CHAPITRE 2

Théorie des graphes et algorithmes fondamentaux

II.1. Introduction

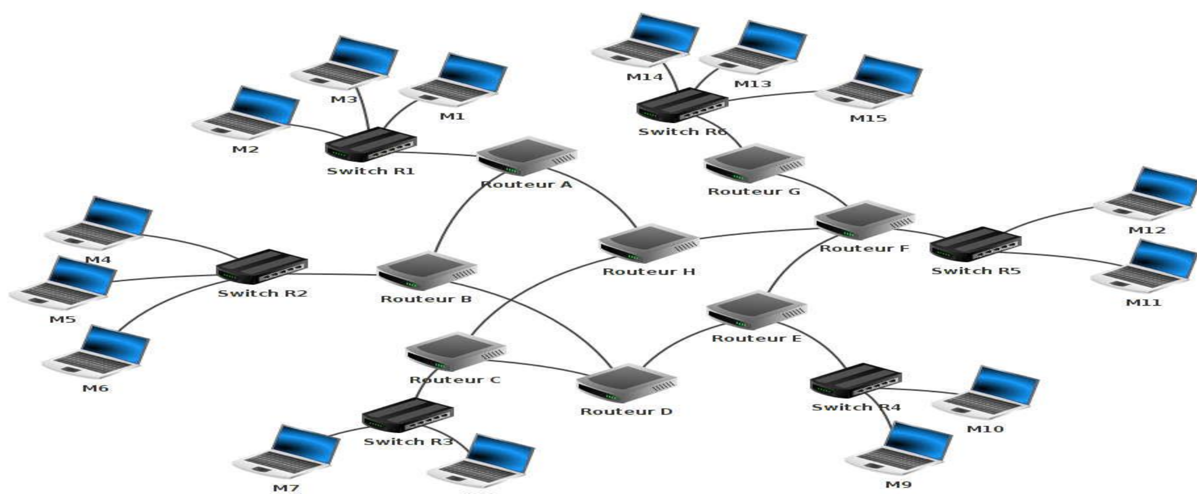
Pour résoudre de nombreux problèmes concrets, on est amené à tracer sur le papier des petits dessins qui représentent (partiellement) le problème à résoudre. Bien souvent, ces petits dessins se composent de points et de lignes continues reliant deux à deux certains de ces points. On appellera ces petits dessins des **graphes**, les points des **sommets** et les lignes des **arcs** ou **arêtes**, selon que la relation binaire sous-jacente est orientée ou non.

En résumé, les graphes constituent donc une méthode de pensée qui permet de modéliser une grande variété de problèmes en se ramenant à l'étude de sommets et d'arcs. Les derniers travaux en théorie des graphes sont souvent effectués par des **informaticiens**, du fait de l'importance qu'y revêt **l'aspect algorithmique**.

Beaucoup de problèmes sur les graphes nécessitent que l'on parcourt l'ensemble des sommets et des arcs/arêtes du graphe. On étudie dans la suite les deux principales stratégies d'exploration :

- le parcours en largeur consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné ;
- le parcours en profondeur consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible (jusqu'à un cul-de-sac ou un cycle), puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

Dans ce chapitre, nous allons présenter les algorithmes de Dijkstra et Ford-Bellman qui résoudre le problème du plus court chemin (i.e., un chemin de poids minimal) d'un sommet u fixé à un sommet quelconque de G. A la fin, nous allons présenter les deux algorithmes Kruskal et Prim qui calculer l'arbre couvrant minimal (ACM) à partir d'un graphe.



II.2. Quelques exemples de modélisation par des graphes

La notion de graphes et les problèmes liés aux graphes peuvent être rencontrés dans différentes situations de la vie réelle ainsi que dans des problèmes d'ingénierie notamment en informatique.

Les graphes représentent, d'abord, un moyen de modélisation, ainsi qu'un moyen permettant de raisonner sur de nombreux problèmes.

Les graphes permettent de modéliser des entités reliées par des liens. La disposition des entités et surtout des liens (ce que l'on appelle la topologie du graphe) permet d'induire plusieurs propriétés intéressantes. Pour illustrer cela, nous allons prendre quelques exemples introductifs.

Exemple 2.1 (modélisation d'un réseau d'amis)

Dans la vie courante, une personne est amie avec plusieurs personnes pouvant aussi être des amis à d'autres personnes, et ainsi de suite. Dans les réseaux sociaux par exemple, ces relations permettent d'établir des communautés et des règles s'appliquant au partage des données. Il est intéressant de noter (et ceci peut être montré par la théorie des graphes) que dans n'importe ensemble de groupes d'amis, il y a toujours au moins deux personnes ayant le même nombre d'amis.

Exemple 2.2 (modélisation d'un réseaux routier)

Le réseau routier d'un pays peut être représenté par un graphe dont les sommets sont les villes. Si l'on considère que toutes les routes sont à double sens, on utilisera un graphe non orienté et on reliera par une arête tout couple de sommets correspondant à deux villes reliées par une route (si l'on considère en revanche que certaines routes sont à sens unique, on utilisera un graphe orienté). Ces arêtes pourront être valuées par la longueur des routes correspondantes. Etant donné un tel graphe, on pourra s'intéresser, par exemple, à la résolution des problèmes suivants :

- ➔ Quel est le plus court chemin, en nombre de kilomètres, passant par un certain nombre de villes données ?
- ➔ Quel est le chemin traversant le moins de villes pour aller d'une ville à une autre ?
- ➔ Est-il possible de passer par toutes les villes sans passer deux fois par une même route ?

Exemple 2.3 (modélisation d'un réseau de télécommunication)

Un réseau informatique est constitué d'un ensemble de machines (des ordinateurs, des hubs, des switches, des routeurs, des répétiteurs, etc.) et des liaisons physiques (câbles métalliques, fibres optiques, ondes radio, etc.). Un routeur permet d'acheminer un paquet vers la bonne sortie afin que ce dernier puisse retrouver son chemin vers la destination. On est notamment intéressée par connaître le chemin optimal dans le réseau (en termes de temps de transmission, énergie utilisée, nombre de sauts, etc.) ainsi que le débit maximal du réseau, ce qui permet d'éviter des situations agaçantes comme la congestion du réseau.

II.3. Différentes notions de graphes

D'une manière informelle, on peut définir un graphe par une représentation figurative où l'on retrouve un ensemble de points (appelés nœuds ou sommets) reliés par un ensemble de courbes

droites ou non. Ces liens représentent généralement une relation ou une dépendance entre les sommets. Ils sont appelés des arêtes si la relation est symétrique (on parle de graphe non-orienté), ou des arcs sinon (on parle de graphe orienté).

II.3.1. Graphes non orientés

Définition 2.1 (Graphe non orienté)

Un graphe non orienté **G** est la donnée d'un couple **G = (S, A)** tel que :

➔ **S** est un ensemble fini de sommets,

➔ **A** est un ensemble de couples non ordonnés de sommets $\{s_i , s_j\} \in S^2$.

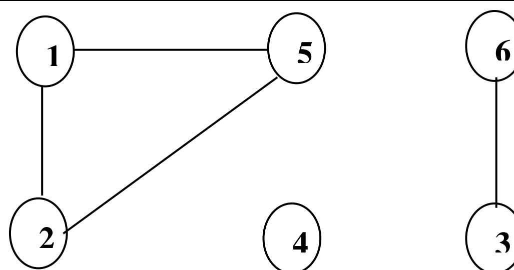
Une paire $\{s_i , s_j\}$ est appelée une **arête**, et est représentée graphiquement par $s_i - s_j$. On dit que les sommets s_i et s_j sont adjacents. L'ensemble des sommets adjacents au sommet $s_i \in S$ est noté $Adj(s_i) = \{s_i \in S, \{s_i , s_j\} \in A\}$.

Exemple

Le graphe ci-contre représente un graphe non orienté

G = (S, A) avec **S = {1, 2, 3, 4, 5, 6}**

et A = {{1, 2}, {1, 5}, {5, 2}, {3, 6}}.



Définition 2.2 (Boucle, simple graphe, multi-graphe, ordre, taille)

➔ Une **boucle** est une arête reliant un sommet à lui-même.

➔ Un graphe non-orienté est dit **simple** s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non-orienté qui n'est pas simple est un **multi-graphe**. Dans le cas d'un multi-graphe, A n'est plus un ensemble mais un multi-ensemble d'arêtes.

➔ On appelle **ordre d'un graphe** le nombre de ses sommets, i.e. c'est **card(S) ou |S|**.

➔ On appelle **taille d'un graphe** le nombre de ses arêtes, i.e. c'est **card(A) ou |A|**.

II.3.2. Graphes orientés

Définition 2.3 (Graphe orienté)

Un graphe orienté **G** est la donnée d'un couple **G = (S, A)** tel que :

➔ **S** est un ensemble fini de sommets,

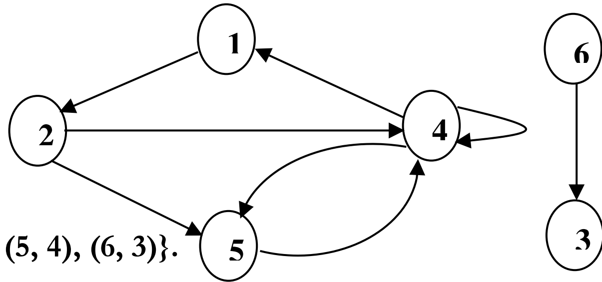
➔ **A** est un ensemble de couples ordonnés de sommets $(s_i , s_j) \in S^2$.

Un couple (s_i , s_j) est appelé un **arc**, et est représenté graphiquement par $s_i \rightarrow s_j$, s_i est le sommet initial ou origine, et s_j le sommet terminal ou extrémité. L'arc $a = (s_i , s_j)$ est dit **sortant** en s_i et **incident** en s_j, et s_j est un **successeur** de s_i, tandis que s_i est un **prédécesseur** de s_j.

➔ L'ensemble des successeurs d'un sommet $s_i \in S$ est noté $Succ(s_i) = \{s_j \in S, (s_i , s_j) \in A\}$.

➔ L'ensemble des prédécesseurs d'un sommet $s_i \in S$ est noté $Pred(s_i) = \{s_j \in S, (s_j , s_i) \in A\}$.

Exemple



Le graphe ci-contre représente un graphe orienté

G = (S, A) avec **S = {1, 2, 3, 4, 5, 6}**

et **A = {(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)}**.

Définition 2.4 (Boucle, élémentaire, p-graphe)

➔ Une **boucle** est un arc reliant un sommet à lui-même.

➔ Un graphe orienté est dit **élémentaire** s'il ne contient pas de boucle.

➔ Un graphe orienté est un **p-graphe** s'il comporte au plus **p** arcs entre deux sommets. Le plus souvent, on étudiera des **1-graphes**.

II.4. Arbres et Arborescences

Les arbres et les arborescences sont des graphes particuliers très souvent utilisés en informatique pour représenter des données.

Définition 2.5 (arbre)

Un **arbre** est un graphe non orienté **G** qui vérifie une des conditions équivalentes suivantes :

➔ **G** est connexe et acyclique

➔ **G** est sans cycle et possède **n - 1** arêtes

➔ **G** est connexe et admet **n - 1** arêtes

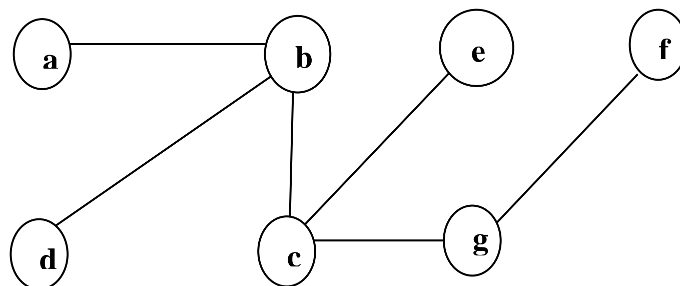
➔ **G** est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,

➔ **G** est connexe, et en supprimant une arête quelconque, il n'est plus connexe,

➔ Il existe une chaîne et une seule entre **2** sommets quelconques de **G**.

Exemple

Le graphe suivant est un arbre



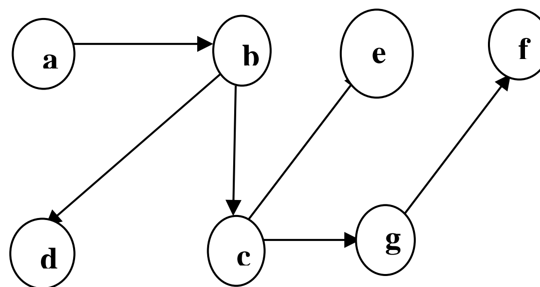
Définition 2.6 (forêt, arborescence)

➔ On appelle **forêt** un graphe dont chaque composante connexe est un arbre.

➔ Une **arborescence** est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i. Si l'arborescence comporte **n** sommets, alors elle comporte exactement **n - 1** arcs.

Exemple

Le graphe suivant est une arborescence de racine **a** :



II.5. Parcours de graphes

Le parcours d'un graphe est un problème type de théorie des graphes auquel peuvent se ramener de nombreux autres problèmes d'algorithmique.

Définition 2.7 (parcours d'un graphe)

Soit **G = (S, A)** un graphe et **x ∈ S** un sommet, un parcours du graphe **G** à partir de **x** est une **visite de chaque sommet accessible depuis x**. Le résultat d'un parcours est un ensemble de chemins partants de **x** allant vers les sommets accessibles depuis **x**. **Un parcours peut être représenté par** sous-graphe de **G** qui est **un arbre de racine x** (ou une forêt si certains sommets de **G** ne sont pas accessibles depuis **x**).

D'un point de vue algorithmique, un parcours correspond à la procédure suivante :

A chaque étape de la boucle **Tant que** un seul sommet **y** est traité qui engendre le début du traitement d'un ou plusieurs autre sommets . . .
On peut définir un **ordre de parcours** en numérotant à un endroit précis de la boucle **Tant que** le sommet **y** en cours de traitement.

Pour déterminer l'ensemble des sommets accessibles à partir d'un sommet donné $\mathbf{s_0}$, il est nécessaire de parcourir le graphe de façon systématique, en partant de $\mathbf{s_0}$ et en utilisant les arcs pour découvrir de nouveaux sommets. Dans ce cas, nous étudions les deux principales stratégies d'exploration :

- le parcours en largeur, qui consiste à explorer les sommets du graphe niveau par niveau, à partir de $\mathbf{s_0}$;
- le parcours en profondeur, qui consiste à partir de $\mathbf{s_0}$ pour suivre un chemin le plus loin possible, jusqu'à un cul-de-sac ou l'arrivée sur un sommet déjà visité, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

Dans les deux cas, l'algorithme procède par coloriage des sommets.

- Initialement, tous les sommets sont blancs. Nous dirons qu'un sommet blanc n'a pas encore été découvert.
- Lorsqu'un sommet est "découvert" (autrement dit, quand l'algorithme arrive pour la première fois sur ce sommet), il est colorié en gris.
- Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

Procédure parcours(G, x)
L = liste des sommets à traiter (vide au départ)
mettre x dans L (début du traitement de x)
Tant que L ≠ ∅ **Faire**
sortir le **1**er sommet y de L (y en cours de traitement)
V = successeurs non traités de y
Pour tout $z \in V$ **Faire**
mettre z dans L (début du traitement de z)
Fin Pour
fin du traitement de y
Fin Tant que
Fin

De façon pratique, l'algorithme utilise une liste "d'attente au coloriage en noir" qui contient l'ensemble des sommets gris. Un sommet est mis dans la liste d'attente dès qu'il est colorié en gris. À chaque itération, un sommet gris de la liste d'attente fait rentrer dans la liste un (ou plusieurs) de ses successeurs qui sont encore blancs (en les coloriant en gris). Quand tous les successeurs d'un sommet gris de la liste d'attente sont soit gris soit noirs, il peut être colorié en noir et sortir de la liste d'attente.

La différence fondamentale entre le parcours en largeur et le parcours en profondeur provient de la façon de gérer cette liste d'attente au coloriage en noir : le parcours en largeur utilise une file d'attente FIFO (First In, First Out), où le premier sommet arrivé dans la file est aussi le premier à en sortir, tandis que le parcours en profondeur utilise une pile LIFO (Last In, First Out), où le dernier sommet arrivé dans la pile est le premier à en sortir.

Arborescence liée à un parcours de graphe

Au fur et à mesure du parcours, l'algorithme construit une arborescence de découverte des sommets accessibles depuis le sommet de départ $\mathbf{s_0}$, appelée arborescence d'un parcours à partir de $\mathbf{s_0}$. Cette arborescence contient un arc ($\mathbf{s_i}$, $\mathbf{s_j}$) si et seulement si le sommet $\mathbf{s_j}$ a été découvert à partir du sommet $\mathbf{s_i}$ (autrement dit, si c'est le sommet $\mathbf{s_i}$ qui a colorié $\mathbf{s_j}$ en gris). Ce graphe est effectivement une arborescence, dans la mesure où chaque sommet **a** au plus un prédécesseur, à partir duquel il a été découvert. La racine de cette arborescence est le sommet de départ du parcours, $\mathbf{s_0}$.

L'arborescence associée à un parcours de graphe est mémorisée dans un tableau $\boldsymbol{\pi}$ tel que $\boldsymbol{\pi[s_j]=s_i}$ si $\mathbf{s_j}$ a été découvert à partir de $\mathbf{s_i}$, et $\boldsymbol{\pi[s_k]} = \mathbf{null}$ si $\mathbf{s_k}$ est la racine, ou s'il n'existe pas de chemin de la racine vers $\mathbf{s_k}$.

II.5.1. Parcours en largeur (Breadth First Search = BFS)

Le parcours en largeur est effectué en gérant la liste d'attente au coloriage comme une file d'attente (FIFO = First In First Out). Autrement dit, l'algorithme enlève à chaque fois le plus vieux sommet gris dans la file d'attente (ce sommet sera nommé $\mathbf{s_{FirstOut}}$), et il introduit tous les successeurs blancs de ce sommet dans la file d'attente, en les coloriant en gris. **L'algorithme 1** décrit ce principe.

Algorithme 1 : Parcours en largeur d'un graphe

1 : **Fonction** *BFS(**g**, **s_0**)*
Entrée : Un graphe **g** et un sommet **s_0** de **g**
Postcondition : Retourne une arborescence **π** d'un parcours en largeur de **g** à partir de **s_0**
Déclaration : Une file (FIFO) **f** initialisée à vide
2 : **Pour** chaque sommet s_i de g **Faire**
3 : π[s_i] ← *null*
4 : Colorier s_i en blanc
5 : **Fin Pour**
6 : Ajouter **s_0** dans la file **f** et colorier **s_0** en gris
7 : **Tant que** la file **f** n'est pas vide **Faire**
8 : Soit **$s_{FirstOut}$** le sommet le plus ancien dans **f**
9 : **Pour** tout sommet **s_i** ∈ **succ($s_{FirstOut}$)** **Faire**
10 : **Si** **s_i** est blanc **Alors**
11 : Ajouter **s_i** dans la file **f** et colorier **s_i** en gris
12 : **π[s_i] ← $s_{FirstOut}$**
13 : **Fin Si**
14 : **Fin Pour**
15 : Enlever **$s_{FirstOut}$** de **f** et colorier **$s_{FirstOut}$** en noir
16 : **Fin Tant que**
17 : **Retourne π**
18 : **Fin**

Complexité. Soient **n** et **p** le nombre de sommets et arcs de **g**, respectivement. Chaque sommet accessible depuis **s_0** est mis au plus une fois dans la file **f**. En effet, seuls les sommets blancs entrent dans la file, et un sommet blanc est colorié en gris quand il entre dans la file, puis en noir quand il en sort, et ne pourra jamais redevenir blanc. À chaque passage dans la boucle lignes **7** à **16**, il y a exactement un sommet qui est enlevé de la file. Cette boucle sera donc exécutée au plus **n** fois. À chaque fois qu'un sommet est enlevé de la file, la boucle lignes **9** à **14** parcourt tous les successeurs du sommet enlevé, de sorte que les lignes 10 à 13 seront exécutées au plus une fois pour chaque arc. Par conséquent, la complexité de **l'algorithme 1 (BFS)** est :

- **$O(n^2)$** dans le cas d'une implémentation par matrice d'adjacence
- **$O(n + p)$** dans le cas d'une implémentation par listes d'adjacence.

II.5.2. Parcours en profondeur (Depth First Search = DFS)

Le parcours en profondeur est obtenu en gérant la liste d'attente au coloriage en noir comme une pile (LIFO = Last In First Out). Autrement dit, l'algorithme considère à chaque fois le dernier sommet gris entré dans la pile, et introduit devant lui tous ses successeurs blancs. Ce principe est décrit dans **l'algorithme 2**.

Algorithme 2 : Parcours en profondeur d'un graphe

1 : **Fonction** *DFS(**g**, **s_0**)*
Entrée : Un graphe **g** et un sommet **s_0** de **g**
Postcondition : Retourne une arborescence **π** d'un parcours en profondeur de **g** à partir de **s_0**
Déclaration : Une pile (LIFO) **p** initialisée à vide
2 : **Pour** chaque sommet s_i de *g* **Faire**
3 : π[s_i] ← *null*
4 : Colorier s_i en blanc
5 : **Fin Pour**
6 : Empiler **s_0** dans la pile **p** et colorier **s_0** en gris
7 : **Tant que** la pile **p** n'est pas vide **Faire**
8 : Soit **s_i** le dernier sommet entré dans **p** (au sommet de **p**)
9 : **Si** $\exists$ **s_j** $\in$ succ(**s_i**) tel que **s_j** soit blanc **Alors**
10 : Empiler **s_j** dans la pile **p** et colorier **s_j** en gris
11 : **π[s_j]** ← s_i
12 : **Sinon**
13 : Dépiler **s_i** de **p** et colorier **s_i** en noir
14 : **Fin Si**
15 : **Fin Tant que**
16 : **Retourne π**
17 : **Fin**

Complexité : Chaque sommet accessible depuis **s_0** est mis, puis enlevé, exactement une fois dans la pile, comme dans BFS, et à chaque passage dans la boucle lignes **7** à **15**, soit un sommet est empilé (si **s_i** a encore un successeur blanc), soit un sommet est dépilé (si **s_i** n'a plus de successeur blanc). Par conséquent, l'algorithme passera au plus **2n** fois dans la boucle lignes **7** à **15**. À chaque passage, il faut parcourir la liste des successeurs de **s_i** pour chercher un successeur blanc. Si nous utilisons un itérateur qui mémorise pour chaque sommet de la pile le dernier successeur de ce sommet qui était blanc. Par conséquent, si le graphe contient **n** sommets (accessibles à partir de **s_0**) et **p** arcs/arêtes, alors la complexité de **l'algorithme 2 (DFS)** est :

- **O(n^2)** dans le cas d'une implémentation par matrice d'adjacence,
- **O(n + p)** dans le cas d'une implémentation par listes d'adjacence.

II.6. Problème du plus court chemin

On se place dans le cas des graphes orientés valués **G = (S, A, ν).** Mais les résultats et les algorithmes présentés se généralisent facilement aux cas des graphes non orientés valués. Une autre solution consiste à transformer le graphe non-orienté en un graphe orienté en remplaçant une arête entre deux sommets par deux arcs de sens inverse entre ces sommets.

Définition 2.8 (coût ou poids d'un chemin, coût d'un plus court chemin)

Le **coût ou poids d'un chemin c =< s_0 , s_1 , s_2 , … , s_k >** est égale à la somme des valuations des arcs composant le chemin, c'est à dire,

$$L(c) = \sum_{i=1}^{k} v(s_{i-1}, s_i)$$

Le **coût d'un plus court chemin** entre deux sommets s_i et s_j est noté $\delta(s_i , s_j)$ est défini par :

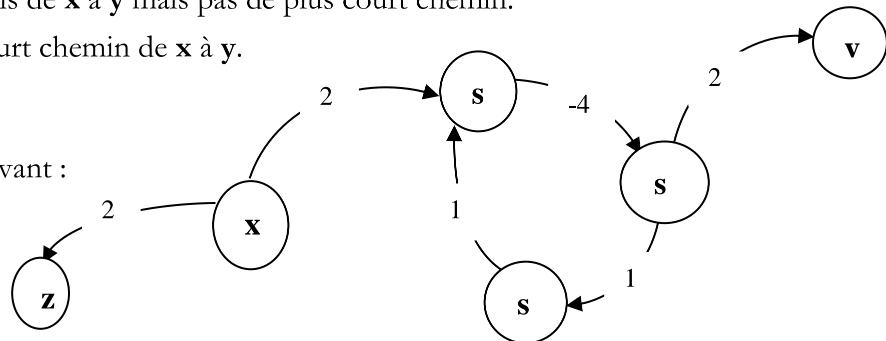
$$\delta'(s_i, s_j) = \begin{cases} \min\{(L(c)/c = \text{chemin de } s_i \text{ à } s_j\} \text{ s'il existe au moins un chemin entre } s_i \text{ et } s_j \\ +\infty \quad \text{Sinon} \end{cases}$$

Dans la recherche d'un plus court chemin de **x** à **y**, trois cas peuvent se présenter :

- Il n'existe aucun chemin de **x** à **y** (par exemple, si **x** et **y** appartiennent à deux composantes fortement connexes différentes de **G**).
- Il existe des chemins de **x** à **y** mais pas de plus court chemin.
- Il existe un plus court chemin de **x** à **y**.

Exemple

On considère le graphe suivant :



Il existe des chemins de **x** à **y** (et même une infinité), mais il n'existe pas de plus court chemin de **x** à **y** (il suffit d'emprunter le cycle de poids négatif autant de fois que nécessaire). Il existe un plus court chemin de **x** à **z** mais pas de chemin de **z** à **x**.

Définition 2.9 (circuit absorbant.)

Un circuit de longueur négative est appelé **circuit absorbant.**

Et alors, une condition nécessaire et suffisante d'existence de plus court chemin est donnée par le résultat suivant.

Proposition 2.1

Dans un graphe orienté valué fortement connexe **G = (S, A, ν)**, il existe un plus court chemin entre tout couple de sommets si et seulement s'il n'existe pas de circuit absorbant dans **G**.

Preuve : On montre que si un chemin entre deux sommets **x** et **y** possède un circuit absorbant, alors $\pi(x, y) = -\infty$. En effet, dès que l'on parcourt le circuit absorbant, on diminue la longueur du chemin, et on peut donc diminuer cette longueur à l'infini. En augmentant "infiniment" le nombre de tours dans le circuit, on obtient $\pi(x , y) = -\infty$.

Définition 2.10 (problème des plus courts chemins à origine unique)

Étant donné un graphe orienté valué **G = (S, A, ν)** et un sommet origine $s_0 \in S$, le **problème des plus courts chemins à origine unique** consiste à calculer pour chaque sommet $s_j \in S$ le coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j.

➔ On supposera que le graphe **G** ne comporte pas de circuit absorbant.

Variantes du problème :

- Si l'on souhaite calculer le plus court chemin allant d'un sommet s_0 vers un autre sommet s_i (la destination est unique), on pourra utiliser la résolution du problème précédant (qui calcule tous les plus courts chemins partant de s_0). En effet, on ne connaît pas d'algorithme plus efficace pour résoudre ce problème.
- Si l'on souhaite calculer tous les plus courts chemins entre tous les couples de sommets possibles, on pourrait aussi utiliser la résolution du problème précédent, mais dans ce cas, on n'obtiendrait pas un algorithme optimal.

II.6.1. Principe commun aux algorithmes

Nous allons étudier deux algorithmes qui permettent de résoudre des problèmes de recherche de plus courts chemins à origine unique :

- un algorithme (Dijkstra) qui peut être utilisé dès lors que tous les coûts sont positifs ou nuls;
- un algorithme (Ford-Bellman) qui peut être utilisé pour n'importe quel graphe ne comportant pas de circuit absorbant.

Les deux algorithmes procèdent de la même façon, en associant à chaque sommet s_i une borne supérieure $d[s_i]$ du coût du plus court chemin de s_0 jusque s_i de sorte que, à tout moment, $d[si] \geq \delta(s_0, s_i)$. Au départ, ces bornes sont initialisées à $d[s_i] = +\infty$. pour tous les sommets s_i sauf pour le sommet initial s_0 pour lequel $d[s_0]$ est initialisé à $\delta(s_0, s_0) = 0$. Ensuite, les algorithmes vont grignoter progressivement ces bornes jusqu'à ce qu'elles ne puissent plus être diminuées et que $d[s_i] = \delta(s_0, s_i)$ pour chaque sommet s_i. Pour grignoter les valeurs de **d**, les algorithmes vont itérativement examiner chaque arc $s_i \rightarrow s_j$ du graphe, et regarder s'il est possible de diminuer la valeur de $d[s_j]$ en passant par s_i. Cette opération de grignotage est appelée "**relâchement de l'arc** (s_i , s_j)", et est décrite dans **l'algorithme 3.**

Les algorithmes de Dijkstra et Bellman-Ford procèdent tous les deux par relâchements successifs d'arcs. La différence entre les deux est que dans l'algorithme de Dijkstra, chaque arc est relâché une et une seule fois, tandis que dans l'algorithme de Bellman-Ford, chaque arc peut être relâché plusieurs fois.

Algorithme 3 : Relâchement de l'arc (s_i, s_j)

```
1 : Procédure relâcher((si, sj), ν , d, π)
       Entrée : Un arc (si, sj) ; ν : valuation des arcs (ν(si, sj) = cout(si, sj))
       Entrée/Sortie : d tableau des bornes maximum des coûts,
                       π arborescence couvrante
       Précondition : d[si] ≥ δ(s0 , si) et d[sj] ≥ δ(s0 , sj)
       Postcondition : δ(s0 , sj) ≤ d[sj] ≤ d[si] + ν(si, sj) et π[sj] = si
                       si la borne d[sj] a été diminuée
2 :    Si d[sj] > d[si] + ν (si, sj) Alors
3 :        d[sj] ← d[si] + ν (si, sj)
4 :        π[sj] ← si
5 :    Fin Si
6 : Fin
```

A. Algorithme de Dijkstra

L'idée consiste à maintenir **2** ensembles disjoints **E** et **F** tels que $E \cup F = S$. L'ensemble **E** contient chaque sommet si pour lequel on connaît un plus court chemin depuis s_0 (c'est-à-dire pour lequel $d[s_i] = \delta(s_0, s_i)$). L'ensemble **F** contient tous les autres sommets. À chaque itération de l'algorithme, on choisit le sommet s_i dans **F** pour lequel la valeur $d[s_i]$ est minimale, on le rajoute dans **E**, et on relâche tous les arcs partant de ce sommet s_i.

Correction de l'algorithme de Dijkstra : On peut montrer qu'à chaque fois qu'un sommet s_i entre dans l'ensemble **E**, on a $d[s_i] = \delta(s_0, s_i)$. En effet, le premier sommet à entrer dans l'ensemble **E** est s_0, pour lequel $d[s_0] = 0 = \delta(s_0 , s_0)$. À chaque itération, on fait entrer dans **E** un sommet $s_i \in F$ tel que $d[s_i]$ soit minimal.

L'idée est que, dans ce cas, s'il existe un autre chemin allant de s_0 jusque s_i, alors il passera nécessairement par un sommet $s_j \in F$ tel que $d[s_j] > d[s_i]$ (puisque $d[s_i]$ est minimal). Sachant que la fin de ce chemin (de s_j à s_i) ne peut faire qu'augmenter la distance du chemin, cet autre chemin sera forcément plus long. Par conséquent, on ne pourra pas trouver de chemin plus court pour aller de s_0 à s_i, et on peut faire entrer s_i dans **E**, et relâcher tous les arcs qui partent de s_i.

Algorithme 4 : Dijkstra(S , A , ν , s_0 , d , π)

Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs, s_0 sommet de départ
Sorties : **d** tableau des bornes maximum des coûts, **π** arborescence couvrante

```
1 :  Pour chaque sommet si ∈ S Faire
2 :     d[si] ← +∞
3 :     Π[si] ← nil
4 :  Fin Pour
5 :     d[s0] ← 0
6 :     E ← ∅
7 :     F ← S
8 :  Tant que F ≠ ∅   Faire
9 :      soit si le sommet de F tel que d[si] soit minimal
10 :     /* d[si] = δ(si , si) */
11 :     F ← F – { si}
12 :     E ← E ∪ { si}
13       Pour tout sommet sj ∈ succ(si) ∩ F Faire
14 :            relâcher((si, sj), ν  , d, π)
15 :      Fin Pour
16 :   Fin Tant que
17 : Fin
```

Complexité : La complexité de cet algorithme dépend de l'implémentation du graphe (par matrice ou par listes d'adjacence), mais aussi de la façon de gérer l'ensemble **F**. On suppose que le graphe possède **n** sommets et **m** arcs. Si on utilise une matrice d'adjacence, l'algorithme sera en $O(n^2)$.

En revanche, si on utilise des listes d'adjacence, alors :

Si **F** est implémenté par une liste linéaire, ou un tableau, il faudra chercher, à chaque itération, le sommet dans **F** ayant la plus petite valeur de **d**. Étant donné qu'il y a **n** itérations, et qu'au premier passage **F** contient **n** éléments, et qu'à chaque passage suivant **F** contient un élément de moins, il faudra au total faire de l'ordre de **n + (n - 1) + (n - 2) + … + 2 + 1** opérations, soit **O(n2).** En revanche, chaque arc étant relâché une seule fois, les opérations de relâchement prendront de l'ordre de **m** opérations. Au total on aura donc une complexité en **O(n2)**.

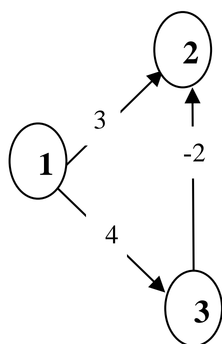
Pour améliorer les performances de l'algorithme, il faut trouver une structure de données permettant de trouver plus rapidement la plus petite valeur dans l'ensemble **F**. Pour cela, on peut utiliser un **tas binaire** : un tas binaire permet de trouver le plus petit élément d'un ensemble en temps constant. En revanche, l'ajout, la suppression ou la modification d'un élément dans un tas binaire comportant **n** éléments prendra de l'ordre de $\log_2(n)$ opérations. Par conséquent, si on implémente **F** avec un tas binaire, on obtient une complexité pour Dijkstra en **O(m ∗ log(n)).**

B. Algorithme de Bellman-Ford

L'algorithme de Dijkstra ne marche pas toujours quand le graphe contient des arcs dont les coûts sont négatifs.

Exemple

Considérons par exemple le graphe suivant, où l'on cherche les plus courts chemins à partir du sommet **1**. On voit immédiatement que la plus courte distance entre les sommets **1** et **2** est **2**, alors que l'algorithme de Dijkstra donne **3**.



			Dist			Préd		
s	**E**	**F**	**1**	**2**	**3**	**1**	**2**	**3**
0	{}	{1, 2, 3}	0	∞	∞	0	0	0
1	{1}	{2, 3}	0	3	4	0	1	1
2	{1, 2}	{3}	0	3	4	0	1	1
3	{1, 2, 3}	{}	0	3	4	0	1	1

L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants).

L'algorithme de Bellman-Ford fonctionne selon le même principe que celui de Dijkstra : on associe à chaque sommet s_i une valeur $d[s_i]$ qui représente une borne maximale du coût du plus court chemin entre s_0 et s_i. L'algorithme diminue alors progressivement les valeurs $d[s_i]$ en relâchant les arcs. Contrairement à Dijkstra, chaque arc va être relâché plusieurs fois. On relâche une première fois tous les arcs ; après quoi, tous les plus courts chemins de longueur **1**, partant de s_0, auront été trouvés. On relâche alors une deuxième fois tous les arcs ; après quoi tous les plus courts chemins de longueur **2**, partant de s_0, auront été trouvés... et ainsi de suite... Après la kième série de relâchement des arcs, tous les plus courts chemins de longueur **k**, partant de s_0, auront été trouvés. Étant donné que le graphe ne comporte pas de circuit absorbant, un plus court chemin est

nécessairement élémentaire. Par conséquent, si le graphe comporte **n** sommets, et s'il ne contient pas de circuit absorbant, un plus court chemin sera de longueur inférieure à **n** et au bout de **n - 1** passages, on aura trouvé tous les plus courts chemins partant de s_0. (Si le graphe contient un circuit absorbant, au bout de **n – 1** passages, on aura encore au moins un arc (s_i, s_j) pour lequel un relâchement permettrait de diminuer la valeur de **d[s_j]**. L'algorithme utilise cette propriété pour détecter la présence de circuits absorbants.)

Algorithme 5 : **Bellman-ford (S , A , ν , s_0 , d , π)**

Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs, s_0 sommet de départ
Sorties : **d** tableau des bornes maximum des coûts, **π** arborescence couvrante

```
1 :  Pour chaque sommet si ∈ S Faire
2 :     d[si] ← +∞
3 :     Π[si] ← nil
4 :  Fin Pour
5 :     d[s0] ← 0
6 :  Pour k variant de 1 à |S| - 1  Faire
7 :     Pour chaque arc (si, sj) ∈ A Faire
8 :           relâcher((si, sj), ν , d, π)
9 :     Fin Pour
10 : Fin Pour
11 : Pour chaque arc (si, sj) ∈ A Faire
12 :    relâcher((si, sj), ν , d, π)
13 :    Si d[sj] > d[si] + ν(si, sj) Alors
14 :          afficher("circuit absorbant")
15 :    Fin Si
16 : Fin Pour
17 : Fin
```

Complexité : Si le graphe comporte **n** sommets et **m** arcs, chaque arc sera relâché **n-1** fois, et on effectuera donc au total **(n-1)m** relâchements successifs. Si le graphe est représenté par une matrice d'adjacence, on aura une complexité en **O(n^3)**, alors que s'il est représenté par des listes d'adjacence, on aura une complexité en **O(nm)**.

Remarque : En pratique, on pourra arrêter l'algorithme dès lors qu'aucune valeur de **d** n'a été modifiée pendant une itération complète. On pourra aussi mémoriser à chaque itération l'ensemble des sommets pour lesquels la valeur de **d** a changé, afin de ne relâcher lors de l'itération suivante que les arcs partant de ces sommets.

II.7. Synthèse

En résumé, en fonction des caractéristiques du problème à résoudre il faudra choisir le bon algorithme :

- Si le graphe ne comporte pas de circuit alors, que l'on recherche un plus court chemin ou un plus long chemin, il suffit de trier les sommets topo-logiquement avec un parcours en

profondeur d'abord, puis de considérer chaque sommet dans l'ordre ainsi défini et relâcher à chaque fois tous les arcs partant de ce sommet ;

- Si le graphe comporte des circuits, alors
  - ✓ Si on recherche un plus court chemin, alors
    - ❖ Si la fonction coût est monotone croissante (le coût d'un chemin ne peut qu'augmenter lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont positifs et que le coût d'un chemin est égal à la somme des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
    - ❖ Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants) ;
  - ✓ Si on recherche un plus long chemin, alors
    - ❖ Si la fonction coût est monotone décroissante (le coût d'un chemin ne peut que diminuer lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont compris entre 0 et 1 et que le coût d'un chemin est égal au produit des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
    - ❖ Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants).

II.8. Arbres couvrants minimaux

Vous êtes chargés de l'installation du câble dans la région M'sila-Alger. Vous disposez pour cela d'une carte de l'ensemble du réseau routier (le câble est généralement disposé le long des routes). On vous demande de définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale et qu'un certain nombre de lieux soient desservis.

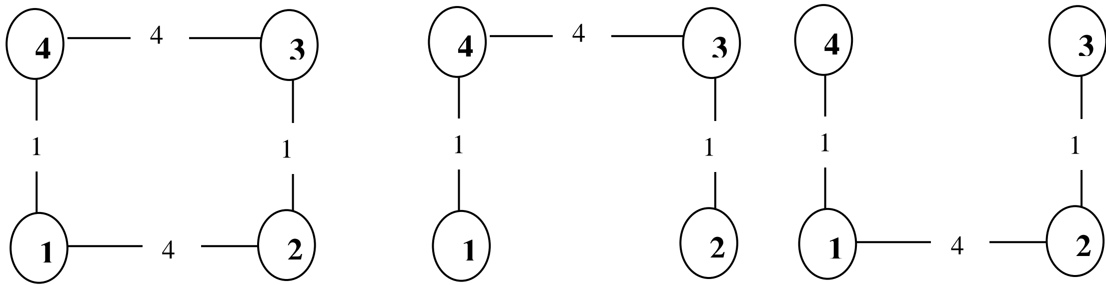
On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe **G = (S, A)**, où **S** associe un sommet à chaque lieu devant être desservi, et **A** contient une arête pour chaque portion de route entre **2** lieux. Ce graphe est valué par une fonction de coûts qui spécifie pour chaque arête $\{s_i; s_j\}$ la longueur de câble nécessaire pour connecter s_i à s_j. Il s'agit alors de trouver un sous-graphe connexe et sans cycle de ce graphe (autrement dit, un arbre) qui recouvre l'ensemble des sommets du graphe. Ce graphe est appelé arbre couvrant. On cherche à minimiser le poids total des arêtes de l'arbre. On dira qu'on cherche l'arbre couvrant minimal **(ACM).**

De façon plus formelle, un arbre couvrant minimal (ACM) (Minimal Spanning Tree / MST) d'un graphe **G = (S, A)** est un graphe partiel **G′= (S, A′)** de **G** tel que **G′** est connexe et sans cycle (**G′** est un arbre), et la somme des coûts des arêtes de **A′** est minimale.

Remarque : il peut exister plusieurs ACM, de même coût, associés à un même graphe.

Exemple

possède les 2 ACMs suivants :



Pour construire un ACM, on adopte une stratégie locale "gloutonne" qui consiste à sélectionner, de pas en pas, une arête devant faire partie de l'ACM. À chaque fois, on choisira la "meilleure" arête selon un certain critère. Il existe deux algorithmes différents suivant une telle stratégie gloutonne et permettant de calculer un ACM à partir d'un graphe : l'algorithme de Kruskal et l'algorithme de Prim. Ces deux algorithmes ont des complexités équivalentes.

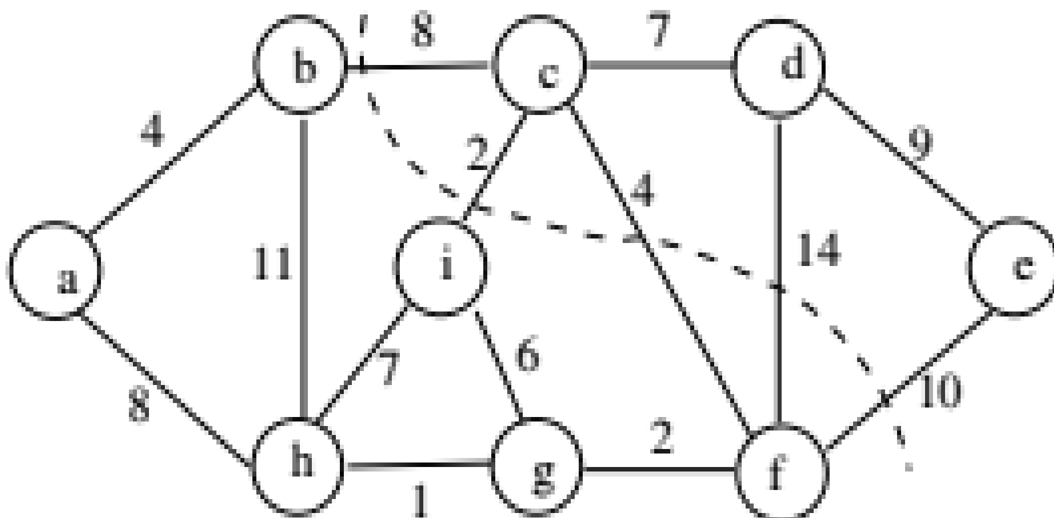
Principe générique

Dans ce chapitre, nous allons étudier deux algorithmes permettant de calculer des MST. Les deux algorithmes fonctionnent selon un principe glouton décrit dans **l'algorithme 6** (principe glouton générique pour calculer un MST) : l'idée est de sélectionner, à chaque itération, une arête de coût minimal traversant une coupure.

Une **coupure** d'un graphe **G = (S, A)** est une partition de l'ensemble des sommets en deux parties **(P, S \ P)**. Une arête **(s_i , s_j) traverse** une coupure **(P, S \ P)** si chaque extrémité de l'arête appartient à une partie différente, i.e., **$s_i \in P$ et $s_j \in S \setminus P$, ou $s_j \in P$ et $s_i \in S \setminus P$**. Une coupure **respecte** un ensemble d'arêtes **E** si aucune arête de **E** n'est traversée par la coupure.

Considérons par exemple le graphe suivant :

La coupure **({a, b, f, g, h, i}, {c, d, e})** est représentée en pointillés et traverse les arêtes **{b, c}, {i, c}, {c, f}, {d, f} et {e, f}**. L'arête de coût minimal traversant cette coupure est **{i, c}.**



Algorithme 6 : Principe glouton générique pour calculer un MST

Fonction MSTgénérique(g, cout)
Entrées : Un graphe **g = (S, A)** et une fonction cout **: A → ℜ**
Postcondition : Retourne un ensemble d'arêtes **E ⊆ A tel que (S, E)** est un MST de **g**

1 : E ← ∅
2 : **Tant que** |E|<|S|-1 **Faire**
3 : Soit **(P, S \ P)** une coupure (quelconque) qui respecte **E**
4 : Ajouter dans **E** une arête de coût minimal traversant la coupure **(P, S \ P)**
5 : **Fin Tant que**
6 : **Retourne** E
7 : **Fin**

Correction de l'algorithme générique : Pour montrer que l'algorithme est correct, nous allons montrer qu'à chaque passage à la **ligne 3**, il existe un MST dont les arêtes sont un sur-ensemble de **E**. Au premier passage, **E** est vide et l'invariant est donc vérifié. Supposons qu'il soit vérifié au $k^{ème}$ passage lorsque **E** contient **k - 1** arêtes, et notons **(S, E′)** un MST contenant **E**. Montrons que l'invariant est vérifié au **(k + 1)**^{ème} passage, après avoir ajouté dans **E** une arête $\{s_i, s_j\}$ de coût minimal traversant une coupure **(P, S \P)** respectant **E**. Nous avons deux cas possibles :

- soit $\{s_i, s_j\} \in E'$, et dans ce cas l'invariant reste trivialement vérifié ;
- soit $\{s_i, s_j\} \notin E'$, et dans ce cas nous devons montrer que l'invariant reste vérifié, c'est-à-dire qu'il existe un autre MST **(S, E″)** contenant $\{s_i, s_j\}$.

Comme **(S, E)** est un arbre couvrant, le graphe $(S, E' \cup \{(s_i, s_j)\})$ contient un et un seul cycle, noté **c**, et ce cycle contient l'arête $\{s_i, s_j\}$. Comme $\{s_i, s_j\}$ traverse la coupure **(P, S \ P)**, il existe nécessairement une autre arête $\{s_k, s_l\}$ de **c** qui traverse également la coupure.

Définissons $E'' = \{\{s_i, s_j\}\} \cup E \setminus \{\{s_k, s_l\}\}$. Comme $\{s_i, s_j\}$ est une arête de coût minimal, parmi l'ensemble des arêtes traversant la coupure, nous avons $cout(s_i, s_j) \leq cout(s_k, s_l)$, et par conséquent **(S, E″) est un MST**.

Mise en œuvre : Toute la difficulté réside dans la recherche efficace, à chaque itération, d'une arête de coût minimal traversant une coupure respectant **E**. Nous allons voir deux façons différentes de faire cela : dans l'algorithme de **Kruskal**, l'ensemble **E** forme une forêt et l'arête sélectionnée est une arête de coût minimal permettant de connecter deux arbres différents; dans l'algorithme de **Prim**, l'ensemble **E** est un arbre et l'arête sélectionnée est une arête de coût minimal connectant un sommet de l'arbre à un sommet n'appartenant pas à l'arbre.

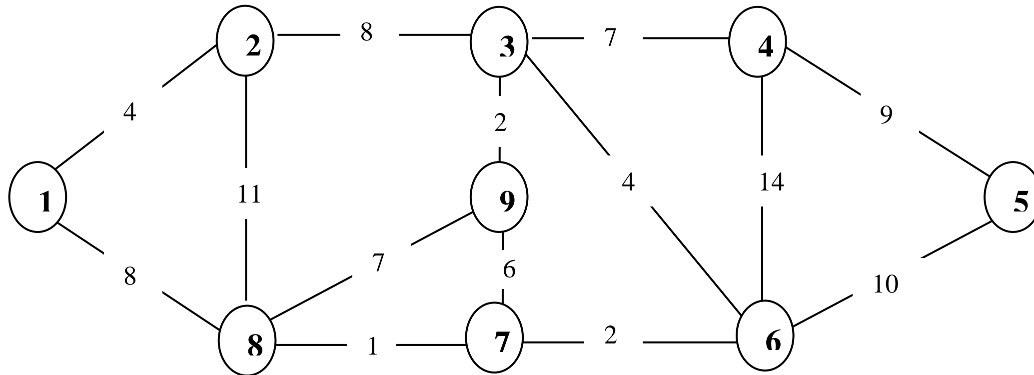
A. Algorithme de Kruskal

Principe :On commence par trier l'ensemble des arêtes du graphe par ordre de coût croissant. On va sélectionner de proche en proche les arêtes devant faire partie de l'ACM. Au début, cet ensemble est vide. On considère ensuite chacune des arêtes du graphe selon l'ordre que l'on vient d'établir (de l'arête de plus faible coût jusqu'à l'arête de plus fort coût). À chaque fois, si l'arête que l'on est

en train de considérer peut-être ajoutée à l'ensemble des arêtes déjà sélectionnées pour l'ACM sans générer de cycle, alors on la sélectionne, sinon on l'abandonne.

Exemple

Considérons par exemple le graphe suivant :

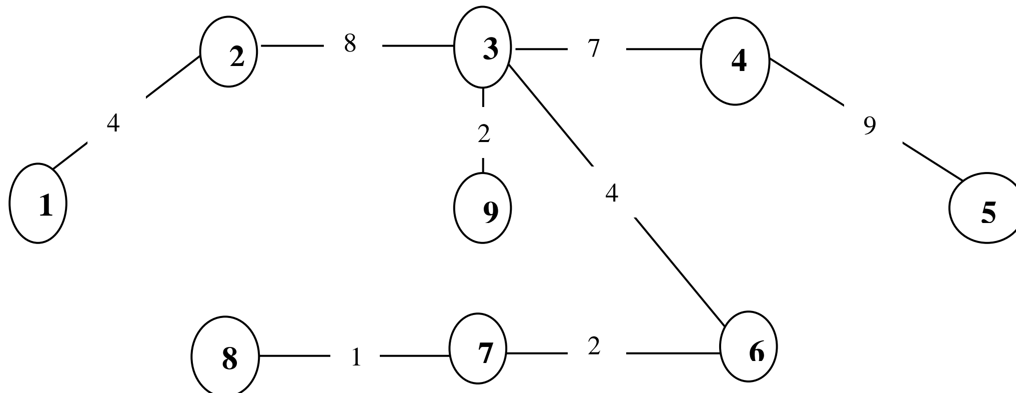


On trie les arêtes du graphe. On obtient l'ordre suivant :

{7, 8} < {3, 9} = {6, 7} < {1, 2} = {3, 6} < {7, 9} < {8, 9} = {3, 4} < {2, 3} = {1, 8} < {4, 5} < {5, 6} < {2, 8} < {4, 6}.

On ajoute alors successivement dans l'ACM les arêtes :

{7, 8}, {3, 9}, {6, 7}, {1, 2}, {3, 6}, {3, 4}, {2, 3}, {4, 5}



Mise en œuvre : La difficulté majeure pour implémenter l'algorithme de Kruskal réside dans la façon de déterminer si l'arête en cours d'examen doit ou non être sélectionnée. Il s'agit de savoir si, en rajoutant l'arête { $\mathbf{s_i}$, $\mathbf{s_j}$ }, on crée un cycle ou non, autrement dit, il s'agit de savoir s'il existe déjà une chaîne entre $\mathbf{s_i}$ et $\mathbf{s_j}$. Afin de pouvoir répondre à cette question, on va partitionner l'ensemble des sommets du graphe en composantes connexes. Pour savoir si on peut sélectionner une arête {$\mathbf{s_i}$, $\mathbf{s_j}$}, il suffira de vérifier que $\mathbf{s_i}$ et $\mathbf{s_j}$ appartiennent à deux composantes connexes différentes. À chaque fois qu'on sélectionnera une arête {$\mathbf{s_i}$, $\mathbf{s_j}$}, on fusionnera les deux composantes connexes correspondantes en une seule.

Représentation d'une composante connexe : Chaque composante connexe étant un arbre, on choisit de représenter les différentes composantes connexes par un vecteur Π de telle sorte que si **$\Pi[\mathbf{s_i}]$ = nil** alors $\mathbf{s_i}$ est la racine d'un arbre, et si $\Pi[\mathbf{s_i}] = \mathbf{s_j}$, alors $\mathbf{s_j}$ est un prédécesseur de $\mathbf{s_i}$ dans l'arbre. Ainsi, pour savoir si deux sommets $\mathbf{s_i}$ et $\mathbf{s_j}$ appartiennent à la même composante connexe, il suffira de remonter dans le vecteur Π de $\mathbf{s_i}$ jusqu'à la racine $\mathbf{r_i}$ de l'arbre contenant $\mathbf{s_i}$, puis de $\mathbf{s_j}$ jusqu'à la racine $\mathbf{r_j}$ de l'arbre contenant $\mathbf{s_j}$, et enfin de comparer $\mathbf{r_i}$ et $\mathbf{r_j}$: si $\mathbf{r_i} = \mathbf{r_j}$ alors les deux sommets $\mathbf{s_i}$ et $\mathbf{s_j}$ appartiennent à la même composante connexe.

Remarque : En pratique, afin d'éviter d'obtenir des arbres déséquilibrés dans **Π**, lorsqu'on fusionne les deux arbres de racines $\mathbf{r_i}$ et $\mathbf{r_j}$, on prendra soin de rattacher l'arbre le moins profond sous l'arbre le plus profond. Pour cela, on gérera un vecteur **prof** qui associe à chaque racine **r** sa profondeur **prof[r]**, et si $\mathbf{prof[r_i] > prof[r_j]}$ alors on fera $\mathbf{\Pi[r_j] \leftarrow r_i}$, sinon on fera $\mathbf{\Pi[r_i] \leftarrow r_j}$. En procédant de cette façon, on garantit que le chemin de n'importe quel nœud de l'arbre jusqu'à sa racine est de longueur inférieure ou égale à $\mathbf{log_2(n)}$ si le graphe comporte **n** sommets.

Algorithme 7 : **Kruskal (S , A , ν , K)**

Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs,
Sorties : **K** ensemble des arêtes de l'ACM ($\mathbf{K \subset A}$)

```
1 :  Pour chaque sommet sᵢ ∈ S Faire
2 :     Π[sᵢ] ← nil
3 :  Fin Pour
4 :  trier les arêtes de A par ordre de valeurs de ν croissant
5 :  K ← ∅
6 :  Tant que |K| < |S| - 1  Faire
7 :     soit {sᵢ, sⱼ} la (|K|+ 1)ieme plus petite arête de A
8 :     /* recherche de la racine rᵢ de la composante connexe de sᵢ */
9 :     rᵢ ← sᵢ
10 :    Tant que Π[rᵢ] ≠ nil Faire
11 :          rᵢ ← Π[rᵢ]
12 :    Fin Tant que
13 :    /* recherche de la racine rⱼ de la composante connexe de sⱼ */
14 :    rⱼ ← sⱼ
15 :    Tant que Π[rⱼ] ≠ nil Faire
16 :          rⱼ ← Π[rⱼ]
17 :    Fin Tant que
18 :    Si rᵢ ≠ rⱼ Alors
19 :         /* on ajoute {sᵢ, sⱼ} à l'ACM */
20 :          K ← K ∪ {{sᵢ, sⱼ}}
21 :          /* on fusionne les deux composantes connexes */
22 :          Π[rᵢ] ← rⱼ
23 :    Fin Si
24 :  Fin Tant que
25 : Fin
```

Complexité : On considère un graphe non orienté de **n** sommets et **m** arêtes. Pour trier l'ensemble des arêtes, avec une procédure de tri efficace (e.g., quicksort, mergesort ou heapsort), il faut exécuter de l'ordre de **m∗log(m)** opérations. On passe ensuite, dans le pire des cas, **m** fois dans la boucle "tant que **|K| < |S| - 1**" (une fois pour chaque arête $\mathbf{\{s_i; s_j\}}$). À chaque fois, il faut remonter des sommets $\mathbf{s_i}$ et $\mathbf{s_j}$ jusqu'aux racines des arbres correspondants. En gérant astucieusement la représentation des arbres par **Π**, on a vu que cette opération pouvait être faite en **O(log(n)).** Par conséquent, on a une complexité totale en **O(m ∗ log(m))** (sous réserve d'utiliser une représentation par listes d'adjacence).

B. Algorithme de Prim

L'algorithme de Prim, décrit dans l'algorithme 7 est également un cas particulier de l'algorithme 8. Cette fois ci, **K** est un ensemble d'arêtes connexes formant un arbre dont la racine est un sommet s_0 choisi arbitrairement au début de l'algorithme. La coupure considérée à chaque itération est **(P, S \ P)** où **P** est l'ensemble des sommets qui appartiennent à l'arbre de racine s_0 et contenant les arêtes de K. Cette coupure respecte les arêtes de **K** puisque tous les sommets appartenant à une arête de **K** appartiennent à **P** . Pour trouver efficacement l'arête de coût minimal traversant la coupure, l'algorithme maintient deux tableaux **c** et **π** tels que, pour chaque sommet $s_i \in P$ pour lequel il existe une arête $\{s_i , s_j\}$ traversant la coupure,

- $\{s_i; \pi[s_i]\}$ est la plus petite arête partant de s_i et traversant la coupure **(P, S \ P)**;
- $c[s_i] = cout(s_i; \pi[s_i])$.

Les éléments du tableau **c** sont initialisés à **1**, sauf pour les sommets s_i adjacents à s_0 pour lesquels $c[s_i]$ est initialisé à $cout(s_0 , s_i)$. À chaque fois qu'un sommet s_i est ajouté à **P**, chaque arête connectant s_i à un sommet s_j de **S \ P** est utilisée pour mettre à jour $c[s_j]$ et $\pi[s_j]$ dans le cas où le coût de $\{ s_i , s_j \}$ est inférieur à la valeur courante de $c[s_j]$.

Algorithme 8 : Prim (S , A , ν , K)

Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs,
Sorties : **K** ensemble des arêtes de l'ACM ($K \subset A$)

```
1 :  Soit s0 un sommet de S choisi arbitrairement
2 :  P ← {s0}
3 :  K ← ∅
4 :  Pour chaque sommet si ∈ S Faire
5 :     Si si ∈ adj(s0) Alors
6 :         Π[si] ← s0
7 :         c[si] ← ν ( s0, si)
8 :     Sinon
9 :         Π[si] ← nil
10 :        c[si] ← ∞
11 :    Fin Si
12 :  Fin Pour
13 :  Tant que  P ≠ S  Faire
14 :    Ajouter dans P le sommet si ∈ S \ P ayant la plus petite valeur de c
15 :    Ajouter { si , Π [si]} à K
16 :    Pour  chaque sommet sj  ∈ adj(si) Faire
17 :        Si (sj ∉ P) et  (ν ( si, sj) < c[sj]) Alors
18 :            Π[sj] ← si
19 :            c[sj] ← ν ( si, sj)
20 :        Fin Si
21 :    Fin Pour
22 :  Fin Tant que
23 :  Retourne K
24 : Fin
```

Complexité : Soient n et p le nombre de sommets et arêtes, respectivement. L'algorithme passe **n - 1** fois dans la boucle **lignes 13 à 22** (initialement **P = {s_0}**, un sommet est ajouté à **P** à chaque passage, et l'itération s'arrête lorsque **P = S**). À chaque passage, il faut chercher le sommet de **S\P** ayant la plus petite valeur de c puis parcourir toutes les arêtes adjacentes à ce sommet. Si les sommets de **S\P** sont mémorisés dans un tableau ou une liste, la complexité est **$O(n^2)$.** Cette complexité peut être améliorée en utilisant une file de priorité, implémentée par un tas binaire, pour gérer l'ensemble **S\P**. Dans ce cas, la recherche du plus petit élément de **S\P** est faite en temps constant. En revanche, à chaque fois qu'une valeur du tableau **c** est diminuée, la mise à jour du tas est en O(log n). Comme il y a au plus **p** mises à jour de **c** (une par arête), la complexité de Prim devient **O(p log n).**

II.9. Quelques problèmes courants de théorie des graphes

Les notions précédentes jouent un rôle fondamental dans un certain nombre de problèmes ≪ type ≫ de théorie des graphes. Nous allons en donner quelques-uns en exemple.

Définition 2.11 (coloriage d'un graphe)

Soit **G = (S, A)** un graphe simple non-orienté, colorier le graphe **G** consiste à assigner une couleur (ou un nombre) à chaque sommet du graphe de telle sorte que deux sommets reliés par un arc/arrêté aient des couleurs différentes en utilisant le moins de couleurs possibles. Le nombre minimal de couleur est appelé **γ(G)** =nombre chromatique du graphe **G**.

De même colorier les arêtes du graphe **G** consiste à assigner une couleur (ou un nombre) à chaque arête du graphe de telle sorte que deux arêtes reliées à un même sommet aient des couleurs différentes en utilisant le moins de couleurs possibles. Le nombre minimal de couleur est appelé **γ'(G)** =indice chromatique du graphe **G**.

L'algorithme ≪ glouton ≫ est le plus simple pour colorier un graphe :

```
1 :  Fonction G = Coloriage(G)
2 :      couleur courante = 1
3 :      Pour tout x sommet de G Faire
4 :              V = liste des voisins de x
5 :              couleur = plus petite couleur non encore utilisée dans V
6 :              Si couleur ≤ couleur courante Alors
7 :                      colorier x avec cette couleur
8 :              Sinon
9 :                      incrémenter la couleur courante et colorier x avec cette couleur
10 :             Fin Si
11 :     Fin Pour
12 : Fin
```

Il existe un autre algorithme intéressant pour colorier un graphe : l'algorithme de Welsh-Powell. Cet algorithme est plus compliqué mais souvent moins long à mettre en œuvre.

Attention, les deux algorithmes ne donnent pas toujours le nombre minimal de couleurs !

L'algorithme Glouton peut être amélioré en traitant les sommets dans l'ordre décroissant de leur degré (comme dans l'algorithme de Welsh-Powell). Enfin notons que la structure du graphe impose certaines contraintes sur le nombre chromatique :

- les sommets d'une même clique doivent être coloriés d'une couleur différente
- les sommets d'un même stable peuvent tous être coloriés de la même couleur

cela permet d'encadrer le nombre Chromatique de G :

- obtenir un coloriage à **k** couleurs permet d'affirmer que **γ(G) ≤ k**
- trouver une clique à **k** sommets permet d'affirmer que **γ(G) ≥ k**
- Le coloriage d'un graphe permet de résoudre de nombreux problèmes d'incompatibilité.

```
1 :  Fonction G = Welsh(G)
2 :      L = liste des sommets classés dans l'ordre décroissant de leur degré
3 :      couleur courante = 0
4 :      Tant que L ≠ Ø Faire
5 :          incrémenter la couleur courante
6 :          Colorier s le premier sommet de L avec la couleur courante
7 :          éliminer s de L
8 :          V = liste des voisins de s
9 :          Pour tout x dans L Faire
10 :             Si x ∉ V Alors 4
11 :                 colorier x avec la couleur courante
12 :                 ajouter les voisins de x à V
13 :             Fin Si
14 :         Fin Pour
15 :         éliminer les sommets coloriés de L
16 :     Fin Tant que
17 : Fin
```