Chapitre 4

Algorithmes sur les graphes

1 Parcours d'un graphe

1.1 Principe

L'idée générale du parcours d'un graphe est de construire une énumération (injective) des sommets accessibles depuis un sommet donné. Tout nouveau sommet rencontré est "traité" en fonction du problème auquel répond le parcours et on ajoute à l'ensemble des sommets à traiter les voisins de ce sommet qui n'ont pas encore été traités.

De manière générale, le schéma d'un parcours est le suivant :

Entrée : Graphe $G = (S, A)$ et sommet x_0
Début algorithme
    Ajouter x_0 aux sommets à traiter et aux sommets visités
    Tant que l'ensemble des sommets à traiter est non vide **faire**
        Extraire x des sommets à traiter
        Traiter x
        Pour tous les sommets y voisins de x **faire**
            Si y n'est pas visité **alors**
                Ajouter y aux sommets à traiter et aux sommets visités

Les différences entre les parcours viennent de la manière dont sont ajoutés et extraits les sommets, donc de la structure de l'ensemble des sommets à traiter.

Arborescence associée à un parcours.
A chaque parcours débutant par un sommet x_0 peut être associé un arbre enraciné en x_0 : on débute avec le graphe $(\{x_0\}, \emptyset)$ puis à chaque ajout d'un nouveau sommet y dans l'ensemble des sommets à traiter de l'algorithme ci-dessus, on ajoute le sommet y et l'arête (x, y). 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 l'ensemble des sommets à traiter, et n'en sort donc aussi qu'au plus une fois. Si ces opérations d'entrée et de sortie sont de coût constant (ce qui sera effectivement le cas dans la suite), le coût total des manipulations de l'ensemble de sommets à traiter est un $O(n)$, avec $n = |S|$. 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 $p = |A|$, 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 $O(n + p)$.

Remarques.

1. Dans ce qui suit, nous considérons un graphe défini par liste d'adjacence avec le type `graphe2`.
2. Pour déterminer si un sommet a déjà été vu en coût constant, la solution que nous adopterons consistera à utiliser une structure de tableau de booléens `t`. Ainsi, si le sommet `i` a déjà été visité, alors `t.(i)` vaudra `true` (`false` sinon).

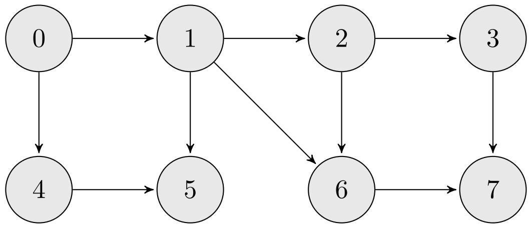
1.2 Parcours en profondeur

L'algorithme de parcours en profondeur (DFS pour "Depth First Search") d'un graphe utilise une structure de pile pour l'ensemble des sommets à traiter.

Ainsi, l'algorithme cherche à aller le plus loin possible dans le graphe avant de rebrousser chemin et d'explorer les autres sommets. On traite la liste des voisins non visités avant de traiter le reste.

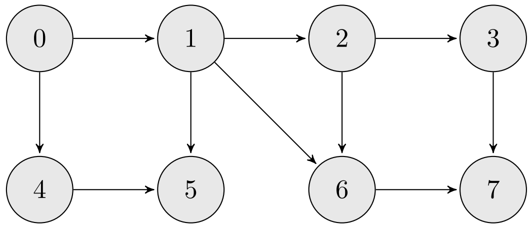
Implémentation.

Exemple. Illustrons cet algorithme sur un exemple :



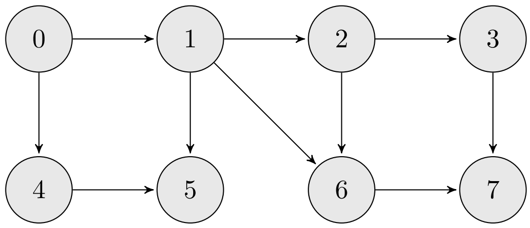
Visités :

A traiter :



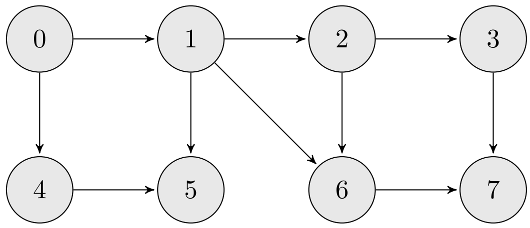
Visités :

A traiter :



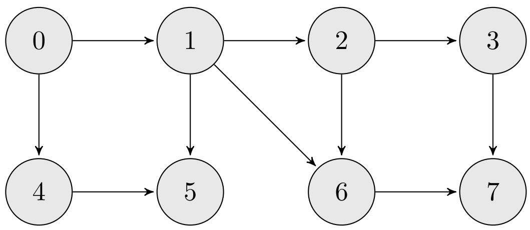
Visités :

A traiter :



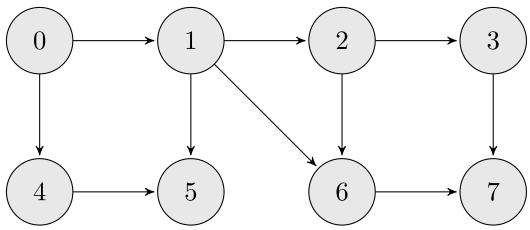
Visités :

A traiter :



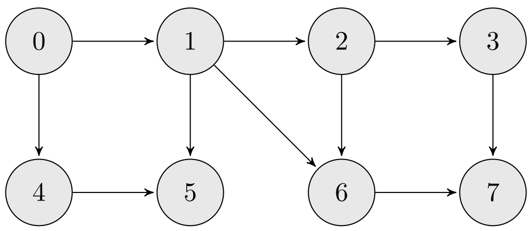
Visités :

A traiter :



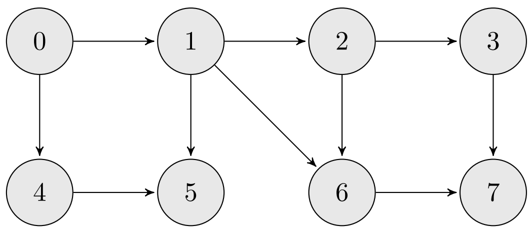
Visités :

A traiter :



Visités :

A traiter :



Visités :

A traiter :

Ainsi, le parcours en profondeur de ce graphe traite les sommets dans l'ordre $0, 1, 2, 3, 7, 5, 6, 4$. On donne l'arborescence associée à ce parcours :

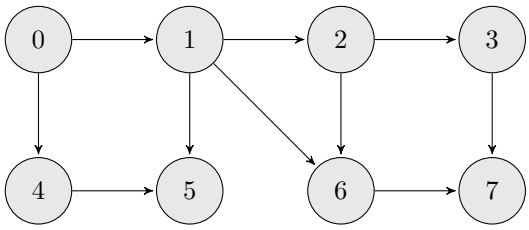
1.3 Parcours en largeur

L'algorithme de parcours en largeur (BFS pour "Breadth First Search") d'un graphe ressemble beaucoup au précédent. Ici, l'ensemble des sommets à traiter utilise une structure de file. Ainsi, on visite d'abord les voisins les plus proches avant d'aller visiter en profondeur le graphe.

Implémentation.

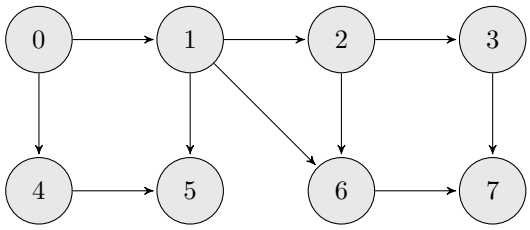
Remarque. La seule différence avec l'algorithme de parcours en profondeur précédent est l'ordre de la concaténation entre la liste des sommets à traiter et les voisins non visités.

Exemple. Illustrons cet algorithme sur un exemple :



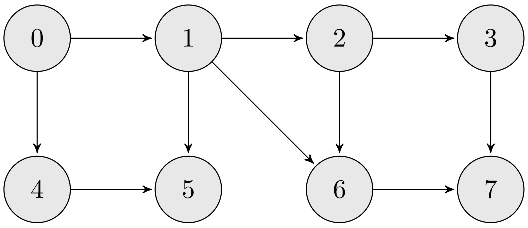
Visités :

A traiter :



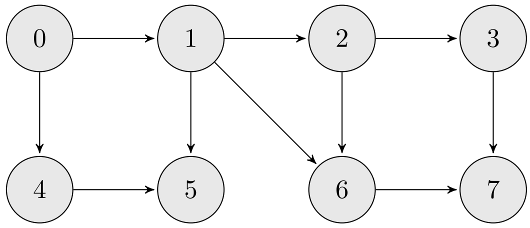
Visités :

A traiter :



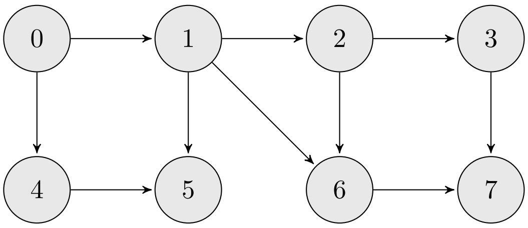
Visités :

A traiter :



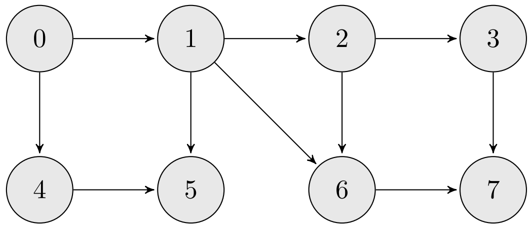
Visités :

A traiter :



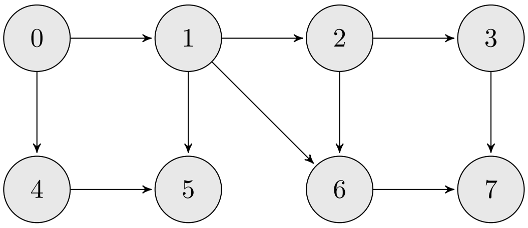
Visités :

A traiter :



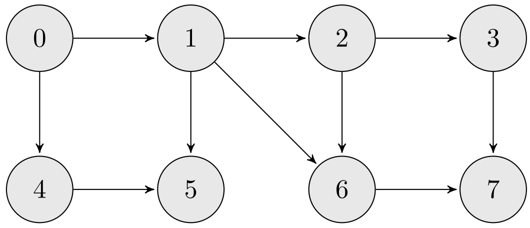
Visités :

A traiter :



Visités :

A traiter :



Visités :

A traiter :

Ainsi, le parcours en largeur de ce graphe traite les sommets dans l'ordre $0, 1, 4, 2, 5, 6, 3, 7$. On donne l'arborescence associée à ce parcours :

Remarque. Lors du parcours en largeur, on traite 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. Il suffit pour cela de maintenir à jour un tableau de prédécesseurs : chaque fois qu'un sommet est ajouté à l'ensemble des sommets à traiter, on indique que son prédécesseur est le sommet en cours de traitement. On retiendra la :

Propriété 1 (Plus courts chemins d'un sommet à tous les sommets accessibles)

Soit $G = (S, A)$ un graphe (orienté ou non) et $x_0 \in S$.
Le parcours en largeur de G depuis x_0 renvoie les plus courts chemins de x_0 à tous les sommets accessibles.

1.4 Applications

Composantes connexes d'un graphe non orienté

Pour tester si un graphe non orienté est connexe, il suffit de lancer un parcours (en largeur ou en profondeur) et de vérifier si l'on a bien traité tous les sommets.

Pour déterminer les composantes connexes d'un graphe non orienté, il suffit d'effectuer un premier parcours, d'enlever les sommets déjà énumérés (et donc les arêtes associées) puis de recommencer à partir d'un autre sommet tant qu'il en reste.

Implémentation.

Recherche d'un cycle

Pour déterminer si un graphe non orienté connexe admet un cycle, on va parcourir le graphe en profondeur. Il admet un cycle si et seulement si au moment d'ajouter un voisin, on retombe sur un sommet déjà traité, autre que son prédécesseur. On garde en mémoire pour cela un tableau de prédécesseurs, mis à jour quand un voisin est ajouté.

Implémentation.

Tri topologique

Définition.

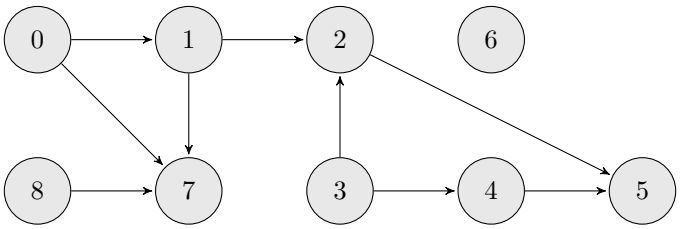
Un tri topologique d'un graphe orienté acyclique $G = (S, A)$ est un ordre linéaire des sommets de G de sorte que, si $(x, y) \in A$ est un arc de G, alors x apparaît avant y dans le tri.

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.

Remarques.

1. Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale tel que tous les arcs du graphe soient orientés de gauche à droite.
2. Le tri topologique d'un graphe orienté acyclique n'est pas forcément unique.

Exercice. Proposer plusieurs tris topologiques du graphe orienté acyclique suivant :



L'hypothèse d'acyclicité est essentielle :

Propriété 2 (CNS d'existence d'un tri topologique)

Un graphe $G = (S, A)$ orienté admet un tri topologique si et seulement s'il est acyclique.

Preuve.

□

Implémentation. L'algorithme de tri topologique se déduit directement du parcours en profondeur : la liste des sommets étudiés dans l'ordre d'un parcours en profondeur d'un graphe $G = (S, A)$ nous construit un tri topologique des sommets du graphe G.

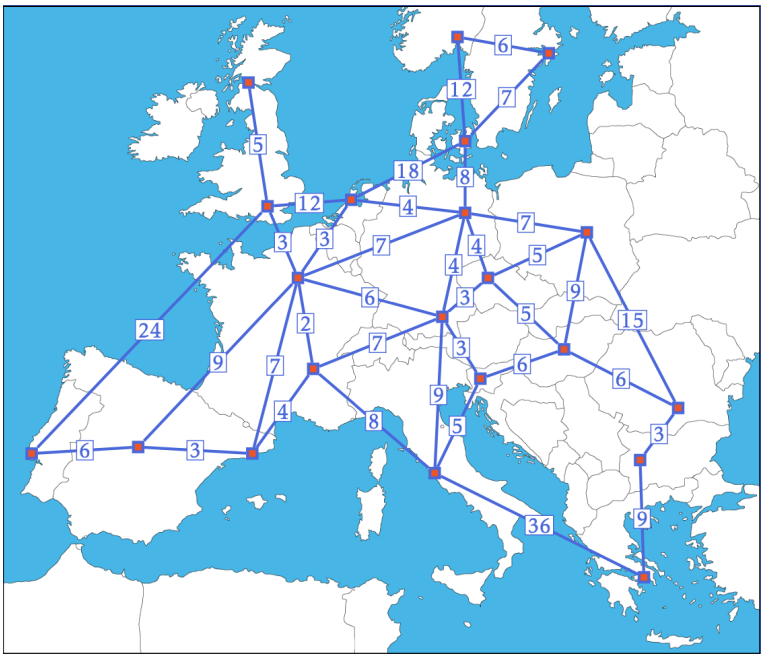
En effet, pour tous sommets x et y du graphe $G = (S, A)$ tels que l'arc $(x, y) \in A$, l'étude du sommet x se terminera après l'étude du sommet y.

2 Plus court chemin

2.1 Graphes pondérés

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 ci-dessous 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.



Commençons par donner quelques définitions :

Définition.

- Un **graphe pondéré** (orienté ou non) est un triplet $G = (S, A, w)$ tel que (S, A) est un graphe et w est une application de A dans $\mathbb{R}_+^*$ appelée **pondération**.
- Pour tout $(a, b) \in A$, on dit que $w(a, b)$ est le **poids** de l'arête (a, b).

Remarque. Il sera par la suite commode de prolonger la définition de w sur $S \times S$ en posant :

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

Définition.

Soit $G = (S, A, w)$ un graphe pondéré.

- Le **poids** d'un chemin est la somme des poids des arêtes qui le composent.
- La **distance** $\delta(a, b)$ entre deux sommets a et b de G est le poids minimal d'un chemin entre a et b s'il en existe. Dans le cas contraire, on posera $\delta(a, b) = +\infty$.
- Un **plus court chemin** entre deux sommets a et b de G est, s'il en existe, un chemin de poids $\delta(a, b)$.

Il existe trois problèmes de plus courts chemins :

1. Calculer le chemin de poids minimal entre une source a et une destination b ;
2. Calculer les chemins de poids minimal entre une source a et tout autre sommet du graphe ;
3. 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 :

Propriété 3 (Principe de sous-optimalité)

Soit $G = (S, A, w)$ un graphe pondéré et a, b, c trois sommets de G.
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.

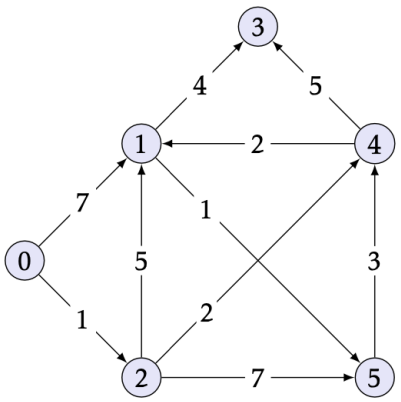
Preuve. S'il existait un chemin plus court entre par exemple a et c, il suffirait de le suivre lors du trajet entre a et b pour contredire le caractère minimal du trajet $a \rightsquigarrow b$. □

2.2 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,

$$\forall (i, j) \in \mathbb{N}^2, \quad m_{i,j} = w(s_i, s_j).$$

Exercice. Considérons le graphe G suivant :



Donner la matrice d'adjacence M associée.

Étant donné un graphe G, l'algorithme de Floyd-Warshall consiste à calculer la suite finie de matrices $M^{(k)}$, $0 \leq k \leq n$, avec $M^{(0)} = M$ et :

$$\forall k < n,\ \forall (i,j) \in [\![1,n]\!]^2, \quad m_{i,j}^{(k+1)} = \min\left(m_{i,j}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Théorème 4 (Algorithme de Floyd-Warshall)

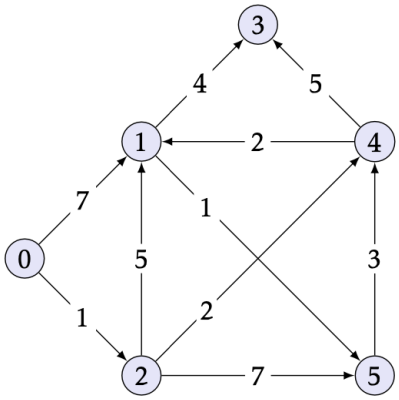
Pour tout $(i,j) \in [\![1,n]\!]^2$, $m_{i,j}^{(k)}$ est égal au poids d'un plus court chemin reliant s_i à s_j et ne passant que par des sommets de la liste $s_1, s_2, \ldots, s_k$.

En particulier, $m_{i,j}^{(n)}$ est égal à la distance $\delta(s_i, s_j)$ entre s_i et s_j.

Preuve.

□

Exercice. Reprenons le graphe donné dans l'exemple précédent :



Expliciter les matrices successives données par l'algorithme de Floyd-Warshall :

Implémentation. Nous allons définir le type :

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

ce qui nous permet de représenter un graphe pondéré par le type `poids array array`. On définit ainsi le graphe G de l'exemple précédent par :

On définit la somme et le minimum de deux objets de type `poids` :

On peut alors donner l'algorithme de Floyd-Warshall (en utilisant une seule matrice dont on modifie les coefficients) :

Remarque. De manière évidente, la complexité temporelle de l'algorithme de Floyd-Warshall est en $O(n^3)$, 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. On peut les stocker dans une matrice annexe :

2.3 Algorithme de Dijkstra

Intéressons nous maintenant au problème des plus courts chemins à partir d'une même source $x_0 \in S$. A partir de x_0, l'algorithme de Dijkstra va progressivement remplir un tableau d de longueur n de sorte qu'à la fin de cet algorithme d_x soit égal à $\delta(x_0, x)$, le poids du plus court chemin entre x_0 et x.

Pour ce faire, on fait évoluer une partition de S, X initialisé à $\{x_0\}$ et son complémentaire $\overline{X}$. L'ensemble X est destiné à représenter les sommets dont on a déterminé dans le tableau d le poids du chemin minimal à partir de x_0. Pour les éléments de $\overline{X}$, le tableau d contiendra le poids du plus court chemin ne passant que par des sommets appartenant à X. À chaque itération on choisit le sommet de $\overline{X}$ dont la valeur associée dans le tableau d est minimale pour le transférer dans X, et on modifie le tableau d en conséquence. Ainsi, chaque élément de $\overline{X}$ va progressivement être transféré dans X.

L'algorithme se déroule de la façon suivante :

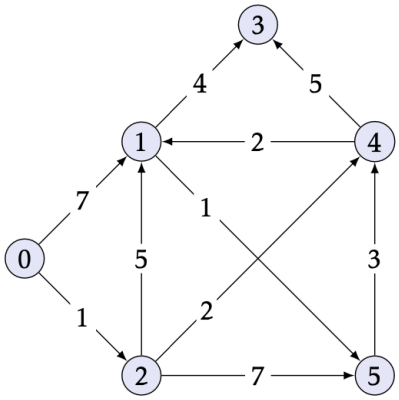
Entrée : Graphe $G = (S, A, w)$ et sommet x_0
Début algorithme
  Pour tout sommet $x \in S$ **faire**
    $d_x \leftarrow w(x_0, x)$
  Tant que l'ensemble $\overline{X}$ est non vide **faire**
    Extraire x de $\overline{X}$ tel que d_x est minimal
    Ajouter x à l'ensemble X
    Pour tous les sommets $y \in \overline{X}$ **faire**
      $d_y \leftarrow \min(d_y, d_x + w(x, y))$

Théorème 5 (Algorithme de Dijkstra)

À l'issue de l'algorithme de Dijkstra on a :

$$\forall x \in S, \quad d_x = \delta(x_0, x).$$

Exercice. Appliquer l'algorithme de Dijkstra au graphe suivant, à partir du sommet 0 :



Remarques.

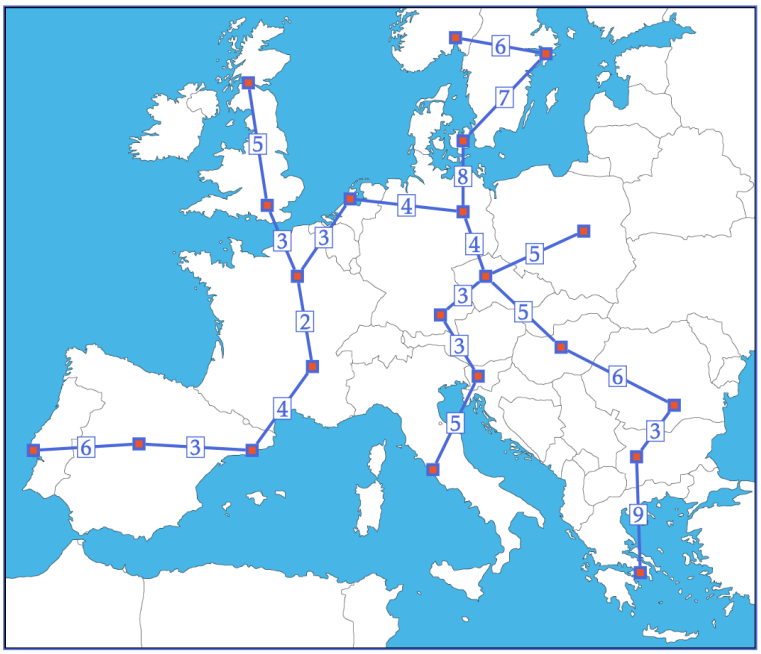
1. Dans l'algorithme proposé, un grand flou est laissé sur la structure de données utilisée pour le graphe et pour les ensembles des sommets. Ce choix change radicalement la complexité de l'algorithme. Étant donnée la façon de choisir l'élément x à extraire, il est naturel de vouloir doter cet ensemble d'une structure de file de priorité et donc de l'implémenter par un tas (c'est-à-dire un tas-min puisqu'on cherche à extraire l'élément avec la plus petite distance).
2. Donnons une estimation de la complexité pour un graphe à n sommets et p arcs :
   - Chaque extraction d'un nouveau sommet dans le tas est de complexité $O(\ln(n))$.
   - Lors du traitement d'un sommet x, chaque mise à jour d'une valeur du tableau d a lui aussi un coût en $O(\ln(n))$. Comme chaque arête ne va intervenir qu'au plus une fois dans les modifications de d, le coût total de ces modifications est un $O(p\ln(n))$.

   En sommant sur tous les sommets, on obtient une complexité $O((n+p)\ln(n))$ donc $O(p\ln(n))$ (pour un graphe connexe).
3. Comme pour l'algorithme de Floyd-Warshall, il est possible de retrouver les chemins permettant d'atteindre ces distances, en maintenant à jour un tableau de prédécesseurs à chaque calcul de minimum.

3 Arbre couvrant minimal d'un graphe non orienté

3.1 Définitions et première propriété

Revenons sur l'exemple du réseau maritime et ferroviaire présenté précédement. 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, dont une solution est présentée ci-dessous :



Définition.

Soit $G = (S, A, w)$ un graphe non orienté connexe.

- Un **sous-graphe** de G est un graphe $G' = (S', A')$ de $G = (S, A)$ muni de la pondération $w_{|A'}$.
- Le poids d'un sous-graphe G' de G est la somme notée $w(G')$ des poids de ces arêtes.
- Un sous-graphe G' de G est dit **couvrant** si G' est connexe et si $S' = S$.

Le problème que nous allons étudier est la recherche d'un sous-graphe couvrant de poids minimal. Le résultat suivant justifie le vocabulaire utilisé :

Théorème 6 (Arbre couvrant minimal d'un graphe non orienté)

Tout graphe $G = (S, A, w)$ non orienté connexe possède un sous-graphe couvrant minimal, et ce sous-graphe est un arbre.

Preuve.

□

Il existe deux algorithmes gloutons pour résoudre ce problème :

- l'algorithme de Kruskal : il utilise le fait qu'un arbre est un graphe acyclique à $n-1$ arêtes ;
- l'algorithme de Prim : il utilise le fait qu'un arbre est un graphe connexe à $n-1$ arêtes.

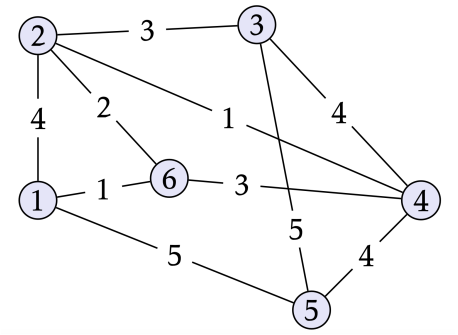
3.2 Algorithme de Kruskal

L'idée de cet algorithme est 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.

L'algorithme se déroule de la façon suivante :

Entrée : Graphe $G = (S, A, w)$
Début algorithme
  $E = \emptyset$
  $A_t = \text{tricroissant}(A)$
  Pour $\{a, b\} \in A_t$ **faire**
    Si $E \cup \{\{a, b\}\}$ est acyclique **alors**
      Ajouter $\{a, b\}$ à l'ensemble E
  Retourner (S, E)

Exercice. Appliquer l'algorithme de Kruskal au graphe suivant :



Complexité. Considérons un graphe $G = (S, A, w)$ à n sommets et p arêtes.

- Trier les arêtes par poids croissants : $O(p \log_2(p))$;
- Pour chaque arête $\{a, b\}$, déterminer si l'ajout de cette arête crée un cycle et, pour le savoir, effectuer un parcours de graphe en $O(n + p)$.

L'algorithme de Kruskal est donc de complexité $O(p(n + p))$.

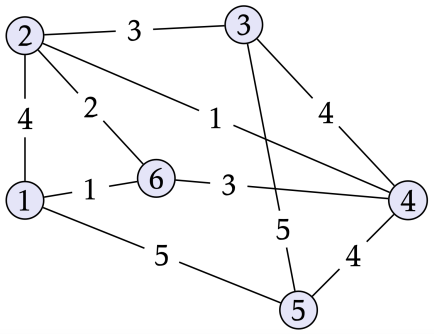
3.3 Algorithme de Prim

L'idée de cet algorithme est de maintenir un sous-graphe partiel connexe, 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 cherche parmi les arêtes sortantes celle de poids le plus faible.

L'algorithme se déroule de la façon suivante :

Entrée : Graphe $G = (S, A, w)$ et sommet s_0 (choisi arbitrairement)
Début algorithme
  $V = \{s_0\}$
  $E = \emptyset$
  Tant que $V \neq S$ **faire**
    Extraire $\{a, b\} \in A$ telle que $a \in V$, $b \in S \setminus V$ et $w(\{a, b\})$ minimal
    Ajouter b à l'ensemble V
    Ajouter $\{a, b\}$ à l'ensemble E
  Retourner (V, E)

Exercice. Appliquer l'algorithme de Prim au graphe suivant (en partant du sommet 1) :



Complexité. Etant donné un graphe $G = (S, A, w)$ à n sommets et p arêtes, la complexité de l'algorithme de Prim est $O(p \log(n))$, c'est-à-dire une complexité semblable à celle de l'algorithme de Kruskal.

4 Graphes bipartis et couplages

4.1 Graphes bipartis

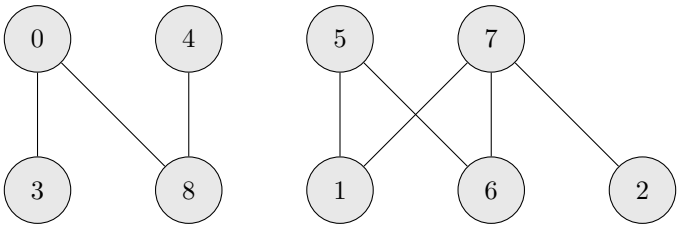
Définition.

Un graphe non orienté $G = (S, A)$ est un **graphe biparti** si il existe une partition de S en deux sous-ensembles S_1 et S_2 telle que :

$$\forall \{a, b\} \in A, \quad (a \in S_1 \text{ et } b \in S_2) \quad \text{ou} \quad (a \in S_2 \text{ et } b \in S_1).$$

Autrement dit, il n'existe aucune arête entre des sommets de S_1 ou entre des sommets de S_2.

Exemple. Voici un graphe biparti avec $S_1 = \{0, 4, 5, 7\}$ et $S_2 = \{1, 2, 3, 6, 8\}$:



Propriété 7 (Caractérisation des graphes bipartis)

Un graphe non orienté est biparti si et seulement si il ne contient pas de cycle de longueur impair.

Preuve.

□

Remarque. Un arbre est un graphe biparti puisqu'il est acyclique.

Implémentation. On considère un graphe connexe g défini sur OCaml par liste d'adjacence avec le type `int list array`.

Ecrire une fonction `biparti g` renvoyant un tableau de couleurs (0 ou 1) des sommets si g est biparti et qui déclenche une exception sinon. On partira arbitrairement du sommet 0 en lui donnant la couleur 0 et on parcourt g en profondeur. A chaque appel récursif sur un voisin, on change de couleur.

4.2 Couplages maximums d'un graphe biparti

Définition.

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

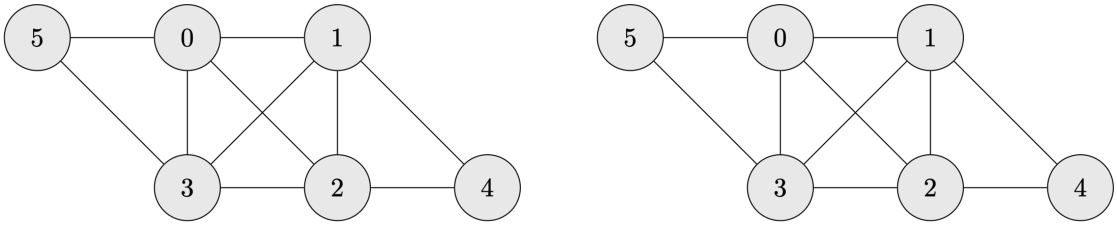
- Un **couplage** M est un ensemble d'arêtes deux à deux non adjacentes :

$$\forall (a_1, a_2) \in M^2, \quad a_1 \neq a_2 \ \Rightarrow \ a_1 \cap a_2 = \emptyset.$$

  Autrement dit, deux arêtes quelconques de M n'ont aucune extrémité en commun.

- Un **couplage maximum** est un couplage M tel que si une arête $a \in A$ est ajoutée à M, alors $M \cup \{a\}$ n'est pas un couplage.

Exercice. Proposer (en coloriant les arêtes) deux couplages maximums différents du graphe suivant :



Définition.

Soit $G = (S, A)$ un graphe non orienté et M un couplage de G.
Un sommet s de G est dit **couplé** par M s'il appartient à une arête de M. Dans le cas contraire, on dira que s est **non couplé** ou **exposé**

De nombreux algorithmes existent pour rechercher un couplage maximum et nous nous limitons ici à une approche basée sur des chemins augmentants :

Définition.

Soit $G = (S, A)$ un graphe non orienté et M un couplage de G.

- Un **chemin alternant** est un chemin dans lequel les arêtes appartiennent alternativement à M et à son complémentaire $A \setminus M$.
- Un **chemin augmentant** est un chemin alternant qui commence et se termine par un sommet non couplé.

La différence symétrique Δ entre un couplage M et les arêtes $A(C)$ d'un chemin alternant C définie par

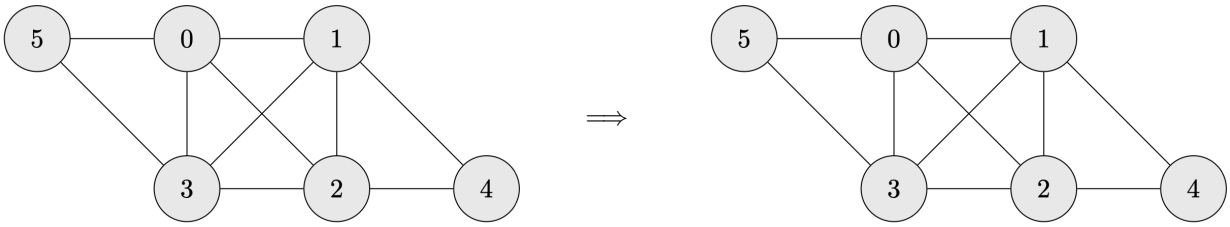
$$M \Delta A(C) = (M \setminus A(C)) \cup (A(C) \setminus M)$$

est aussi un couplage de taille égale à

- $|M| - 1$ si les deux extrémités de C sont couplés ;
- $|M|$ si une extrémité de C est non couplé ;
- $|M| + 1$ si les deux extrémités de C sont non couplés.

On peut ainsi utiliser un chemin augmentant C pour transformer M en un couplage plus grand.

Exercice. Faire une augmentation du couplage $M = \{\{0, 1\}, \{2, 3\}\}$ par le chemin augmentant $C = 5-3-2-4$:



Propriété 8 (Chemin augmentant et couplage maximal)

Soit $G = (S, A)$ un graphe biparti et M un couplage de G.
M est un couplage maximum si et seulement si il n'existe pas de chemin augmentant.

Preuve.

□

Implémentation. Le théorème précédent donne la trame générale d'un algorithme de recherche d'un couplage maximum dans un graphe :

Entrée : Graphe biparti $G = (S, A)$
Sortie : Un couplage maximum de G
Début algorithme
    $M = \emptyset$
    $C =$ chemin augmentant de (G, M)
    Tant que $C \neq \emptyset$ **faire**
        $M = M \Delta A(C)$
        $C =$ chemin augmentant de (G, M)
    Retourner M