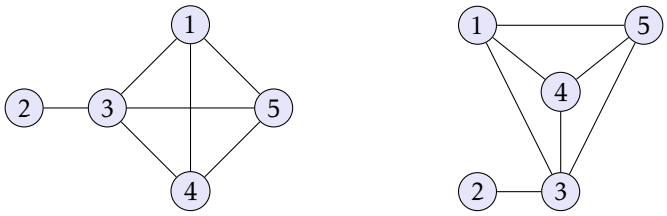
1. Introduction

De manière générale, un graphe permet de représenter les connexions d'un ensemble en exprimant les relations entre ses éléments : réseau de communication, réseau routier, circuit électronique, ... , mais aussi relations sociales ou interactions entre espèces animales.
Le vocabulaire de la théorie des graphes est utilisé dans de nombreux domaines : chimie, biologie, sciences sociales, etc., mais c'est avant tout une branche à part entière et déjà ancienne des mathématiques (le fameux problème des ponts de Königsberg d'Euler date de 1736). Néanmoins, l'importance accrue que revêt l'aspect algorithmique dans ses applications pratiques en fait aussi un domaine incontournable de l'informatique. Pour schématiser, les mathématiciens s'intéressent avant tout aux propriétés globales des graphes (graphes eulériens, graphes hamiltoniens, arbres, coloration, dénombrement, ...) là où les informaticiens vont plutôt chercher à concevoir des algorithmes efficaces pour résoudre un problème faisant intervenir un graphe (recherche du plus court chemin, problème du voyageur de commerce, recherche d'un arbre couvrant, ...). Tout ceci forme un ensemble très vaste, dont nous n'aborderons que quelques aspects, essentiellement de nature algorithmique.

1.1 Graphes non orientés

Un graphe G = (V, E) est défini par l'ensemble fini V = $\{v_1, v_2, \ldots, v_n\}$ dont les éléments sont appelés *sommets* (*vertices* en anglais), et par l'ensemble fini E = $\{e_1, e_2, \ldots, e_m\}$ dont les éléments sont appelés *arêtes* (*edges* en anglais).
Une arête e de l'ensemble E est définie par une paire non ordonnée de sommets, appelés les *extrémités* de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont *adjacents* (ou incidents avec e), ou bien que l'arête e est *incidente* avec les sommets a et b.
Enfin, on appelle *ordre* d'un graphe le nombre de sommets n de ce graphe. Le *degré* d'un sommet est le nombre de sommets qui lui sont adjacents, et le *degré* d'un graphe le degré maximum de tous ses sommets.

Les graphes tirent leur nom du fait qu'on peut les représenter graphiquement : à chaque sommet de G on fait correspondre un point du plan et on relie les points correspondant aux extrémités de chaque arête. Il existe donc une infinité de représentations possibles. Par exemple, les deux dessins qui suivent représentent le même graphe G, avec V = {1, 2, 3, 4, 5} et E = {(1, 3), (1, 4), (1, 5), (2, 3), (3, 4), (3, 5), (4, 5)} :



G est un graphe d'ordre 5 ; il possède un sommet de degré 1 (le sommet 2), trois sommets de degré 3 (les sommets 1, 4 et 5) et un sommet de degré 4 (le sommet 3).

Remarque. On peut observer que ce graphe a la particularité de posséder une représentation (celle de droite) pour laquelle les arêtes ne se coupent pas. Un tel graphe est dit *planaire*. Cette notion ne sera pas abordée dans la suite de ce cours.

Un graphe est dit *simple* lorsqu'aucun sommet n'est adjacent à lui-même. Par la suite, nous nous restreindrons à l'étude des graphes simples [1].

1. Un graphe présentant des arêtes reliant un sommet à lui-même ou plusieurs arêtes reliant les deux mêmes sommets est appelé un *multigraphe*.

Théorème. — *Si* G *est un graphe non orienté simple, La somme des degrés de ses sommets est égal à deux fois le nombre de ses arêtes :*

$$\sum_{v\in \mathrm{V}} \deg(v) = 2|\mathrm{E}|.$$

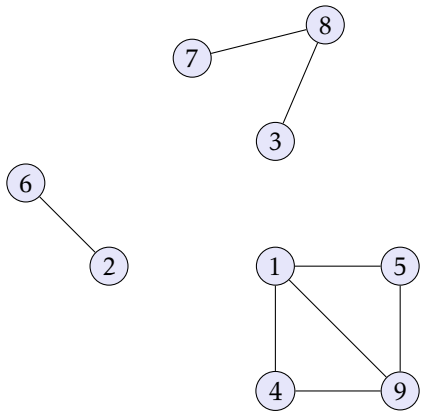
Preuve. Lorsqu'on fait la somme des degrés des sommets, chaque arête est comptée deux fois. □

• Chemins

Un *chemin* de longueur k reliant les sommets a et b est une suite finie $x_0 = a, x_1, \ldots, x_k = b$ de sommets tel que pour tout $i \in [\![0, k-1]\!]$, la paire (x_i, x_{i+1}) soit une arête de G. Ce chemin est dit *cyclique* lorsque $a = b$.

La *distance* entre deux sommets a et b est la plus petite des longueurs des chemins reliant a et b, si tant est qu'il en existe. Lorsque tous les sommets sont à distance finie les uns des autres, on dit que le graphe est *connexe*. Un graphe non connexe peut être décomposé en plusieurs *composantes connexes*, qui sont des sous-graphes connexes maximaux.

Par exemple, le graphe suivant possède trois composantes connexes :



Démontrons maintenant quelques résultats généraux qui nous seront utiles dans la suite de ce cours.

Théorème. — *Un graphe connexe d'ordre n possède au moins* $n-1$ *arêtes.*

Preuve. Ce résultat, comme la plus-part des résultats de ce chapitre, se prouve par récurrence sur l'ordre n du graphe.

- Ce résultat est évident si $n = 1$.
- Si $n > 1$, supposons ce résultat acquis au rang $n-1$, et considérons un graphe connexe G d'ordre n.
  Nous allons distinguer deux cas. Si G possède un sommet x de degré 1, supprimons-le ainsi que l'unique arête qui le relie au reste du graphe. Le graphe ainsi obtenu est toujours connexe donc comporte au moins $n-2$ arêtes, ce qui prouve que G possède au moins $1 + (n-2) = n-1$ arêtes.
  Dans le cas contraire, tous les sommets de G sont au moins de degré 2. Or la somme des degrés d'un graphe est égal à deux fois son nombre d'arêtes, donc G possède au moins n arêtes.

□

Lemme. — *Si dans un graphe* G *tout sommet est de degré supérieur ou égal à 2, alors* G *possède au moins un cycle.*

Preuve. Partons d'un sommet arbitraire v_1, et construisons une suite finie de sommets $v_1, v_2, \ldots, v_k$ de la façon suivante :

- $v_i \notin \{v_1, v_2, \ldots, v_{i-1}\}$;
- v_i est voisin de v_{i-1}.

Puisque les sommets de G sont en nombre fini, cette construction se termine. Or v_k est au moins de degré 2 donc il possède, outre v_{k-1}, un autre voisin v_j dans la séquence. Alors $(v_j, v_{j+1}, \ldots, v_k, v_j)$ est un cycle de G. □

Théorème. — *Un graphe acyclique d'ordre n comporte au plus* $n-1$ *arêtes.*

Preuve. Raisonnons par récurrence sur n.

– Ce résultat est bien évident si $n = 1$.

– Si $n > 1$, supposons le résultat acquis au rang $n-1$ et considérons un graphe acyclique d'ordre n. D'après le lemme précédent, G possède au moins un sommet x de degré 0 ou 1. Supprimons-le, ainsi éventuellement que l'arête qui lui est reliée. Le graphe obtenu est toujours acyclique et d'ordre $n-1$ donc par hypothèse de récurrence possède au plus $n-2$ arêtes. Ainsi, G possède au plus $n-1$ arêtes.

□

• Arbres

Un *arbre* est un graphe connexe acyclique [2]. D'après les deux théorèmes précédents, un arbre d'ordre n possède exactement $n-1$ arêtes. Plus précisément, on démontre le résultat suivant :

Théorème. — *Pour un graphe* G *d'ordre n, il y a équivalence des assertions suivantes :*

(i) G *est un arbre ;*

(ii) G *est un graphe connexe à* $n-1$ *arêtes ;*

(iii) G *est un graphe acyclique à* $n-1$ *arêtes.*

D'une certaine façon, un arbre est un graphe connexe minimal et un graphe acyclique maximal.

Preuve. Compte tenu de la définition d'un arbre, il suffit de montrer l'équivalence des propriétés *(ii)* et *(iii)*.

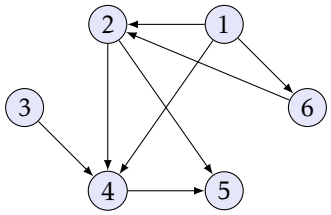
– Supposons qu'un graphe connexe G à $n-1$ arêtes possède un cycle. La suppression d'une arête de ce cycle créerai un graphe à $n-2$ arêtes toujours connexe, ce qui est absurde. G est donc aussi acyclique.

– Supposons qu'un graphe acyclique G à $n-1$ arêtes ne soit pas connexe. Il existerait alors deux sommets x et y qui ne peuvent être reliés par un chemin. Rajoutons alors une arête entre ces deux sommets. Le graphe obtenu est toujours acyclique mais possède maintenant n arêtes, ce qui est absurde. G est donc aussi connexe.

□

1.2 Graphes orientés

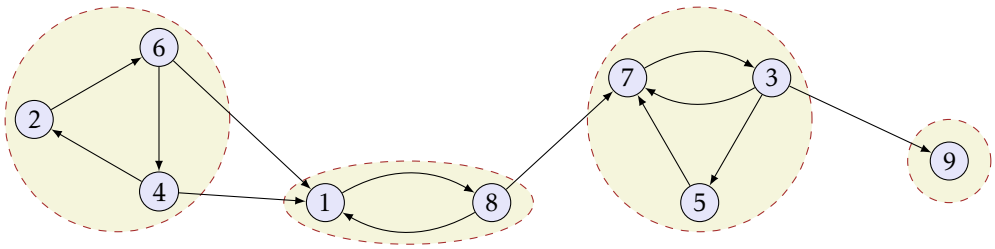
On obtient un graphe *orienté* en distinguant la paire de sommets (a,b) de la paire (b,a). Dans ce cas, on définit pour chaque sommet son degré *sortant* (égal au nombre d'arcs [3] dont il est la première composante) de son degré *entrant* (égal au nombre d'arcs dont il est la seconde composante). Par exemple, le graphe qui suit est défini par $V = \{1,2,3,4,5,6\}$ et $E = \{(1,2),(1,4),(1,6),(2,4),(2,5),(3,4),(4,5),(6,2)\}$:



Le sommet 1 a un degré sortant égal à 3 et un degré entrant égal à 0, tandis que le sommet 2 a un degré entrant et un degré sortant tous deux égaux à 2.

Les notions de chemin et de distance s'étendent sans difficulté aucune au cas des graphes orientés.
Un graphe orienté est dit *fortement connexe* lorsque pour tout couple de sommets (a,b) il existe un chemin reliant a à b et un chemin reliant b à a. Un graphe orienté peut être décomposé en composantes fortement connexes (des sous-graphes fortement connexes maximaux).
Par exemple, le graphe orienté suivant possède quatre composantes fortement connexes :

2. Un graphe acyclique mais non connexe est appelé une *forêt*, chacune de ses composantes connexes étant un arbre.
3. Dans le cas d'un graphe orienté, on parle souvent d'arc plutôt que d'arêtes.



• Arbre enraciné

Considérons de nouveau un arbre, c'est-à-dire, rappelons-le, un graphe connexe acyclique non orienté, et prouvons le résultat suivant :

Théorème. — *Si a et b sont deux sommets distincts d'un arbre* G, *il existe un unique chemin reliant a et b.*

Preuve. G est connexe, ce qui assure l'existence d'un tel chemin. Et s'il en existait un deuxième, nous pourrions former un cycle en empruntant le premier entre a et b puis le second entre b et a et ainsi contredire le caractère acyclique de G. □

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 accorde usuellement en informatique.

Preuve. Raisonnons par récurrence sur l'ordre n de l'arbre G.

- Si $n = 1$, il n'y a pas d'arête à orienter.
- Si $n > 1$, supposons le résultat acquis au rang $n-1$. Nous savons qu'un arbre G d'ordre n possède $n-1$ arêtes donc que le degré de G est égal à $2n-2$. Ceci montre qu'il existe au moins deux sommets de degré 1 donc au moins un qui soit différent de r ; notons-le a. Si on supprime ce sommet de G ainsi que l'arête qui le relie à l'arbre, on obtient un arbre d'ordre $n-1$ à qui on peut appliquer l'hypothèse de récurrence pour l'enraciner en r. Il reste à orienter l'arête supprimée en direction de a pour enraciner G en r.

□

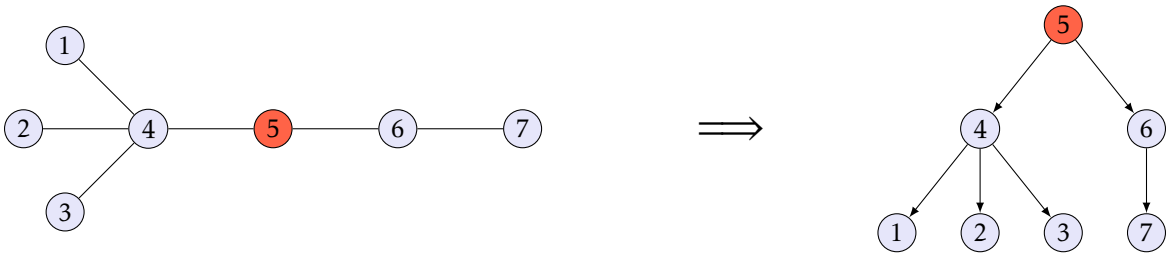


Figure 1 – Un exemple d'enracinement.

1.3 Mise en œuvre pratique

Deux méthodes principales s'offrent à nous 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 de *matrices d'adjacence* (le coefficient d'indice (i, j) traduit l'existence ou non d'une liaison entre deux sommets).

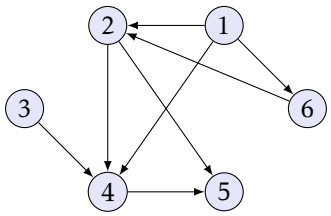
• Listes d'adjacence (première version)

Nous pouvons définir un sommet comme un enregistrement formé d'un identifiant et de la liste des identifiants de ses voisins, et un graphe comme une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```

Cette représentation est la plus souple : elle présente l'avantage de permettre d'ajouter ou de supprimer facilement des sommets et des arêtes, mais l'inconvénient de ne pas permettre un accès rapide à un sommet ou une arête particulière.

En général, les sommets de chaque liste d'adjacence sont rangés selon un ordre arbitraire.




```
# let g = [{Id = 1; Voisins = [2; 4; 6]};
           {Id = 2; Voisins = [4; 5]};
           {Id = 3; Voisins = [4]};
           {Id = 4; Voisins = [5]};
           {Id = 5; Voisins = []};
           {Id = 6; Voisins = [2]}] ;;
```

FIGURE 2 – un graphe orienté et sa représentation par le type *int graph*.

Ajout et suppression d'un sommet ou d'une arête

L'ajout et la suppression d'une arête dans un graphe de type *'a graph* se réalisent de la façon suivante :

```
let rec ajoute_arete g a b = match g with
  | []                -> failwith "ajoute_arete"
  | s::q when s.Id = a && mem b s.Voisins -> g
  | s::q when s.Id = a -> {Id = a ; Voisins = b::(s.Voisins)}::q
  | s::q               -> s::(ajoute_arete q a b) ;;


let rec supprime_arete g a b =
  let rec aux = function
    | []              -> failwith "supprime_arete"
    | t::q when t = b -> q
    | t::q            -> t::(aux q)
  in match g with
    | []                -> failwith "supprime_arete"
    | s::q when s.Id = a -> {Id = a ; Voisins = aux s.Voisins}::q
    | s::q               -> s::(supprime_arete q a b) ;;
```

On peut observer que ces fonctions ne s'appliquent qu'à des graphes orientés. Pour des graphes non orientés, il faut ajouter/supprimer à la fois l'arête reliant a et b et l'arête reliant b et a.

L'ajout et la suppression d'un sommet, toujours pour le type *'a graph*, se réalisent ainsi :

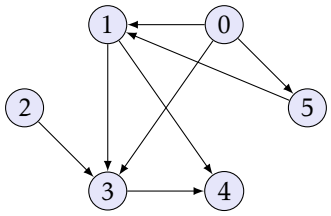
```
let rec ajoute_sommet g a = match g with
  | []                -> [{Id = a ; Voisins = []}]
  | s::q when s.Id = a -> g
  | s::q               -> s::(ajoute_sommet q a) ;;


let rec supprime_sommet g a =
  let rec aux = function
    | []              -> []
    | t::q when t = a -> q
    | t::q            -> t::(aux q)
  in match g with
    | []                -> []
    | s::q when s.Id = a -> supprime_sommet q a
    | s::q               -> let ns = {Id = s.Id ; Voisins = aux s.Voisins}
                            in ns::(supprime_sommet q a) ;;
```

• Listes d'adjacence (seconde version)

La représentation que nous venons d'étudier possède la propriété avantageuse de ne demander qu'une quantité minimale de mémoire ($\Theta(n+p)$ si n désigne le nombre de sommets et p le nombre d'arêtes) et de permettre l'ajout et la suppression des sommets comme des arêtes. En contrepartie, elle présente l'inconvénient de ne pas permettre un accès rapide, ni aux sommets, ni aux arêtes.
Sachant que par la suite les algorithmes que nous allons étudier vont opérer sur des graphes statiques (c'est-à-dire sans ajout ni suppression de sommet ou d'arête), nous allons modifier cette première représentation en utilisant désormais un tableau pour stocker les listes d'adjacence. Ainsi, nous pourront accéder *en temps constant* à la liste d'adjacence de chacun des sommets.
Notons enfin que dans ce cas, il peut être intéressant de faire coincider indices du tableau et identifiants des sommets. En Caml par exemple, on numérotera les sommets de 0 à $n-1$ et on utilisera le type :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```




```
# let g = [| [1; 3; 5];
             [3; 4];
             [3];
             [4];
             [];
             [1] |] ;;
```

Figure 3 – un graphe orienté et sa représentation par le type *graphe*.

Cette nouvelle représentation des listes d'adjacence est toujours aussi économique en espace ($\Theta(n+p)$) et permet encore l'ajout ou la suppression d'une arête, mais plus l'ajout ou la suppression d'un sommet.

Désorientation d'un graphe
Un inconvénient potentiel de la représentation par listes d'adjacence est que, pour déterminer si un arc (a,b) est présent dans un graphe, il n'existe pas de moyen plus rapide que de rechercher b dans la liste d'adjacence de a. En particulier, il est difficile de déterminer rapidement si un graphe est non orienté : il faut pour chacune des arêtes vérifier que l'arête réciproque est aussi présente.
Désorienter un graphe consiste à calculer le plus petit graphe non orienté g' contenant g. Pour toute arête reliant les sommets a et b il faut donc ajouter l'arête reliant b à a, si celle-ci ne figure pas déjà dans le graphe.

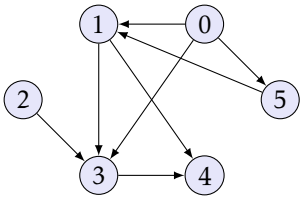
```
let desoriente g =
  let n = vect_length g in
  let aux i j = if not mem i g.(j) then g.(j) <- i::g.(j) in
  for i = 0 to n-1 do
    do_list (aux i) g.(i)
  done ;;
```

• Matrices d'adjacence

Si on veut en plus accéder en coût constant à chacune des arêtes, il devient nécessaire d'ordonner les sommets $V = \{v_1, v_2, \ldots, v_n\}$ et utiliser une matrice $M \in \mathcal{M}_n(\{0,1\})$ pour représenter les arêtes :

$$m_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 0 & \text{sinon} \end{cases}$$

Cette représentation a des avantages : l'ajout et la suppression d'une arête a un coût constant ; déterminer si un graphe est orienté est chose aisée (il suffit de vérifier que la matrice est symétrique), mais elle présente l'inconvénient d'occuper beaucoup plus d'espace mémoire, en particulier lorsque le nombre d'arêtes est réduit vis-à-vis du nombre de sommets : si $n = |V|$ et $p = |E|$, le coût spatial de la représentation par matrice d'adjacence est un $\Theta(n^2)$, contre un $\Theta(n+p)$ pour une liste d'adjacence.



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

FIGURE 4 – un graphe orienté et sa matrice d'adjacence.

Il peut être utile de passer d'une représentation à une autre, aussi allons-nous écrire une fonction qui calcule la matrice d'adjacence d'un graphe. Pour en simplifier l'écriture, nous supposerons que les sommets du graphe sont les entiers de 0 jusqu'à $n-1$ et nous utiliserons le type *graphe* pour représenter le graphe.

```
let graphe_to_mat g =
  let n = vect_length g in
  let m = make_matrix n n 0 in
  for a = 0 to n-1 do
      do_list (function b -> m.(a).(b) <- 1) g.(a)
    done ;
  m ;;
```

À l'inverse, la fonction qui suit détermine le graphe associée à une matrice d'adjacence donnée :

```
let mat_to_graphe m =
  let n = vect_length m in
  let g = make_vect n [] in
  for a = 0 to n-1 do
    for b = 0 to n-1 do
      if m.(a).(b) = 1 then g.(a) <- b::g.(a)
    done
  done ;
  g ;;
```

2. Parcours d'un graphe

Parcourir un graphe, c'est énumérer l'ensemble des sommets accessibles par un chemin à partir d'un sommet donné, dans l'objectif de leur faire subir un certain traitement. Différentes solutions sont possibles mais en règle générale celles-ci tiennent à jour deux listes : la liste des sommets rencontrés (« 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 $t \in$ voisins(s) **do**
      if $t \notin$ déjàVus **then**
        àTraiter $\leftarrow t$
        déjàVus $\leftarrow t$

Arborescence associée à un parcours

À chaque parcours débutant par un sommet 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 » de l'algorithme ci-dessus on ajoute le sommet t et l'arête (s,t). On construit ainsi un graphe connexe ayant k sommets et $k-1$ arêtes, autrement dit 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 aussi qu'au plus une fois. Si ces opérations d'entrée et de sortie dans la liste sont de coût constant (ce qui sera effectivement le cas dans la suite), le coût total des manipulations de la liste « àTraiter » est un $O(n)$, avec $n = |V|$.
Chaque liste d'adjacence est parcourue au plus une fois donc le temps total consacré à scruter les listes de voisinage est un $O(p)$ avec $p = |E|$, *à condition de déterminer si un sommet a déjà été vu en coût constant*. Dans ce cas, le coût total d'un parcours est un $O(n+p)$.
Pour réaliser cette condition, la solution que nous adopterons consistera à utiliser un tableau booléen pour représenter « déjàVus », destiné à marquer chaque sommet au moment où il entre dans la liste « àTraiter ».

Nous allons maintenant 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. Dans ce qui suit, nous considérerons un graphe défini par liste d'adjacence (avec le type *graphe*) ainsi qu'une fonction **`traitement`** de type *int -> unit*.

2.1 Parcours en largeur

L'algorithme de parcours en largeur (appelé BFS, pour *Breadth First Search*) consiste à utiliser une file d'attente [4] pour stocker les sommets à traiter : tous les voisins sont traités avant de parcourir le reste du graphe. Ainsi, sont traités successivement tous les sommets à une distance égale à 1 du sommet initial, puis 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.

Nous aurons besoin du module **`queue`** de la bibliothèque standard :

```
#open "queue" ;;
```

Le parcours en largeur s'écrit alors :

```
let bfs g s =
  let dejavu = make_vect (vect_length g) false
  and atraiter = new() in
  add s atraiter ; dejavu.(s) <- true ;
  let rec ajoute_voisin = function
    | []                    -> ()
    | t::q when dejavu.(t) -> ajoute_voisin q
    | t::q                  -> add t atraiter ; dejavu.(t) <- true ;
                               ajoute_voisin q
  in
  try while true do
      let s = take atraiter in
      traitement s ;
      ajoute_voisin g.(s)
    done
  with Empty -> () ;;
```

Illustrons cette fonction à partir du sommet $s_0 = 2$ du graphe présenté figure 5, en appliquant **`print_int`** en guise de traitement :

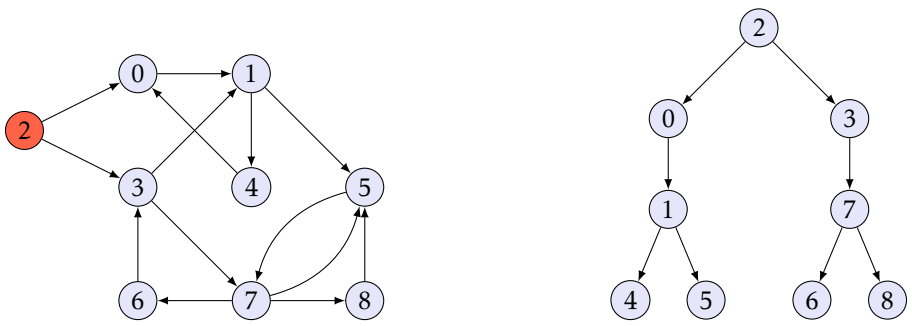


FIGURE 5 – un graphe et l'arborescence associée à un parcours BFS.

4. FIFO, pour *First In, First Out*, voir cours de première année.

```
# bfs g 2 ;;
203174568- : unit = ()
```

Lors du traitement, la file d'attente présente les états suivants :

$$\boxed{2} \Longrightarrow \boxed{0\,3} \Longrightarrow \boxed{3\,1} \Longrightarrow \boxed{1\,7} \Longrightarrow \boxed{7\,4\,5} \Longrightarrow \boxed{4\,5\,6\,8} \Longrightarrow \boxed{5\,6\,8} \Longrightarrow \boxed{6\,8} \Longrightarrow \boxed{8}$$

On constate que le parcours en largueur correspond à un parcours hiérarchique dans l'arborescence associée.

2.2 Parcours en profondeur

L'algorithme de parcours en profondeur (appelé DFS, pour *Depth First Search*) consiste à utiliser une pile [5] pour stocker les éléments à traiter. Cette fois, on explore chaque chemin jusqu'au bout avant de passer au chemin suivant.
Sa mise en œuvre ne diffère que très peu de l'algorithme BFS : il suffit de remplacer la file d'attente par une pile du module **stack** :

```
#open "stack" ;;
```

Le parcours en profondeur s'écrit alors :

```
let dfs g s =
  let dejavu = make_vect (vect_length g) false
  and atraiter = new() in
  push s atraiter ; dejavu.(s) <- true ;
  let rec ajoute_voisin = function
    | []                   -> ()
    | t::q when dejavu.(t) -> ajoute_voisin q
    | t::q                 -> push t atraiter ; dejavu.(t) <- true ;
                              ajoute_voisin q
  in
  try while true do
      let s = pop atraiter in
      traitement s ;
      ajoute_voisin g.(s)
    done
  with Empty -> () ;;
```

Appliqué au graphe présenté figure 6 on obtient cette fois :

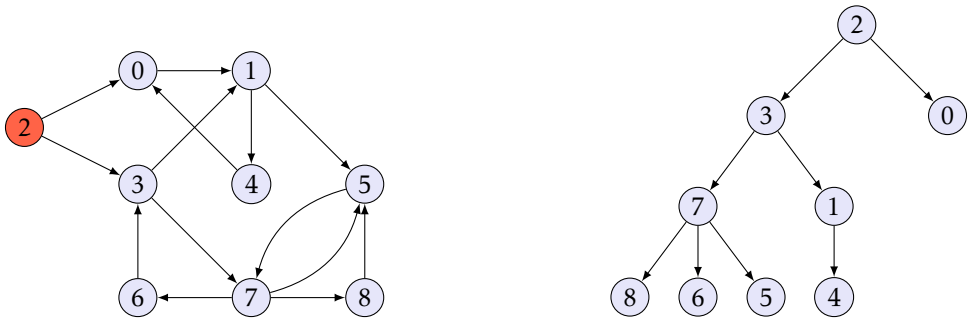
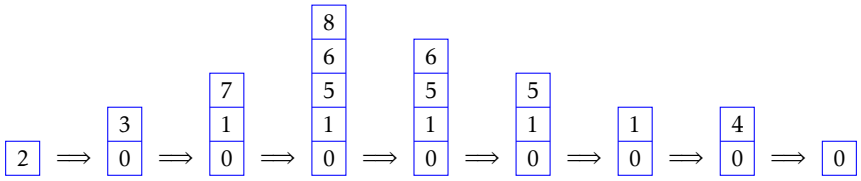


Figure 6 – un graphe et l'arborescence associée à un parcours DFS.

```
# dfs g 2 ;;
237865140- : unit = ()
```

et la pile présente les états suivants :

5. LIFO, pour *Last In, First Out*.



On constate que le parcours en profondeur correspond à un parcours préfixe dans l'arborescence associée.

Notons enfin que l'usage d'une pile laisse présager l'existence d'un algorithme récursif pour le DFS :

```
let dfs_rec g s =
  let dejavu = make_vect (vect_length g) false in
  let rec aux = function
    | s when dejavu.(s) -> ()
    | s                 -> dejavu.(s) <- true ;
                           traitement s ;
                           do_list aux g.(s)
  in aux s ;;
```

Nous allons pour finir donner deux applications du parcours en profondeur : le calcul des composantes connexes d'un graphe connexe non orienté et le tri topologique d'un graphe orienté acyclique.

2.3 Calcul des composantes connexes d'un graphe non orienté

Dans le cas d'un graphe non orienté, les sommets atteints par un algorithme de parcours correspondent à la composante connexe du sommet initial. Pour obtenir toutes les composantes connexes d'un graphe, il suffit donc de reprendre un nouveau parcours à partir d'un sommet non encore atteint, tant qu'il en existe.

Pour réaliser cet algorithme, on utilise la version récursive du parcours en profondeur en remplaçant le traitement par un accumulateur transportant la liste des sommets rencontrés :

```
let liste_composantes g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           it_list dfs (s::lst) g.(s)
  and aux comp = function
    | s when s = vect_length g -> comp
    | s when dejavu.(s)        -> aux comp (s+1)
    | s                        -> aux ((dfs [] s)::comp) (s+1)
  in aux [] 0 ;;
```

`lst` est un accumulateur qui transporte les sommets faisant partie de la composante connexe en cours d'exploration, **`comp`** est un accumulateur qui transporte les composantes connexes déjà trouvées.

2.4 Tri topologique

Le *tri topologique* d'un graphe orienté acyclique $G = (V, E)$ consiste à ordonner tous les sommets de sorte que si $G\ (a, b) \in E$ est un arc de G alors a apparaît avant b dans le tri. On rencontre cette notion dans de nombreux problèmes d'ordonnancement : chaque sommet représente une tâche à effectuer, et les arcs indiquent celles de ces tâches qui doivent être réalisées avant une autre tâche.
En d'autres termes, les arcs orientés définissent un ordre partiel sur les sommets, et il s'agit de prolonger cet ordre en un ordre total sur l'ensemble des sommets (illustration figure 7).
L'algorithme de tri topologique est simple : à partir de chaque sommet vierge on effectue un parcours en profondeur ; une fois le parcours achevé ce sommet est inséré en tête d'une liste chaînée. On réitère ce procédé jusqu'à exhaustion des sommets vierges.
Une fois le parcours achevé la liste chaînée classe les sommets par ordre croissant.

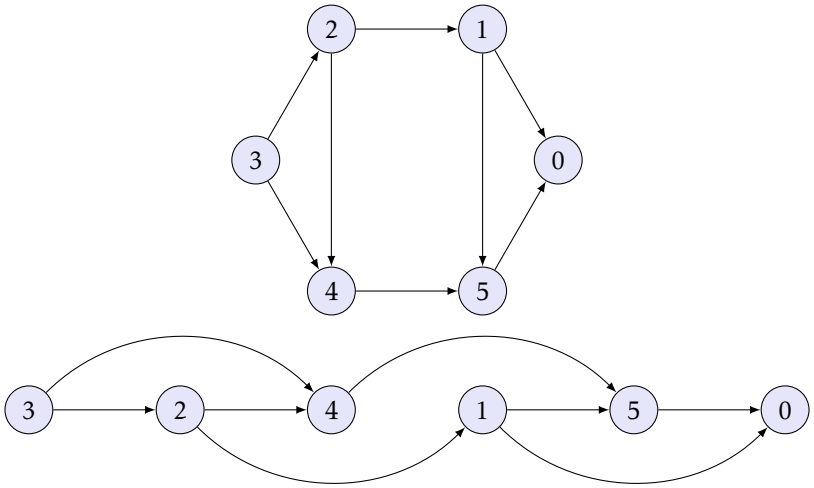


FIGURE 7 – Un exemple d'ordre topologique d'un graphe orienté acyclique. On notera que lorsque le graphe est dessiné suivant l'ordre topologique les arcs sont tous orientés de gauche à droite.

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

La validité de cet algorithme repose sur le résultat suivant :

LEMME. — *S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.*

Preuve. Lors du parcours en profondeur vient forcément un moment où a est vu pour la première fois et ses voisins examinés. À ce moment a n'est pas encore entré dans la liste chaînée.

- Si b n'a pas encore été vu, le parcours en profondeur se poursuit à partir de b ; une fois ce dernier achevé b rentre dans la liste chaînée, et a y rentrera plus tard.
- Si b a déjà été vu et est déjà rentré dans la liste chaînée, le résultat est acquis.
- Reste à examiner le cas où b a déjà été vu mais n'est pas encore entré dans la liste chaînée. Ceci signifie que l'algorithme est en train de parcourir une branche issue de b, avec pour conséquence l'existence d'un chemin reliant b à a. Mais puisque b est voisin de a, ceci implique l'existence d'un cycle dans G, ce qui est exclu.

□

3. Plus court chemin

Déterminer le plus court chemin entre deux sommets d'un graphe non pondéré n'est pas difficile : il suffit d'effectuer un parcours en largeur à partir d'un des deux 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 et définissent le *poids* d'un chemin comme la somme des poids des arêtes qui le composent. Imaginons par exemple que les arêtes du graphe de la figure 8 représentent un réseau de transport maritime et ferroviaire et les nœuds des centres de transit. On peut rechercher un trajet qui va de Lisbonne à Bucarest en minimisant le nombre de transits : une solution est de

passer par Londres, Amsterdam, Berlin et Varsovie (ou par Madrid, Paris, Berlin et Varsovie). Mais on peut aussi prendre en compte la durée associée à chaque trajet ; dans ce cas le chemin le plus rapide ne sera pas forcément égal au trajet précédent : il est plus intéressant de passer par Madrid, Barcelone, Lyon, Munich, Prague et Budapest.

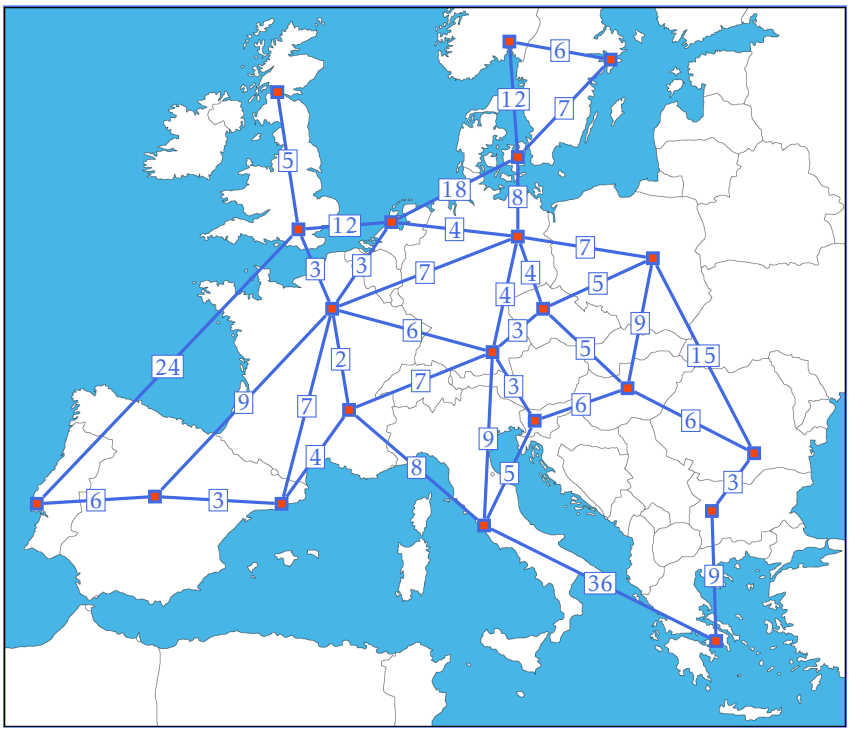


Figure 8 – Un exemple de graphe pondéré.

Commençons par donner quelques définitions.

Étant donné un graphe (orienté ou non) $G = (V, E)$, on appelle *pondération* une application $w : E \to \mathbb{R}$, et pour tout $(a, b) \in E$ on dit que $w(a, b)$ est le *poids* de l'arête (a, b).
Il sera par la suite commode de prolonger la définition de w sur $V \times V$ en posant :

$$\forall (a,b) \in (V \times V) \setminus E, \quad w(a,b) = \begin{cases} 0 & \text{si } a = b \\ +\infty & \text{sinon} \end{cases}$$

Le *poids* d'un chemin est la somme des poids des arêtes qui le composent. On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b, s'il en existe. Dans le cas contraire, on posera $\delta(a, b) = +\infty$.

Remarque. En présence de poids négatifs il convient de prendre quelques précautions pour assurer l'existence de ce minimum. En particulier, s'il existe un chemin menant de a et b et comprenant un circuit fermé de poids strictement négatif, il convient de poser $\delta(a, b) = -\infty$ car dans ce cas on peut faire indéfiniment décroitre le poids de ce chemin en multipliant les passages par cette boucle. On peut éviter cette situation en supposant par exemple que tous les poids sont positifs.

Il existe trois problème de plus courts chemins :

(i) calculer le chemin de poids minimal entre une source a et une destination b ;

(ii) calculer les chemins de poids minimal entre une source a et tout autre sommet 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 donnerons une solution de ces deux problèmes en étudiant l'algorithme de Dijkstra. Il faut cependant noter qu'il existe de multiples algorithmes de plus courts chemins, souvent adaptés à un type particulier de graphe.

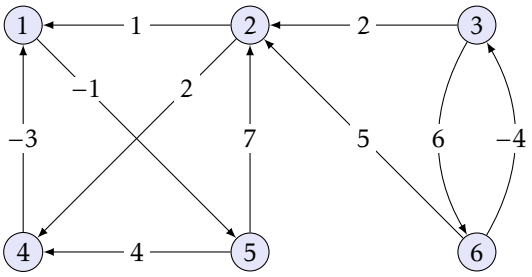
On notera enfin que les algorithmes que nous allons étudier sont basés sur le résultat suivant :

Lemme (principe de sous-optimalité). — *Si $a \rightsquigarrow b$ est un plus court chemin qui passe par c, alors $a \rightsquigarrow c$ et $c \rightsquigarrow b$ sont eux aussi des plus courts chemins.*

Preuve. S'il existait un chemin plus court entre par exemple a et c, il suffirait de le suivre lors du trajet entre a et b pour contredire le caractère minimal du trajet $a \rightsquigarrow b$. □

3.1 Algorithme de Floyd-Warshall

Pour intégrer la notion de poids à la définition des graphes, nous allons modifier la définition de la matrice d'adjacence de G en convenant que désormais, $m_{ij} = w(v_i, v_j)$ (voir figure 9).



$$M = \begin{pmatrix} 0 & +\infty & +\infty & +\infty & -1 & +\infty \\ 1 & 0 & +\infty & 2 & +\infty & +\infty \\ +\infty & 2 & 0 & +\infty & +\infty & 6 \\ -3 & +\infty & +\infty & 0 & +\infty & +\infty \\ +\infty & 7 & +\infty & 4 & 0 & +\infty \\ +\infty & 5 & -4 & +\infty & +\infty & 0 \end{pmatrix}$$

Figure 9 – Un exemple de graphe pondéré et de sa matrice d'adjacence.

Si n désigne l'ordre du graphe G, l'algorithme de Floyd-Warshall consiste à calculer la suite finie de matrices $M^{(k)}$, $0 \leqslant k \leqslant n$ avec $M^{(0)} = M$ et :

$$\forall k < n, \quad \forall (i,j) \in \mathbb{N}^2, \qquad m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Théorème. — *Si G ne contient pas de cycle de poids strictement négatif, alors $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne passant que par des sommets de la liste $v_1, v_2, \ldots, v_k$.*

De ceci il découle immédiatement que $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

Preuve. On raisonne par récurrence sur k.

- Si $k = 0$, $m_{ij}^{(0)} = m_{ij}$ est bien entendu le poids du chemin minimal reliant v_i à v_j sans passer par aucun autre sommet.
- Si $k < n$, supposons le résultat acquis au rang k et considérons un chemin $v_i \rightsquigarrow v_j$ ne passant que par les sommets $v_1, \ldots, v_{k+1}$ et de poids minimal.

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

  Si ce chemin passe par v_{k+1}, alors par principe de sous-optimalité les chemins $v_i \rightsquigarrow v_{k+1}$ et $v_{k+1} \rightsquigarrow v_j$ sont minimaux et ne passent que par des sommets de la liste $v_1, \ldots, v_k$ donc par hypothèse de récurrence son poids total est égal à $m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}$ (car il n'existe pas de cycle de poids négatif reliant v_{k+1} à lui-même).

  De ceci il résulte que $\min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right) = m_{ij}^{(k+1)}$ est le poids minimal d'un plus court chemin reliant v_i et v_j et ne passant que par des sommets de la liste $v_1, \ldots, v_{k+1}$.

□

• Mise en œuvre pratique

Nous allons supposer que les poids sont à valeurs entières et définir le type :

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

ce qui nous permet de représenter un graphe pondéré par le type *poids vect vect*.
On définit la somme et le minimum de deux objets de type *poids* :

```
let som = fun
  | Inf _       -> Inf
  | _ Inf       -> Inf
  | (P a) (P b) -> P (a + b) ;;

let mini = fun
  | Inf x       -> x
  | x Inf       -> x
  | (P a) (P b) -> P (min a b) ;;

let inferieur = fun
  | Inf _       -> false
  | _ Inf       -> true
  | (P a) (P b) -> a < b ;;
```

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

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

reste valable si on remplace $m_{i,k+1}^{(k)}$ par $m_{i,k+1}^{(k+1)}$ ou $m_{k+1,j}^{(k)}$ par $m_{k+1,j}^{(k+1)}$. En effet, en l'absence de cycles de poids négatif on a $m_{i,k+1}^{(k)} = m_{i,k+1}^{(k+1)}$ et $m_{k+1,j}^{(k)} = m_{k+1,j}^{(k+1)}$ donc l'ordre dans lequel sont modifiées les cases d'indices (i,j), $(i,k+1)$ et $(k+1,j)$ n'a pas d'importance.

```
let floydwarshall w =
  let n = vect_length w in
  let m = make_matrix n n Inf in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(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 ;;
```

Appliqué au graphe pondéré donné figure 9, cet algorithme retourne la matrice :

$$M^{(6)} = \begin{pmatrix} 0 & 6 & +\infty & 3 & -1 & +\infty \\ -1 & 0 & +\infty & 2 & -2 & +\infty \\ 1 & 2 & 0 & 4 & 0 & 6 \\ -3 & 3 & +\infty & 0 & -4 & +\infty \\ 1 & 7 & +\infty & 4 & 0 & +\infty \\ -3 & -2 & -4 & 0 & -4 & 0 \end{pmatrix}$$

On observe que les sommets 3 et 6 ne sont pas accessibles à partir des sommets 1, 2, 4 et 5. En revanche, on peut aller du sommet 6 au sommet 1 pour un coût total égal à -3 (il n'est pas difficile de deviner qu'il faut passer par les sommets 3, 2 et 4) ou du sommet 3 au sommet 5 pour un coût total nul (en passant par les sommets 2, 4 et 1).

Remarque. De manière évidente la complexité temporelle de l'algorithme de Floyd-Warshall est en $\Theta(n^3)$, où n est l'ordre du graphe et la complexité spatiale en $O(n^2)$.

Détermination des chemins de poids minimal

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

```
let pluscourtschemins w =
  let n = vect_length w in
  let m = make_matrix n n Inf
  and c = make_matrix n n [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(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)@[k+1]@c.(k).(j))
      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 ;;
```

La matrice c contient la liste des sommets intermédiaires par lesquels passer pour obtenir le chemin de poids minimal. La fin du code ci-dessus ne sert qu'à y rajouter les extrémités.

Appliqué au graphe donné en exemple figure 9, on obtient la matrice des plus courts chemins suivante :

$$\begin{pmatrix} [] & [1;5;2] & [] & [1;5;4] & [1;5] & [] \\ [2;4;1] & [] & [] & [2;4] & [2;4;1;5] & [] \\ [3;2;4;1] & [3;2] & [] & [3;2;4] & [3;2;4;1;5] & [3;6] \\ [4;1] & [4;1;5;2] & [] & [] & [4;1;5] & [] \\ [5;4;1] & [5;2] & [] & [5;4] & [] & [] \\ [6;3;2;4;1] & [6;3;2] & [6;3] & [6;3;2;4] & [6;3;2;4;1;5] & [] \end{pmatrix}$$

On peut observer que parmi tous ces chemins de poids minimal, le plus long relie le sommet 6 au sommet 5 en passant par les sommets 3, 2, 4 et 1, pour un poids total égal à -4.

• Application au calcul de la fermeture transitive d'un graphe

Considérons de nouveau un graphe non pondéré, orienté ou non. Le problème de la *fermeture transitive* consiste à déterminer si deux sommets a et b peuvent être reliés par un chemin allant de a à b. Pour le résoudre, nous allons utiliser la matrice d'adjacence associée à ce graphe, mais cette fois en utilisant les valeurs booléennes **`true`** pour dénoter l'existence d'une arête et **`false`** pour en marquer l'absence. Remplaçons maintenant dans l'algorithme de Floyd-Warshall la relation de récurrence sur les coefficients des matrices $\mathrm{M}^{(k)}$ par :

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

Il n'est pas difficile de prouver que le booléen $m_{ij}^{(k)}$ dénote l'existence d'un chemin reliant les sommets v_i et v_j en ne passant que par les sommets $v_1, v_2, \ldots, v_k$ et que par voie de conséquence la matrice $\mathrm{M}^{(n)}$ résout le problème de la fermeture transitive.

L'algorithme ainsi modifié est connu sous le nom d'algorithme de Warshall :

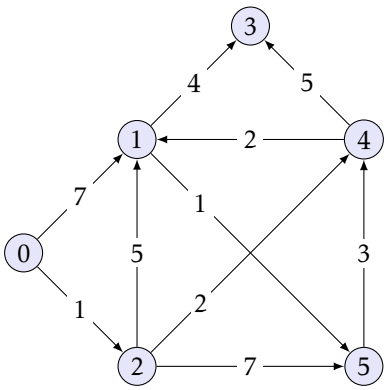
```
let warshall w =
  let n = vect_length w in
  let m = make_matrix n n false in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(i).(j) = 1
    done
  done ;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- m.(i).(j) || (m.(i).(k) && m.(k).(j))
      done
    done
  done ;
  m ;;
```

3.2 Algorithme de Dijkstra

Intéressons nous maintenant au problème des plus courts chemins à partir d'une même source $s \in V$. L'algorithme que nous allons étudier nécessite de supposer tous les poids positifs [6].
Nous allons supposer en outre les sommets $V = \{0, 1, \dots, n-1\}$ ordonnés et la matrice d'adjacence M modifiée comme pour l'algorithme de Floyd-Warshall. À partir d'une source $s \in V$ l'algorithme de Dijkstra va progressivement remplir un tableau d de longueur n de sorte qu'à la fin de cet algorithme d_v soit égal à $\delta(s, v)$, le poids du plus court chemin entre s et v. Pour ce faire, on fait évoluer une partition de $[\![1, n]\!]$, S initialisé à $\{s\}$ et son complémentaire $\overline{S}$.
L'ensemble S est destiné à représenter les sommets dont on a déterminé dans le tableau d le poids du chemin minimal à partir de s. Pour les éléments de $\overline{S}$, le tableau d contiendra le poids du plus court chemin *ne passant que par des sommets appartenant à* S. À chaque itération on choisit le sommet de $\overline{S}$ dont la valeur associée dans le tableau d est minimale pour le transférer dans S, et on modifie le tableau d en conséquence. Ainsi, chaque élément de $\overline{S}$ va progressivement être transféré dans S.
L'algorithme se déroule de la façon suivante :

function Dijkstra(sommet : s)
  $S \leftarrow \{s\}$
  for all $v \in V$ **do**
    $d_v \leftarrow w(s, v)$
  while $\overline{S} \neq \emptyset$ **do**
    Soit $u \in \overline{S} \mid d_u = \min\left(d_v \mid v \in \overline{S}\right)$
    $S \leftarrow S \cup \{u\}$
    for $v \in \overline{S}$ **do**
      $d_v \leftarrow \min\left(d_v, d_u + w(u, v)\right)$
  return d

Avant de justifier la validité de cet algorithme, voyons comment il se déroule dans le graphe suivant, à partir du sommet 0 :



6. L'exercice 12 présente un algorithme qui s'affranchit de cette hypothèse.

Observons l'évolution de la liste d au cours de l'algorithme :

S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8
$\{0,2,4\}$	·	5	·	8	·	8
$\{0,2,4,1\}$	·	·	·	8	·	6
$\{0,2,4,1,5\}$	·	·	·	8	·	·
$\{0,2,4,1,5,3\}$	·	·	·	·	·	·

Nous avons donc $\delta(0,1)=5$, $\delta(0,2)=1$, $\delta(0,3)=8$, $\delta(0,4)=3$, $\delta(0,5)=6$.

Théorème. — *À l'issue de l'algorithme de* Dijkstra *on a :* $\forall u \in V,\ d_u = \delta(s,u)$.

Preuve. On raisonne par récurrence sur $|S|$ en prouvant l'invariant suivant :

$$\forall u \in S,\quad d_u = \delta(s,u)$$
$$\forall v \in \overline{S},\quad d_v = \min\Big\{d_u + w(u,v) \;\Big|\; u \in S\Big\}$$

- Lorsque $|S|=1$ nous avons $S=\{s\}$, $d_s=0$ et $\forall v \neq s,\ d_v = w(s,v)$ donc le résultat annoncé est bien vrai.
- Lorsque $|S|>1$, supposons l'invariant acquis à l'étape précédente et distinguons trois cas :
  (i) u est déjà dans S : dans ce cas $d_u = \delta(s,u)$ et cette valeur n'est pas modifiée.
  (ii) u entre dans S : dans ce cas $u \in \overline{S}$ vérifie $\forall v \in \overline{S},\ d_u \leqslant d_v$ et il s'agit de prouver que $d_u = \delta(s,u)$.
  Considérons un plus court chemin $s \rightsquigarrow u$ et notons v le premier sommet de ce parcours qui ne soit pas dans S (éventuellement on peut avoir $v=u$). D'après le principe de sous-optimalité, $s \rightsquigarrow v$ et $v \rightsquigarrow u$ sont des plus courts chemins.
  Puisque toutes les pondérations sont positives, $\delta(s,u) \geqslant \delta(s,v)$. Or par hypothèse de récurrence appliquée à v, $d_v = \min\Big\{d_x + w(x,v) \;\Big|\; x \in S\Big\} = \min\Big\{\delta(s,x) + w(x,v) \;\Big|\; x \in S\Big\} = \delta(s,v)$ (n'oublions pas que $s \rightsquigarrow v$ est un plus court chemin qui ne passe que par des sommets de S).
  De ceci il résulte que $\delta(s,u) \geqslant d_v$. Par ailleurs, le choix de u impose $d_u \leqslant d_v$ donc $\delta(s,u) \geqslant d_u$. Mais d_u est le poids d'un chemin menant de s à u donc en définitive $d_u = \delta(s,u)$.
  (iii) v reste dans $\overline{S}$. Dans ce cas, si v n'est pas voisin de l'élément qui entre dans S la valeur de d_v n'est pas modifiée, et dans le cas contraire sa valeur est modifiée pour continuer à respecter l'égalité $d_v = \min\Big\{d_u + w(u,v) \;\Big|\; u \in S\Big\}$.

□

Étude de la complexité

Étudier la complexité de l'algorithme de Dijkstra est délicat car tributaire du choix de la représentation des structures de données. Grossièrement, on effectue $n-1$ transferts de $\overline{S}$ vers S en ayant à chaque fois déterminé le plus petit élément d'une partie du tableau d puis en ayant modifié certaines de ses valeurs en conséquence. Tout ceci a un coût linéaire, donc la complexité totale est *a priori* un $O(n^2)$.
Cette complexité est optimale pour des graphes denses, c'est-à-dire pour lesquels les sommets ont de nombreux voisins (le pire des cas correspondant aux graphes complets dont le nombre d'arêtes, égal à $\binom{n}{2}$, impose de toute façon un coût au moins quadratique). En revanche, lorsque le graphe est creux, il est possible d'améliorer cette complexité en utilisant une file de priorité (autrement dit un tas) pour représenter $\overline{S}$.
On sait qu'un tas-min permet la récupération de l'élément de priorité minimale (ici la valeur d_i) pour un coût en $O(\log n)$ (correspondant à la reformation du tas) donc le coût total du transfert de $\overline{S}$ à S est un $O(n \log n)$. Par ailleurs, chaque mise a jour d'une valeur du tableau d a lui aussi un coût en $O(\log n)$ (là encore pour reformer le tas $\overline{S}$), mais chaque arête ne va intervenir qu'au plus une fois dans ces modifications de d, donc si p désigne le nombre d'arêtes du graphe le coût total de ces modifications est un $O(p \log n)$.
Au final, ceci permet d'envisager un coût total en $O((n+p)\log n)$.

Pour qu'il soit intéressant d'utiliser un tas, il faut donc que $p \log n = O(n^2)$, autrement dit que $p = O\Big(\dfrac{n^2}{\log n}\Big)$.

• Mise en œuvre pratique

Pour que le coût de l'algorithme de Dijkstra soit effectivement un $O((n+p)\log n)$, il convient de faire certains choix, en particulier pour les structures de données utilisées pour représenter le graphe. Nous aurons besoin d'accéder en coût constant à chaque sommet ainsi qu'à chacune des listes d'adjacence de ceux-ci, et c'est pourquoi nous allons utiliser le type *graphe* dont on rappelle la définition :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```

Chaque sommet est désormais identifié par un entier de $[\![0, n-1]\!]$.
On supposera en outre connue la matrice des poids w, de type *poids vect vect* [7].

Pour représenter l'ensemble $\overline{S}$, nous allons utiliser un tas-min modélisé par un vecteur **t** de taille n de type *int vect*. La case d'indice 0 contiendra l'indice du dernier élément du tas (ce dernier pourra donc contenir au maximum $n-1$ sommets, ce qui sera le cas au début de l'algorithme).
Nous aurons besoin de définir trois fonctions pour :

- ajouter un élément au tas ;
- extraire l'élément minimal ;
- modifier l'ordonnancement du tas lorsque le tableau **d** est modifié.

Cette dernière opération demande à être un peu détaillée [8]. En effet, lorsqu'une valeur du tableau **d** est modifiée, le sommet correspondant n'est plus nécessairement à la place adéquate dans le tas. Rechercher cet élément pour le déplacer induirait un coût linéaire qui réduirait à néant nos efforts pour utiliser un tas. Il est donc nécessaire de « marquer » l'emplacement de chaque sommet dans le tas de manière à pouvoir y accéder à coût constant. Ceci sera réalisé en utilisant un second tableau **m** de type *int vect* qui à chaque sommet du graphe associera son indice dans le tas. On veillera donc à maintenir l'invariant : **t.(m.(i)) = i**.
Pour ne pas alourdir inutilement l'exposé, les définitions relatives à la manipulation du tas **t** ont été placées en annexe de ce chapitre, page 27. On considère désormais définies les fonctions suivantes :

- **ajoute d t m**, de type *int –> unit*, qui ajoute un sommet au tas **t** ;
- **extrait d t m**, de type *unit –> int*, qui extrait du tas le sommet de distance minimale ;
- **remonte d t m**, de type *int –> unit*, qui reforme le tas à partir de l'indice passé en paramètre lorsque le tableau **d** est modifié [9].

L'algorithme s'écrit alors :

```
let dijkstra g w s =
  let n = vect_length g in
  let d = make_vect n Inf
  and t = make_vect n 0 and m = make_vect n 0 in
  for k = 0 to n-1 do
    d.(k) <- w.(s).(k) ;
    if k <> s then ajoute d t m k ;
  done ;
  let rec modif i = function
    | []   -> ()
    | j::q -> let x = som d.(i) w.(i).(j) in
              if inferieur x d.(j) then (d.(j) <- x ; remonte d t m m.(j)) ;
              modif i q
  in
  for k = 0 to n-1 do
    let i = extrait d t m in
    modif i g.(i)
  done ;
  d ;;
```

Illustrons cette fonction à l'aide de l'exemple qui nous a servi à décrire l'algorithme de Dijkstra :

7. Le type *poids* est défini page 13.
8. Relire la partie consacrée aux files de priorités dans le premier chapitre.
9. Puisque d_i ne peut que décroitre, le sommet i ne peut que remonter dans le tas.

```
# let g = [| [1; 2]; [3; 5]; [1; 4; 5]; []; [1; 3]; [4] |] ;;

# let w = [| [| P 0; P 7; P 1; Inf; Inf; Inf |];
             [| Inf; P 0; Inf; P 4; Inf; P 1 |];
             [| Inf; P 5; P 0; Inf; P 2; P 7 |];
             [| Inf; Inf; Inf; P 0; Inf; Inf |];
             [| Inf; P 2; Inf; P 5; P 0; Inf |];
             [| Inf; Inf; Inf; Inf; P 3; P 0 |] |] ;;

# dijkstra g w 0 ;;
- : poids vect = [|P 0; P 5; P 1; P 8; P 3; P 6|]
```

Détermination du chemin de poids minimal
Nous l'avons dit, il n'existe pas d'algorithme spécifique déterminant uniquement le chemin de poids minimal entre deux sommets a et b : on se contente d'appliquer l'algorithme de Dijkstra à partir de a. Tout au plus peut-on stopper la recherche dès lors que b rentre dans S.

Pour garder trace du chemin parcouru, il suffit d'utiliser un tableau c que l'on modifie en même temps que d : lorsque d_j est remplacé par $d_i + w(v_i, v_j)$, on mémorise i dans c_j. Ainsi, à la fin de l'algorithme c_j contient le sommet précédant v_j dans un chemin minimal allant de v_a à v_j, ce qui permet de reconstituer le chemin optimal une fois l'algorithme terminé.

Tout ceci conduit à la modification suivante :

```
let pluscourtchemin g w a b =
  let n = vect_length g in
  let d = make_vect n Inf and c = make_vect n 0
  and t = make_vect n 0 and m = make_vect n 0 in
  for k = 0 to n-1 do
    d.(k) <- w.(a).(k) ;
    if d.(k) <> Inf then c.(k) <- a ;
    if k <> a then ajoute d t m k ;
  done ;
  let rec modif i = function
    | []   -> ()
    | j::q -> let x = som d.(i) w.(i).(j) in
              if inferieur x d.(j) then
                (d.(j) <- x ; c.(j) <- i ; remonte d t m m.(j)) ;
              modif i q
  in
  let rec affiche_chemin l =
    if hd l = a then l else affiche_chemin (c.(hd l)::l)
  in
  let rec aux s =
    let i = extrait d t m in
    modif i g.(i) ;
    if i = b then affiche_chemin [b] else aux (i::s)
  in aux [a] ;;
```

Calculons par exemple le plus court chemin reliant les sommets 0 et 5 dans le graphe donné en exemple :

```
# pluscourtchemin g w 0 5 ;;
- : int list = [0; 2; 4; 1; 5]
```

4. Arbre couvrant minimal d'un graphe non orienté

Revenons un instant sur l'exemple du réseau maritime et ferroviaire représenté figure 8. 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 transits. De plus, elle souhaite que son réseau reste le plus performant possible, autrement dit que la somme des durées des liaisons subsistantes soit la plus petite possible : c'est le problème de la recherche de l'arbre couvrant de poids minimal (*minimal spanning tree*), dont une solution est présentée figure 10.

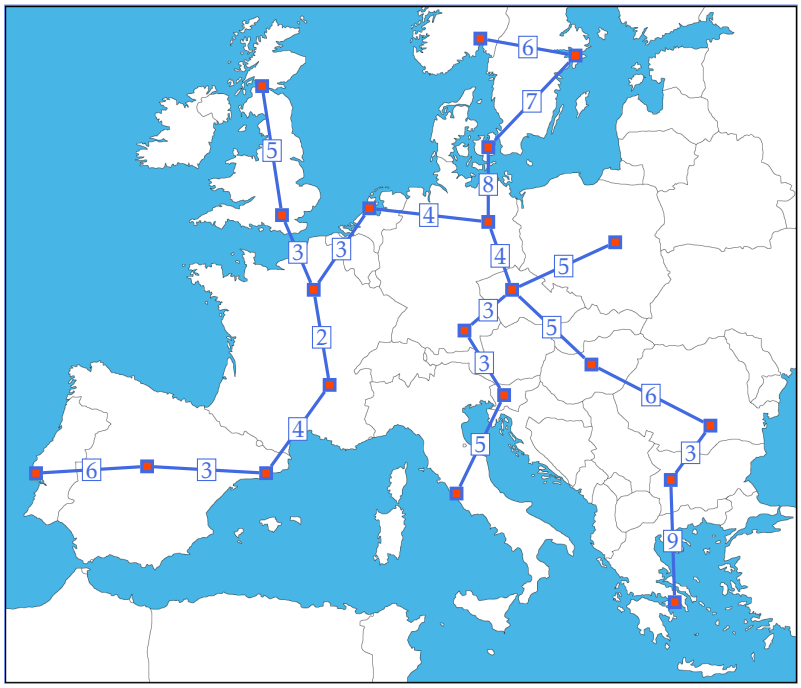


FIGURE 10 – L'arbre couvrant minimal du graphe de la figure 8.

Dans toute cette partie, $G = (V, E)$ est un graphe non orienté connexe muni d'une pondération $w : E \to \mathbb{R}$ à valeurs strictement positives.
Un *sous-graphe* $G' = (V', E')$ de G est un graphe tel que $V' \subset V$ et $E' \subset E$ muni de la pondération $w|_{E'}$. Il est dit *couvrant* lorsque il est lui-aussi connexe et lorsque $V' = V$. Le problème que nous allons étudier est la recherche d'un sous-graphe couvrant de poids minimal.
Mais tout d'abord, prouvons un résultat qui justifie le vocabulaire utilisé :

THÉORÈME. — *Si w est à valeurs dans $\mathbb{R}_+^*$, alors G possède un sous-graphe couvrant minimal, et ce dernier est un* arbre *(c'est-à-dire rappelons-le un graphe acyclique connexe).*

Preuve. L'ensemble des sous-graphes couvrants est non vide puisqu'il contient G, et il est fini, ce qui justifie l'existence d'un sous-graphe G' couvrant minimal.
Si jamais ce sous-graphe contenait un cycle, on pourrait supprimer une arête de ce cycle et le sous-graphe obtenu serait toujours couvrant et de poids strictement inférieur, ce qui est absurde ; G' est donc bien un arbre. □

Il existe deux algorithmes célèbres pour résoudre le problème de l'arbre couvrant de poids minimum. Chacun de ces deux algorithmes utilise plus particulièrement une des caractérisations des arbres que nous avons établies page 3 : le premier, l'algorithme de PRIM, utilise le fait qu'un arbre est un graphe connexe à $n-1$ arêtes ; le second, l'algorithme de KRUSKAL, qu'un arbre est un graphe acyclique à $n-1$ arêtes.

4.1 Algorithme de PRIM

L'idée sous-jacente de cet algorithme est de maintenir un sous-graphe partiel connexe $G' = (S, A)$, en connectant un nouveau sommet à chaque étape, jusqu'à ce que l'arbre couvre tous les sommets du graphe. Pour choisir ce sommet à connecter, on utilise une constatation simple : dans un arbre couvrant, il existe nécessairement une arête qui relie l'un des sommets de S avec un sommet en dehors de S. Pour construire un arbre couvrant minimal, il suffit de choisir parmi ces arêtes sortantes celle de poids le plus faible.

function PRIM(graphe connexe : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ un sommet choisi arbitrairement
  $A = \emptyset$
  while $S \neq V$ **do**
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ et $w(a, b)$ est minimal
    $S \leftarrow S \cup \{b\}$

$A \leftarrow A \cup \{(a,b)\}$

return (S, A)

Un exemple d'application de l'algorithme de Prim illustrant l'évolution de S est présenté figure 11.

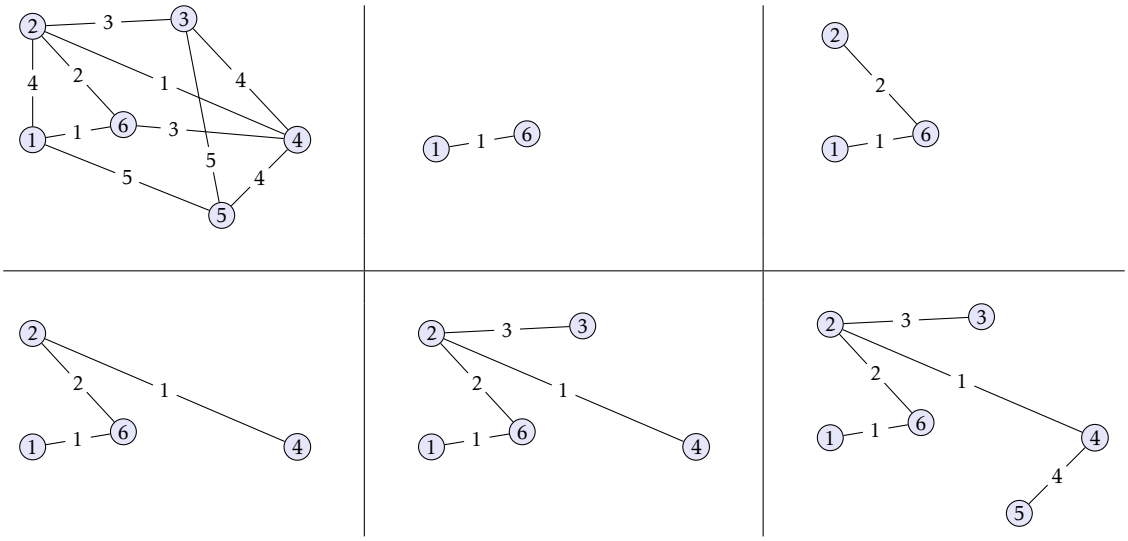


Figure 11 – Un exemple d'application de l'algorithme de Prim à partir du sommet 1.

Théorème. — *L'algorithme de* Prim *calcule un arbre couvrant de poids minimal.*

Preuve. Notons tout d'abord que le graphe construit par cet algorithme 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 prouver qu'il est de poids minimal.

Nous allons prouver par récurrence sur $|S|$ qu'à chaque étape de l'algorithme il existe un arbre couvrant de poids minimal qui contient le graphe (S, A), ce qui suffira à prouver le résultat souhaité.

- C'est bien évident lorsque $|S| = 1 : A = \emptyset$.
- Si $|S| \geqslant 1$, supposons l'existence d'un arbre couvrant T de poids minimal contenant l'arbre (S, A), et notons (a, b) l'arête que l'algorithme de Prim ajoute à A.

  Si cette arête appartient à T, nous en avons terminé. Dans le cas contraire, ajoutons cette arête à T. Les caractérisations des arbres que nous avons énoncées page 3 montrent que nous créons nécessairement un cycle. Ce cycle parcours à la fois des éléments de S (parmi eux, a) et des éléments de $V \setminus S$ (parmi eux, b). Il existe donc nécessairement une autre arête $(a', b') \neq (a, b)$ de ce cycle tel que $a' \in S$ et $b' \in V \setminus S$. Considérons alors l'arbre T′ obtenu en supprimant cette arête. Il s'agit de nouveau d'un arbre couvrant et il est de poids minimal car par choix de (a, b) on a $w(a, b) \leqslant w(a', b')$.

□

Étude de la complexité

Si p dénote 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 pré-traitement 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 celui-ci. Sachant que le coût du pré-traitement est un $O(p) = O(n^2)$, le coût total de l'algorithme est un $O(n^2)$.

Notons enfin que l'utilisation d'un tas pour stocker les différents sommets n'appartenant pas à S permet de réduire le coût, qui devient dès lors un $O(p \log n)$.

4.2 Algorithme de Kruskal

L'idée sous-jacente à cet algorithme est cette fois de maintenir un graphe partiel acyclique (autrement dit une forêt) jusqu'à ne plus obtenir qu'une seule composante connexe (un arbre). Pour ce faire, on débute avec le graphe à n sommets et aucune arête, et à chaque étape on ajoute l'arête de poids minimal qui permet de réunir deux composantes connexes distinctes.

function KRUSKAL(graphe : G = (V, E))
    A = ∅
    E_t = tri_croissant(E)
    for $(a, b) \in E_t$ **do**
        if A ∪ {(a, b)} est acyclique **then**
            A ← A ∪ {(a, b)}
    return (V, A)

Un exemple d'application de l'algorithme de Kruskal illustrant l'évolution de A est présenté figure 12

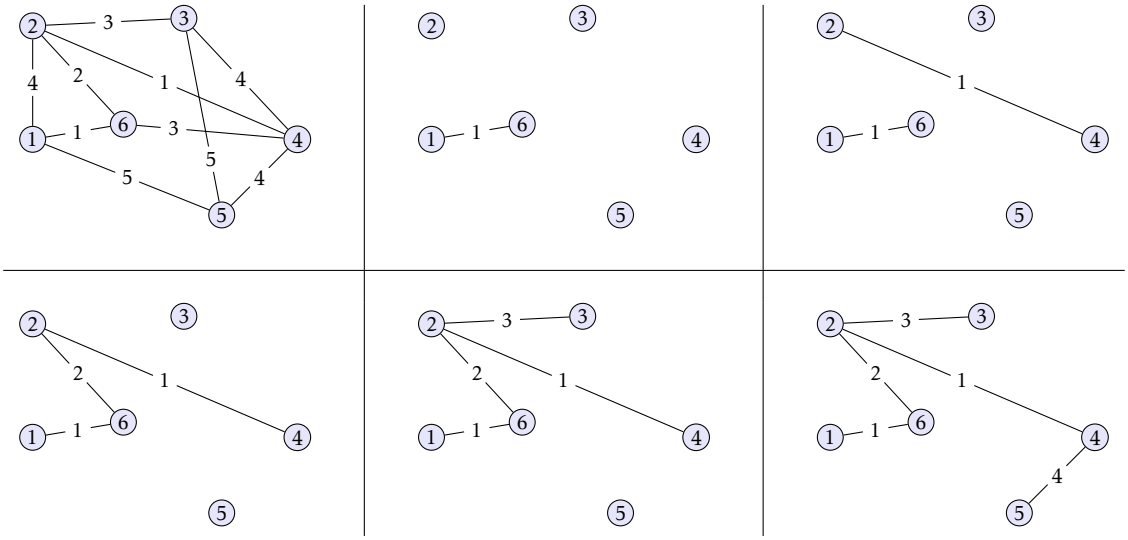


Figure 12 – Un exemple d'application de l'algorithme de Kruskal.

On peut noter qu'ainsi écrit, cet algorithme s'applique à un graphe non nécessairement connexe et retourne dans ce cas la forêt couvrante de poids minimal de G, c'est-à-dire une forêt dont chaque composante connexe est un arbre couvrant minimal d'une composante connexe de G. Si on applique cet algorithme à un graphe qu'on sait être connexe, on peut stopper cet algorithme dès lors que $|A| = |V| - 1$ pour obtenir l'arbre couvrant de poids minimal.

La preuve de validité de cet algorithme repose sur le résultat préliminaire suivant :

Lemme. — *Toutes les forêts couvrantes d'un graphe* G *ont même nombre d'arêtes.*

Preuve. Notons G = (V, E), et $C_1, \dots, C_p$ les composantes connexes de G. Le nombre d'arêtes d'une forêt couvrante de G est alors égale à : $\sum_{i=1}^{p} \big(|C_i| - 1\big) = |G| - p$. □

Nous sommes maintenant en mesure de prouver le :

Théorème. — *L'algorithme de* Kruskal *calcule une forêt couvrante de poids minimal.*

Preuve. Notons tout d'abord que le graphe ainsi construit est bien une forêt couvrante puisqu'on ajoute des arêtes sans jamais créer de cycle.

Il reste à établir que cette forêt est de poids minimal. Pour ce faire, on note $A = (e_1, \dots, e_k)$ les arêtes choisies par l'algorithme de Kruskal, rangées par ordre de poids croissant, et on considère une autre forêt couvrante $F = (V, A')$ dont les arêtes $A' = (e'_1, \dots, e'_k)$ sont elles aussi rangées par ordre de poids croissant. Nous allons montrer que pour tout $i \in [\![1, k]\!]$ on a $w(e_i) \leqslant w(e'_i)$, ce qui permettra de conclure.

Raisonnons par l'absurde en supposant qu'il existe un entier $i \in [\![1,k]\!]$ tel que $w(e'_i) < w(e_i)$, et considérons alors le graphe $G' = (V, E')$, avec $E' = \left\{e \in E \;\middle|\; w(e) \leqslant w(e'_i)\right\}$.
Appliqué à G', l'algorithme de Kruskal se déroule comme sur G et retourne un ensemble d'arêtes inclus dans $\{e_1, \ldots, e_{i-1}\}$, autrement dit une forêt couvrante de G' comportant *au plus* $i-1$ arêtes. Or la forêt $F' = (V, A'')$ avec $A'' = (e'_1, \ldots, e'_i)$ est une forêt couvrante de G' qui comporte i arêtes, ce qui contredit le résultat du lemme précédent. □

Étude de la complexité
L'algorithme de Kruskal consiste avant tout à trier les arêtes puis à les énumérer par ordre croissant ; clairement l'usage d'un tas s'impose. Si on note p le nombre d'arêtes de G, le coût de la formation du tas est un $O(p)$. Par ailleurs, il existe des structures de données qui permettent de gérer efficacement une partition d'objets et qui permettent ici de représenter l'évolution des différentes composantes connexes [10]. Si on utilise une telle structure, le coût total de l'algorithme de Kruskal devient un $O(p \log p)$. Enfin, sachant que $p = O(n^2)$ on peut simplifier le coût en $O(p \log n)$ et retrouver ainsi le même coût que l'algorithme de Prim.

5. Exercices

5.1 Combinatoire des graphes

Exercice 1

a) Montrer qu'un graphe simple non orienté a un nombre pair de sommets de degré impair.

b) Montrer que dans une assemblée de n personnes ($n \geqslant 2$), il y a toujours au moins deux personnes qui ont le même nombre d'amis présents (on considère que la relation d'amitié est symétrique).

c) Est-il possible de créer un réseau de 15 ordinateurs de sorte que chaque machine soit reliée avec exactement trois autres ?

Exercice 2 Une suite finie décroissante (au sens large) est dite *graphique* s'il existe un graphe simple dont les degrés des sommets correspondent à cette suite.

a) Les suites suivantes sont-elles graphiques ?

$$(3,3,2,2), \quad (3,3,1,1), \quad (3,3,2,1,1)$$

b) Trouver deux graphes différents correspondants à la suite $(3,2,2,2,1)$.

c) Soit $n \geqslant 2$ et $(d_1, d_2, \ldots, d_n)$ une suite décroissante. Montrer l'équivalence des deux propriétés suivantes :

(i) la suite $(d_1, d_2, \ldots, d_n)$ est graphique ;

(ii) la suite $(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \ldots, d_n)$ est graphique.

Ce résultat constitue le théorème de Havel et Hakimi.

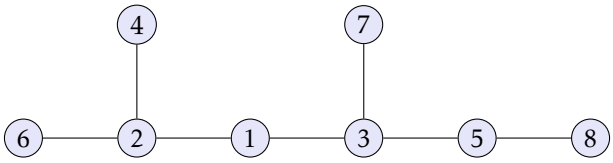
d) Déduire de la preuve de ce résultat un graphe correspondant à la suite $(4,4,3,2,2,1)$. Dans ce graphe, les deux sommets de degré 2 peuvent-ils être voisins ?

Exercice 3 Considérons un arbre $G = (V, E)$ avec $V = \{1, 2, \ldots, n\}$ et $n \geqslant 2$. Le codage de Prüfer permet de décrire précisément cet arbre à l'aide d'une suite finie de $n-2$ entiers de l'intervalle $[\![1, n]\!]$. Il se déroule de la façon suivante :

function prufer(arbre : G = (V, E))
  L = ∅
  while |V| > 2 **do**
    $i \leftarrow \min\left\{j \in V \;\middle|\; \deg(j) = 1\right\}$
    Soit $j \in V \;\middle|\; (i,j) \in E$
    L ← L ∪ {j}
    V ← V \ {i}, E ← E \ {(i, j)}
  return L

a) Déterminer le codage de Prüfer de l'arbre ci-dessous :

10. Cette structure de données s'appelle *Union-Find*.



b) Proposer un algorithme de décodage du codage de Prüfer.

c) À quel arbre correspond le code $(2,2,1,3,3,1,4,4)$?

Remarque. Le codage de Prüfer réalise une bijection entre les arbres étiquetés par $[\![1,n]\!]$ et $[\![1,n]\!]^{n-2}$, ce qui constitue une preuve de la formule de Cayley : le nombre d'arbres distincts que l'on peut construire avec n sommets est égal à n^{n-2}.

5.2 Représentation d'un graphe

Exercice 4 On considère un graphe orienté représenté par listes d'adjacence à l'aide de type *graphe*. Rédiger deux fonctions Caml permettant de calculer le vecteur contenant les degrés sortants de chacun des sommets, puis celui contenant les degrés entrants. Analysez le coût de ces algorithmes.

Exercice 5 La *transposée* d'un graphe orienté $G = (V, E)$ est le graphe $G^T = (S, E^T)$, où $E^T = \left\{(a,b) \in S \times S \;\middle|\; (b,a) \in E\right\}$ (autrement dit, tous les arcs sont inversés).
Décrire des algorithmes efficaces permettant de calculer G^T lorsque G est représenté par une matrice d'adjacence, puis lorsque G est représenté par listes d'adjacence.
Analysez le temps d'exécution de vos algorithmes, et rédigez celui correspondant au type *graphe*.

Exercice 6 Soit M la matrice d'adjacence d'un graphe orienté G d'ordre n, et $p \in \mathbb{N}$.

a) Que représente le coefficient d'indice (i,j) de la matrice M^p ? Et la trace de cette matrice ?

b) Montrer que G contient un cycle si et seulement si la matrice M^n n'est pas nulle. Quel serait le coût d'un algorithme qui utiliserait ce critère pour déterminer l'existence d'un cycle dans un graphe ?

5.3 Parcours dans un graphe

Exercice 7 Soit $G = (V, E)$ un graphe orienté. Montrer qu'il n'y a pas de circuit dans G si et seulement s'il est possible d'attribuer un nombre $r(v)$ (appelé le *rang* de v) à tout sommet de G de telle sorte que pour tout arc $(u,v) \in E$ on ait $r(u) < r(v)$.
Rédiger en pseudo-code un algorithme qui attribue un rang à tout sommet d'un graphe orienté sans circuit, et évaluer son coût pour un graphe défini par une matrice d'adjacence.

Exercice 8 On considère un graphe non orienté $G = (V, E)$. Pour tout $v \in V$ on note $\mathscr{V}(v)$ l'ensemble de ses voisins. On lui applique la fonction suivante :

```
function alpha(graphe G = (V, E))
    if E = ∅ then
        return |V|
    else
        Soit v ∈ V | deg(v) ⩾ 1
        p ← alpha(G \ {v})
        q ← alpha(G \ ({v} ∪ 𝒱(v)))
        return max(p, q + 1)
```

(par abus de notation, on note $G \setminus \{v\}$ le graphe obtenu en supprimant v de V ainsi que toutes les arêtes qui le concernent dans E).

a) Que retourne la fonction α lorsque :

- G est un graphe sans arête ?
- G est un graphe complet ?
- G est un chemin non ramifié ?
- G est un cycle ?

b) On appelle *stable* un sous-ensemble de V dans lequel ne figure aucun couple de voisins. Prouver avec soin que la fonction α calcule le cardinal maximal d'un stable de G (le *nombre de stabilité* de G).

c) Quel est le coût de cet algorithme ?

Exercice 9 On appelle *diamètre* d'un graphe non orienté la longueur du plus long chemin acyclique qu'il est possible de tracer.

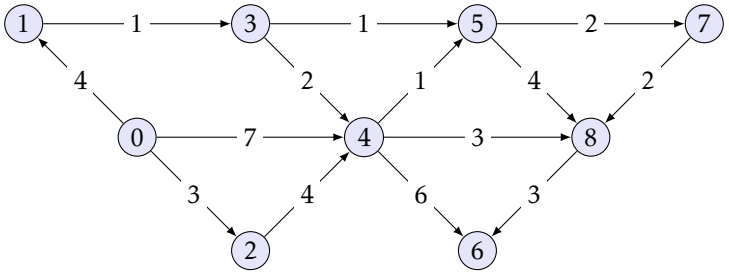
a) Proposer un algorithme naïf pour calculer le diamètre d'un graphe, et analyser son temps d'exécution.

b) Proposer un algorithme efficace pour calculer le diamètre d'un arbre, et analyser son temps d'exécution.

5.4 Plus court chemin

Exercice 10 Comment utiliser l'algorithme de Floyd-Warshall pour déterminer l'existence d'un cycle de poids strictement négatif ?

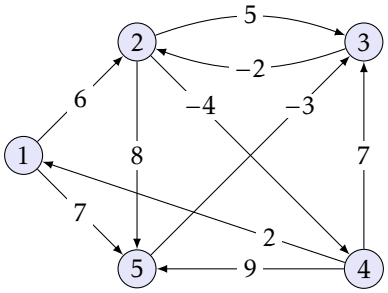
Exercice 11 Dans le graphe suivant, il existe de nombreux chemins de poids minimal menant du sommet 0 au sommet 8, mais lequel est retourné par l'algorithme de Dijkstra ? (on conviendra qu'à chaque itération, le chemin optimal menant à un sommet v n'est mis à jour qu'en cas de découverte d'un chemin de poids strictement inférieur).



Exercice 12 L'algorithme de Bellman-Ford permet de déterminer le plus court chemin à partir d'une source dans un graphe $G = (V, E)$ pondéré par une fonction $w : E \to \mathbb{R}$ pouvant prendre des valeurs négatives. Il se déroule de la façon suivante :

function Bellman-Ford(sommet : s)
  for all $v \in V \setminus \{s\}$ **do**
    $d_v \leftarrow +\infty$
  $d_s \leftarrow 0$
  for $k \in [\![1, |V| - 1]\!]$ **do**
    for all $(u, v) \in E$ **do**
      $d_v \leftarrow \min\big(d_v, d_u + w(u, v)\big)$
  for all $(u, v) \in E$ **do**
    if $d_u + w(u, v) < d_v$ **then**
      return Faux
  return Vrai

a) Appliquer cet algorithme au graphe présenté ci-dessous, à partir du sommet $s = 1$.



Que ce passerait-t'il si l'arc $4 \to 5$ était de poids 8 et non pas de poids 9 ?

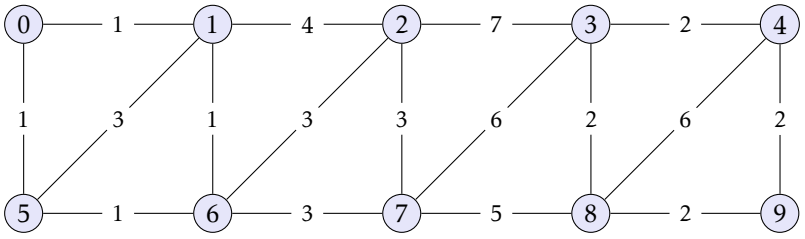
b) Montrer que si le graphe ne présente pas de cycle de poids négatif, alors à la fin de l'algorithme on a $d_v = \delta(s, v)$ pour tout $v \in V$ et la valeur retournée est « Vrai ».

c) Montrer que si le graphe possède un cycle de poids strictement négatif la valeur retournée est « Faux ».

d) Faire une analyse du coût de cet algorithme.

5.5 Arbre couvrant minimal

Exercice 13 Appliquer les algorithmes de Prim et de Kruskal au graphe ci-dessous.



Exercice 14 Soit $G = (V, E)$ un graphe non orienté connexe, et A un ensemble d'arêtes acyclique. Montrer l'existence d'un arbre couvrant contenant toutes les arêtes de A.

Exercice 15 Soit $G = (V, E)$ un graphe non orienté connexe, et $w : E \to \mathbb{R}_+$ une pondération des arêtes. On considère deux arbres couvrants minimaux distincts (V, A) et (V, A'), et on choisit une arête e de poids minimal dans $A \triangle A' = (A \setminus A') \cup (A' \setminus A)$. On suppose par exemple $e \in A \setminus A'$.
Montrer qu'il existe une arête $e' \in A' \setminus A$ telle que $w(e') = w(e)$, et en déduire que si w est injective il existe un unique arbre couvrant minimal.

Exercice 16 Soit $G = (V, E)$ un graphe non orienté connexe muni d'une pondération injective $w : E \to \mathbb{R}$. L'exercice précédent a montré l'unicité de l'arbre couvrant minimum.

a) Montrer qu'en revanche, le second arbre couvrant par ordre croissant de poids n'est pas nécessairement unique (donner un exemple).

b) Soit (V, A) l'arbre couvrant minimum, et (V, B) un arbre couvrant second par ordre croissant de poids. Montrer qu'il existe deux arêtes $e \in A$ et $e' \notin A$ tel que $B = A \setminus \{e\} \cup \{e'\}$.

c) Soit (V, B) un arbre couvrant quelconque. Pour tout $(u, v) \in V^2$ on note $\max_B(u, v)$ une arête de poids maximal sur le chemin (unique) qui relie u et v dans (V, B).
Décrire un algorithme qui, à partir de B, calcule toutes valeurs $\max_B(u, v)$ pour un temps total en $O(n^2)$, avec $n = |V|$.

d) En déduire un algorithme efficace permettant de calculer un arbre couvrant second par ordre croissant de poids.

Exercice 17 On considère un ensemble de villes $V = \{v_1, v_2, \ldots, v_n\}$ ainsi que les distances $d(v_i, v_j)$ séparant ces villes. On rappelle qu'une distance vérifie l'inégalité triangulaire :

$$\forall (a, b, c) \in V^3, \quad d(a, c) \leqslant d(a, b) + d(b, c).$$

On note G le graphe pondéré complet associé.

Le problème du voyageur de commerce est de passer par chacune des villes puis de revenir au point de départ, et ce en minimisant la distance parcourue.
En d'autres termes, il s'agit de déterminer un *cycle couvrant minimal*, c'est à dire un cycle $C_m = (x_1, x_2, \ldots, x_n)$ qui passe par toutes les villes de V et qui minimise la somme $d(C_m) = d(x_1, x_2) + d(x_2, x_3) + \cdots + d(x_{n-1}, x_n) + d(x_n, x_1)$.
On observera que l'inégalité triangulaire montre qu'une solution minimale passe une et une seule fois par chaque ville (et donc que $|C_m| = n$).

a) Soit A un arbre inclus dans G, et C le cycle décrit par un parcours en profondeur de A.
Montrer que $d(C) \leqslant 2d(A)$, où $d(A)$ désigne la somme de toutes les distances des arêtes de A.

b) En déduire que si C est un cycle associé à un arbre couvrant minimal alors $d(C) \leqslant 2d(C_m)$.

Annexe : utilisation d'un tas pour l'algorithme de Dijkstra

Nous devons définir les fonctions nécessaires à la manipulation d'un tas **t** de type *int vect* trié par ordre de priorités croissantes, celles-ci étant stockées dans la tableau **d** de type *poids vect*.
Nous devons en outre tenir à jour un tableau **m** de type *int vect* qui marque l'emplacement de chaque sommet dans le tas, autrement dit qui vérifie l'invariant : **t.(m.(i)) = i**.
La permutation de deux éléments de ces tableaux sera effectuée par la fonction :

```
let swap v i j =
  let temp = v.(i) in v.(i) <- v.(j) ; v.(j) <- temp ;;
```

Commençons par les deux fonctions qui permettent de remonter un élément dans le tas, ou au contraire de le descendre.
Le père de l'élément d'indice $k > 1$ a pour indice $\left\lfloor \frac{k}{2} \right\rfloor$, ce qui conduit à la fonction :

```
let rec remonte d t m = function
  | 1                                        -> ()
  | k when inferieur d.(t.(k)) d.(t.(k/2)) -> swap m t.(k) t.(k/2) ; swap t k (k/2) ;
                                                 remonte d t m (k/2)
  | _                                        -> () ;;
```

À l'inverse, les deux fils (s'ils existent) de l'élément d'indice k ont pour indices $2k$ et $2k+1$. Sachant qu'on stocke dans **t.(0)** l'indice du dernier élément du tas, ceci conduit à la définition :

```
let rec descend d t m = function
  | k when 2*k > t.(0) -> ()
  | k -> let j = if t.(0) = 2*k || inferieur d.(t.(2*k)) d.(t.(2*k+1))
                  then 2*k else 2*k+1 in
         if inferieur d.(t.(j)) d.(t.(k)) then
            (swap m t.(k) t.(j) ; swap t k j ; descend d t m j) ;;
```

Pour insérer un élément dans un tas, on le place en dernière position et on le remonte :

```
let ajoute d t m k =
  t.(0) <- t.(0) + 1 ;
  t.(t.(0)) <- k ;
  m.(k) <- t.(0) ;
  remonte d t m t.(0) ;;
```

Enfin, pour extraire l'élément minimal d'un tas (qui se trouve nécessairement à la racine) on le permute avec le dernier et on fait descendre celui-ci :

```
let extrait d t m =
  let k = t.(1) in
  swap m t.(1) t.(t.(0)) ;
  swap t 1 t.(0) ;
  t.(0) <- t.(0) - 1 ;
  descend d t m 1 ;
  k ;;
```

Reportez-vous à la partie consacrée aux tas binaires dans le premier chapitre de ce cours pour une meilleure compréhension de ces fonctions.