LI311 - Cours
Algorithmique

BARON Benjamin

Chapitre 1

Graphes

1.1 Graphe non orienté

Définition (Graphe non orienté)**.** Un graphe non orienté G est défini par un couple $G = (S, A)$ où :
- S (ou V – *vertices*) est un ensemble de sommets ;
- A (ou E – *edges*) un ensemble d'arêtes.

Définition (Sommets adjacents)**.** Ensemble des sommets $\Gamma(u)$ adjacents à u :

$$\forall u \in V,\ \Gamma(u) = v \in V,\ u,\ v \in E$$

Degré d'un sommet u : $d(u) = |\Gamma(u)|$
Toute arête $e = \{u, v\} \in S$ est incidente à u et v.

Définition (Sous-graphe)**.** Un sous-graphe de $G = (S_G, A_G)$ est un graphe

$$H = (S_H, A_H) \text{ tel que } S_H \subset S_G \text{ et } A_H \subseteq A_G$$

Définition (sous-graphe induit)**.** Le sous-graphe induit par un ensemble de sommets $V' \subset S_G$ est le sous-graphe $G' = (S', A')$ avec

$$E' = \{e = \{u, v\} \in A,\ u, v \in S'\}$$

Définition (Chaînes)**.** Une *chaîne* est une séquence de sommets et d'arêtes :

$$\nu = v_1 e_1 v_2 e_2 \dots v_n e_n v_{n+1}$$

- $v_i \in S$ pour $i \in \{1, \dots, n+1\}$
- $e_i = \{v_i, v_i + 1\} \in A$ pour $i \in \{1, \dots, n\}$

Une *chaîne élémentaire* est une chaîne qui passe pas deux fois par le même sommet.
Un *cycle* est une chaîne ν telle que $v_{n+1} = v_1$.

Définition (Graphe connexe)**.** Un graphe *connexe* est tel que pour tout couple $(u, v) \in V^2$, il existe une chaîne entre u et v.

1.2 Graphe orienté

Définition (Graphe orienté)**.** Un graphe orienté G est défini par un couple $G = (S, A)$ où :
- S est un ensemble de sommets ;
- A est un ensemble d'arcs.

Pour tout sommet $u \in A$, on a :
- Successeurs de u : $\Gamma^+(u) = \{v \in S,\ (u, v) \in A\}$
- Prédécesseurs de u : $\Gamma^-(u) = \{v \in S,\ (v, u) \in A\}$

- Demi-degré entrant de u : $d^+(u) = |\Gamma^+(u)|$
- Demi-degré sortant de u : $d^-(u) = |\Gamma^-(u)|$
- Degré de u : $d(u) = d^+(u) + d^-(u)$

Définition (Chemins)**.** Un *chemin* est une séquence de sommets et d'arcs :

$$\nu = v_1 e_1 v_2 e_2 \dots v_n e_n v_{n+1}$$

- $v_i \in S$ pour $i \in \{1, \dots, n+1\}$
- $e_i = (v_i, v_i + 1) \in A$ pour $i \in \{1, \dots, n\}$

Un *chemin élémentaire* est un chemin qui ne passe pas deux fois par le même sommet.

Un *circuit* est un chemin ν tel que $v_{n+1} = v_1$.

Définition (Composante fortement connexe)**.** Une composante fortement connexe d'un graphe orienté G est un sous-graphe maximal $\hat{G}$ de G tel que pour tout couple $(u, v) \in \hat{G}$, il existe un chemin de u à v **et** un chemin de v à u.

Un graphe est dit fortement connexe s'il est formé d'une seule composante fortement connexe. De manière générale, un graphe se décompose de manière unique comme union de composantes fortement connexes disjointes.

1.3 Arbres

Définition (Arbre)**.** Soit $T = (S, S)$ un graphe non orienté. T est un arbre si T est connexe sans cycle.

Les propriétés suivantes sont équivalentes :

- T est un arbre.
- T est minimal connexe : $\forall e \in A$, $T = (S, A \backslash \{e\})$ n'est pas connexe.
- T est maximal acyclique : $\forall \{x, y\} \in S$ non adjacents dans T, $T' = (S, A \cup \{x, y\})$ contient un cycle.
- Entre deux sommets quelconques, il existe une chaîne unique.

Définition (Arborescence)**.** Une arborescence est un graphe orienté $G = (S_G, A_G)$ construite à partir d'un arbre $T = (S, A)$:

- $S_G = S$
- Soit $r \in S$. Les arcs de A_G correspondent aux arêtes de A orientées du sommet r vers les feuilles.

r est la racine de $G = (S_G, A_G)$. C'est l'unique sommet sans prédécesseur.

1.4 Propriétés sur les graphes

1.4.1 Représentation en mémoire des graphes

Définition (Matrice sommet-arête $G = (S, A)$ non orienté)**.** Pour tout couple $(i, j) \in V \times A$,

- $M[i, j] \in \{0, 1\}$
- $M[i, j] = 1$ ssi i incidente à j, 0 sinon.

Complexité-mémoire : $\Theta(|S| \times |A|)$

Définition (Liste d'adjacence $G = (S, A)$ non orienté)**.** Pour $i \in S$, $L[i]$ est la liste des sommets adjacents à i.

Complexité-mémoire : $\Theta(\max(|S|, |A|))$

1.4.2 Propriétés

Proposition. Soit G un graphe non orienté, alors :

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

Proposition. Tout graphe à $n \geqslant 2$ sommets possède au moins deux sommets de même degré.

Proposition. Soit $G = (S, A)$ un graphe orienté, alors :

$$\sum_{v \in S} d^+(v) = \sum_{v \in S} d^-(v) = |A|$$

Lemme (Kœnig)**.** S'il existe une chaîne entre deux sommets u et v, alors il existe une chaîne élémentaire entre u et v (Toute chaîne peut se résumer à une chaîne élémentaire).

Proposition. Soit $G = (S, A)$ un graphe non orienté tel que $\forall v \in S,\ d(v) \geqslant 2$, alors G contient un cycle.

Proposition. Deux chaînes de longueur maximale dans un graphe connexe ont au moins un sommet en commun.

Proposition. Soit G un graphe à n sommets, m arêtes et k composantes connexes, alors :

$$n - k \leqslant m \leqslant \frac{1}{2}(n-k)(n-k+1)$$

Soit $T = (S, A)$ un graphe non orienté. On a les propriétés :
- Si $|A| \geqslant |S|$, alors T contient un cycle.
- Si $|A| < |S| - 1$, alors T n'est pas connexe.
- Si T est un arbre, alors $|A| = |S| - 1$.

Chapitre 2

Parcours de graphes

2.1 Parcours de graphes non orientés

2.1.1 Parcours générique

Définition (Bordure)**.** Soit $G = (S, A)$ un graphe non orienté (n sommets, m arcs).

La bordure de $T \subset S$, $B(T, G)$ est le sous-ensemble des sommets de S
T dont au moins un voisin est dans T.

Notation. Notations associées à une liste $L = (s_1, s_2, \ldots, s_p)$ de sommets distincts :
- $L[i \ldots j]$: la sous-liste $(s_i, \ldots, s_j)$, $1 \leqslant i \leqslant j \leqslant p$
- $G[i \ldots j]$: le sous-graphe de G induit par $\{s_i, \ldots, s_j\}$, $1 \leqslant i \leqslant j \leqslant p$

Définition (Parcours d'un graphe non orienté $G = (S, A)$)**.** Liste $L = (s_1, s_2, \ldots, s_p)$ des n sommets de G telle que pour tout $i \in \{2, \ldots, n\}$, le sommet s_i appartient à $B(L[1 \ldots i-1], G)$ si cette bordure n'est pas vide.

Si $B(L[1 \ldots i-1], G)$ est vide, alors s_i est un point de régénération de L.
Par convention, s_1 est un point de régénération.

Remarque. Etape i du parcours : ajouter s_i (nouveau sommet visité) à la sous-liste $L[1 \ldots i-1]$ des sommets déjà visités.

Un sommet de $L[1 \ldots i]$ est *ouvert* s'il possède au moins un voisin non visité (ie. dans $L[i+1 \ldots n]$). Dans le cas contraire, il est *fermé*.

Proposition (Décomposition d'un parcours sur les composantes connexes)**.** Soit $L = (s_1, s_2, \ldots, s_p)$ un parcours de G ; et soient $(i_1, \ldots, i_r)$ les indices des points de régénération de L.
- $G[i_1 \ldots i_2 - 1]$, $G[i_2 \ldots i_3]$, $\ldots$, $G[i_r \ldots n]$ sont les sous-graphes induits par les composantes connexes de G.
- $L[i_1 \ldots i_2 - 1]$, $L[i_2 \ldots i_3]$, $\ldots$, $L[i_r \ldots n]$ sont les parcours des sous-graphes induits par les composantes connexes de G.

Définition (Graphe de choix)**.** Soit $L = (s_1, s_2, \ldots, s_p)$ un parcours de G connexe. Pour $i \geqslant 2$, chaque nouveau sommet s_i visité possède au moins un voisin déjà visité (graphe connexe).

Choisissons un tel voisin, noté $\text{père}_L(s_i)$.

Le graphe partiel F de G constitué des $n-1$ arêtes $\{s_i, \text{père}_L(s_i)\}$ est un graphe de choix de L.

Proposition. Tout graphe de choix F d'un parcours L de G connexe est un arbre couvrant de G.

2.1.2 Parcours en largeur

Définition (Parcours en largeur)**.** Soit $L = (s_1, s_2, \ldots, s_p)$ un parcours de G.

L est un parcours en largeur de G si tout sommet visité s_i, qui n'est pas un point de régénération, est voisin du **premier sommet visité ouvert** de $L[1 \ldots i-1]$.

Algorithme 1 : Parcours en largeur

Données : $G = (S, A)$ un graphe, s premier sommet, F une file, L une liste, VNV voisins non visités

Résultat : L : parcours en largeur de G

début

$L \longleftarrow \{s\}$

allouer et initialiser la file F

pour $x \in S$ **faire** $VNV[x] \longleftarrow d_G(x)$

pour $x \in adj[s]$ **faire** $VNV[x] \longleftarrow VNV[x] - 1$

si $VNV[s] > 0$ **alors** `Enfiler`(s, F)

tant que $F \neq \varnothing$ **faire**

$y \longleftarrow tête[F]$

tant que *VNV[y]=0* **faire** `Defiler`(F)

$z \longleftarrow$ voisin non visité de F dans L

$L \longleftarrow L \cup \{z\}$

pour $t \in adj[z]$ **faire** $VNV[t] \longleftarrow VNV[t] - 1$

si $VNV[z] > 0$ **alors** `Enfiler`(z, F)

retourner L

Remarque. Le sommet père$_L(s_i)$ est par convention le premier sommet visité ouvert de $L[1 \dots i-1]$.

Complexité (Parcours largeur)**.** La complexité de l'algorithme de parcours en largeur est $O(n+m)$

Proposition (Décomposition sur les composantes connexes)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours en largeur de G et soient $(i_1, \dots, i_r)$ les indices des points de régénération de L.

- $G[i_1 \dots i_2 - 1],\ G[i_2 \dots i_3], \dots, G[i_r \dots n]$ sont les sous-graphes induits par les composantes connexes de G.
- $L[i_1 \dots i_2 - 1],\ L[i_2 \dots i_3], \dots, L[i_r \dots n]$ sont des parcours en largeur des sous-graphes induits par les composantes connexes de G.

Proposition (Distances minimales à l'origine du parcours)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours en largeur de G connexe.

Soient s l'origine s_1 du parcours et $d(x)$ la longueur minimum d'une chaîne de s à x.

On a alors :

- $d(s_1) \leqslant d(s_2) \leqslant \dots \leqslant d(s_n)$
- $\forall i \in 2, \dots, n,\ d(\text{père}_L(s_i)) + 1$

2.1.3 Parcours en profondeur

Définition (Parcours en profondeur)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours de G.

L est un parcours en profondeur de G si tout sommet visité s_i, qui n'est pas un point de régénération, est voisin du **dernier sommet visité ouvert** de $L[1 \dots i-1]$.

Remarque. Le sommet père$_L(s_i)$ est par convention le dernier sommet visité ouvert de $L[1 \dots i-1]$.

Complexité (Parcours profondeur)**.** La complexité de l'algorithme de parcours en profondeur est $O(n+m)$

Proposition (Décomposition sur les composantes connexes de G)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours en profondeur de G et soient $(i_1, \dots, i_r)$ les indices des points de régénération de L.

- $G[i_1 \dots i_2 - 1],\ G[i_2 \dots i_3], \dots, G[i_r \dots n]$ sont les sous-graphes induits par les composantes connexes de G.
- $L[i_1 \dots i_2 - 1],\ L[i_2 \dots i_3], \dots, L[i_r \dots n]$ sont des parcours en profondeur des sous-graphes induits par les composantes connexes de G.

Proposition (Liste suffixe)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours en profondeur de G et soit F sa foret sous-jacente.

Pour tout $1 \leqslant i \leqslant n$, la liste suffixe $L[i \dots n]$ est un parcours en profondeur du sous-graphe $G[i \dots n]$; et sa forêt sous-jacente est le sous-graphe de F induit par $\{s_i, \dots, n\}$.

Proposition (Absence d'arêtes transverses)**.** Soit $L = (s_1, s_2, \dots, s_p)$ un parcours en profondeur de G et soit F sa forêt sous-jacente.

Si $\{s_i, s_j\},\ i < j$ est une arête non dans F, alors s_j est un descendant de s_i dans F.

2.2 Parcours de graphes orientés

Soit $G = (S, A)$ un graphe orienté avec n sommets et m arcs.

Algorithme 2 : Parcours en profondeur

Données : $G = (S, A)$ un graphe, s premier sommet, P une pile, L une liste, VNV voisins non visités

Résultat : L : parcours en profondeur de G

début

```
L ⟵ {s}
allouer et initialiser la pile P
pour x ∈ S faire VNV[x] ⟵ d_G(x)

pour x ∈ adj[s] faire VNV[x] ⟵ VNV[x] − 1

si VNV[s] > 0 alors Empiler(s, P)

tant que P ≠ ∅ faire
    y ⟵ tête[P]
    tant que VNV[y]=0 faire Depiler(P)

    z ⟵ voisin non visité de P dans L
    L ⟵ L ∪ {z}
    pour t ∈ adj[z] faire VNV[t] ⟵ VNV[t] − 1

    si VNV[z] > 0 alors Empiler(z, P)

    retourner L
```

Procédure ParcoursProfRec

Données : $G = (S, A)$ un graphe, s premier sommet, L liste des sommets visités

début

```
L ⟵ L ∪ {s}
tant que z ∈ adj[s], z ∉ L faire
    ParcoursProfRec(G, z, L)
```

Définition (Bordure de $T \subset S$)**.** $B(T, G)$ est le sous-ensemble des sommets de $S \backslash T$ dont au moins un prédécesseur est dans T.

Définition (Parcours d'un graphe $G = (S, A)$)**.** Liste $L = (s_1, s_2, \ldots, s_n)$ des n sommets de G telle que pour tout $i \in \{1, \ldots, n\}$, $s_i \in B(L[1 \ldots i-1], G)$ si $B(L[1 \ldots i-1], G) \neq \varnothing$.

Remarque. Soit $L = (s_1, \ldots, s_n)$ un parcours de G. Si $B(L[1 \ldots i-1], G) = \varnothing$, alors s_i est un point de régénération de L.

Par convention, s_1 est un point de régénération.

Théorème (Points de régénération et décomposition)**.** Soit $L = (s_1, s_2, \ldots, s_n)$ un parcours de G. Soient $(i_1, \ldots, i_r)$ les indices des points de régénération de L. Considérons la décomposition de L associée :

$$L = L[i_1 \ldots i_2 - 1].L[i_2 \ldots i_3 - 1]. \ldots .L[i - r \ldots n]$$

- Pour tout $k \in \{1 \ldots r\}$, le $k^{\text{ième}}$ point de régénération est une racine du sous-graphe $G[i_k \ldots i_{k+1} - 1]$.
- $L[i_1 \ldots i_2 - 1]$, $L[i_2 \ldots i_3 - 1]$, $\ldots$, $L[i_r \ldots n]$ sont des parcours des sous-graphes respectifs $G[i_1 \ldots i_2 - 1]$, $G[i_2 \ldots i_3 - 1]$, $\ldots$, $G[i_r \ldots n]$

Lemme. Si $x \in L[i_k \ldots i_{k+1} - 1]$, alors tout successeur de x appartient à $L[1..i_{k+1} - 1]$.

Définition. Soit $L = (s_1, s_2, \ldots, s_n)$ un parcours de G. Soit $(i_1, \ldots, i_r)$ les indices des points de régénération de L. Soit F la forêt couvrante de G associée à un graphe de choix de L.

On établit la topologie suivante des arcs de G :

- Un arc de la forêt est un arc de F ;
- Un arc (x, y) est **avant** si y est un descendant de x dans F ;
- Un arc (x, y) est **arrière** si x est un descendant de y dans F ;
- Un arc (x, y) est **transverse** si x et y ont un ancêtre commun dans F distinct de x ou de y ;
- Un arc (x, y) est un **liaison** si x et y ne sont pas dans la même arborescence de la forêt F.

2.2.1 Parcours en largeur

Définition (Parcours en largeur)**.** Soit $L = (s_1, s_2, \ldots, s_n)$ un parcours de G.

L est un parcours en largeur de G si pour tout sommet visité s_i, qui n'est pas un point de régénération, est successeur du premier sommet visité ouvert de $L[1 \ldots i-1]$.

Théorème (Décomposition)**.** Soit $L = (s_1, \ldots, s_n)$ un parcours en largeur de G. Soient $(i_1, \ldots, i_r)$ les indices des points de régénération de L.

$L[i_1 \ldots i_2 - 1]$, $L[i_2 \ldots i_3 - 1, \ldots, L[i_R \ldots n]]$ sont les parcours en largeur des sous-graphes $G[i_1 \ldots i_2 - 1]$, $G[i_2 \ldots i_3 - 1, \ldots, G[i_R \ldots n]]$.

Théorème (Plus courts chemins issus de l'origine du parcours)**.** Soit s une racine de G. Soit $L = (s_1, \ldots, s_n)$ un parcours en largeur de G à partir de s (F ne contient alors qu'une seule arborescence).

Notons $d(x)$ la longueur minimum d'un chemin de s à x.

- $d(s_1) \leqslant \cdots \leqslant d(s_n)$;
- Pour $i \in \{2, \ldots, n\}$, $d(\text{père}_L(s_i)) + 1$

2.2.2 Parcours en profondeur

Définition (Parcours en profondeur)**.** Soit $L = (s_1, s_2, \ldots, s_n)$ un parcours de G.

L est un parcours en profondeur de G si tout sommet visité s_i, qui n'est pas un point de régénération, est successeur du dernier sommet visité ouvert de $L[1 \ldots i-1]$.

Théorème (Décomposition)**.** Soit $L = (s_1, \ldots, s_n)$ un parcours en profondeur de G. Soient $(i_1, \ldots, i_r)$ les indices des points de régénération de L.

$L[i_1 \ldots i_2-1]$, $L[i_2 \ldots i_3-1, \ldots, L[i_R \ldots n]]$ sont les parcours en profondeur des sous-graphes $G[i_1 \ldots i_2-1]$, $G[i_2 \ldots i_3 - 1, \ldots, G[i_R \ldots n]]$.

Proposition (Liste suffixe)**.** Soit $L = (s_1, \ldots, s_n)$ un parcours en profondeur de G, et soit F sa forêt sous-jacente.

Pour tout $1 \leqslant i \leqslant n$, la liste suffixe $L[i \ldots n]$ est un parcours en profondeur du sous-graphe $G[i \ldots n]$ et sa forêt sous-jacente est le sous-graphe de F induit par $\{s_i, \ldots, s_n\}$.

Lemme (Arcs transverses)**.** Soit $L = (s_1, s_2, \ldots, s_n)$ un parcours en profondeur de G et soit F sa forêt sous-jacente.

Si (s_i, s_j) est un arc transverse, alors on a $i > j$.

Lemme (Sommets accessibles)**.** Soit $L = (s_1, \ldots, s_n)$ un parcours en profondeur de G, et soit F sa forêt sous-jacente.

Si, après visite de s_i, il existe un chemin de sommets non visités (sauf s_i) de s_i à x, alors x est un descendant de s_i dans F.

Proposition (Existence de circuits)**.** Soit $L = (s_1, \ldots, s_n)$ un parcours en profondeur et soit F sa forêt sous-jacente.

G est sans circuit si, et seulement si il n'existe pas d'arc arrière pour L.

Remarque. Algorithme pour tester l'existence de circuit dans G (variante d'un parcours en profondeur) :
On utilise une pile pour maintenir le chemin de F de la racine au dernier sommet visité.
Pour chaque nouveau sommet, on teste si l'un de ses successeur est dans la pile.
Si oui, c'est donc un arc arrière et il existe un circuit.

Chapitre 3

Chemins de coût minimum

Soit $G = (S, A)$ un graphe orienté à n sommets et m arcs ; soit $c\colon A \longrightarrow \mathbb{R}$ une fonction coût sur les arcs de G. Soient un sommet origine (source) s et un sommet destination p.

Définition (Coût d'un chemin)**.** Soit $l = (a_1, a_2, \ldots, a_p)$ un chemin. Le coût de l, noté $c(l)$ est égal à la somme $\sum_{k\in\{1,\ldots,p\}} c(a_k)$.

Proposition. Il existe un chemin de s à p de coût minimum si, et seulement si p est un descendant de s dans G.

Définition (Circuit absorbant)**.** Un circuit est absorbant si son coût est strictement négatif.

Proposition. Soit p un descendant de s dans G.

Si G ne possède pas de circuit absorbant, alors il existe un chemin de coût minimum de s à p qui est élémentaire.

Proposition (Arborescence des chemins de coût minimum)**.** Soit $G = (S, A)$ un graphe et soit $c\colon A \longrightarrow \mathbb{R}$ une fonction coût appliquée sur les arcs de G. Soit s un sommet de G tel que s est une racine de G. Supposons que G ne possède pas de circuit absorbant.

G possède une arborescence couvrante H de racine s telle que pour tout sommet x de G, le chemin de s à x dans H est un chemin de coût minimum de s à x dans G.

H est appelée arborescence des chemins de coût minimum d'origine s.

3.1 Algorithme de Bellman-Ford

Algorithme 3 : Algorithme de Bellman-Ford

début

$d^0(s) = 0$

pour tous les *sommets* $x \neq s$ **faire** $d^0(x) \longleftarrow +\infty$

pour $k = 1$ *à* n **faire**

$d^k(s) = 0$

pour tous les *sommets* $x \neq s$ **faire** $d^k(x) \longleftarrow \min\{d^{k-1}(a^- + c(a) \mid a^+ = x)$

si $\forall x \in S,\ d^k(x) = d^{k-1}$ **alors** stop : optimum

si $\exists x \in S,\ d^n(x) \neq d^{n-1}(x)$ **alors**

Il n'existe pas de plus court chemin entre s et x

Complexité (Algorithme de Bellman-Ford)**.** Complexité temporelle en $\mathcal{O}(mn)$.
Complexité mémoire en $\mathcal{O}(n^2)$ (réduite en $\mathcal{O}(n)$ si l'on ne stocke que les valeurs de $d^k(x)$ et $d^{k-1}(x)$).

3.2 Algorithme de Bellman

Algorithme 4 : Algorithme de Bellman

Données : $G = (S, A)$ un graphe sans circuits ; s : sommet racine de G ; c: $A \longrightarrow \mathbb{R}$ fonction coût sur les arcs

début

```
Liste topologique L = (s_1, ..., s_n) des sommets de G telle que s = s_1
d(s_1) ⟵ 0
ouvrir(s_1)
pour k = 2 à n faire  d(s_k) ⟵ +∞

pour k = 1 à n faire
    pour tous les successeurs y de s_k faire
        si d(y) > d(s_k) + c(x, y) alors
            d(y) ⟵ d(s_k) + c(s_k, y)
            ouvrir(y)
    fermer(s_k)
```

Remarque. Un sommet x est dit fermé si son évaluation $d(x)$ décroit. Il est dit fermé à l'issue de la seconde boucle.

Complexité (Algorithme de Bellman)**.** Complexité temporelle en $\mathcal{O}(n + m)$

3.3 Algorithme de Dijkstra

Algorithme 5 : Algorithme de Dijkstra

Données : $G = (S, A)$ un graphe orienté ; s : sommet racine de G ; c: $A \longrightarrow \mathbb{R}$ fonction coût sur les arcs telle que $\forall (x, y) \in A,\ c(x, y) \geqslant 0$

début

```
d(s_1) ⟵ 0
ouvrir(s_1)
pour k = 2 à n faire  d(s_k) ⟵ +∞

pour k = 1 à n faire
    x ⟵ sommet ouvert tel que d(x) minimum
    pour tous les successeurs y de x faire
        si d(y) > d(x) + c(x, y) alors
            d(y) ⟵ d(x) + c(x, y)
            ouvrir(y)
    fermer(s_k)
```

Complexité (Algorithme de Dijkstra)**.** Complexité temporelle : $\mathcal{O}(nm)$
Complexité temporelle en utilisant un tas : $\mathcal{O}((n + m)\log(n))$

Chapitre 4

Arbre couvrant de coût minimum

Définition (Arbre couvrant)**.** Soit $G = (S, A)$ un graphe non orienté connexe avec n sommets et m arêtes.

Un arbre couvrant de G est un graphe partiel de G qui est un arbre.

Définition (Coût d'un arbre couvrant H)**.** Soit $g = (S, A)$ un graphe connexe non orienté. Soit H un arbre courant de G et soit $c\colon A \longrightarrow \mathbb{R}$ une fonction coût.

Le coût de H, noté $c(H)$ est la sommet $\sum_{e \in H} c(e)$ de ses arêtes.

4.1 Algorithme générique

Définition (Approximant)**.** Soit (X, Y) un couple de parties disjointes de A.

(X, Y) est un approxiamant s'il existe un arbre optimal $H*$ tel que $X \subset H*$ et $Y \subset A \backslash H*$.

Remarque.

- $(\varnothing, \varnothing)$ est un approximant.
- Le graphe (S, X) est une forêt couvrante de G.
- Si $|X| = n - 1$, alors X est un arbre optimal.
- Si $|Y| = m - (n - 1)$, alors $A \backslash Y$ est un arbre optimal.

Définition (Cocycle)**.** Soit $G = (S, A)$ un graphe connexe non orienté et soit $S' \subset S$.

Le cocycle $\omega(S')$ associé à S est l'ensemble des arêtes $\{x, y\} \in A$ telles que

$$x \in S' \text{ et } y \notin S' \text{ ou } x \notin S' \text{ et } y \in S'$$

Théorème (Règle des cocycles)**.** Soit (X, Y) un approximant et soit C un cocycle tel que pour toute arête $a = \{x, y\} \in A,\ a \in Y$ ou $a \in Z = A \backslash (X \cup Y)$.

Soit e une arête telle que $e \in Z$ de coût minimum. Alors $(X \cup \{e\}, Y)$ est un approximant.

Théorème (Règle des cycles)**.** Soit (X, Y) un approximant et soit C un cocycle tel que pour toute arête $a = \{x, y\} \in A,\ a \in X$ ou $a \in Z = A \backslash (X \cup Y)$.

Soit e une arête telle que $e \in Z$ de coût maximum. Alors $(X, Y \cup \{e\})$ est un approximant.

Algorithme 6 : Algorithme générique

début
- $(X, Y) \longleftarrow (\varnothing, \varnothing)$
- $Z \longleftarrow A \backslash (X \cup Y)$
- **répéter**
  - **si** *Règle des cocyles applicable à un cycle* C **alors**
    - e : arête de coût minimum de $C \cap Z$
    - $(X, Y) \longleftarrow (X \cup \{e\}, Y)$
  - **sinon si** *Règle des cycles applicable à un cycle* C **alors**
    - e : arête de coût maximum de $C \cap Z$
    - $(X, Y) \longleftarrow (X, Y \cup \{e\})$
- **jusqu'à** $|X| = n - 1$ *ou* $|Y| = m - (n - 1)$
- **retourner** X

4.2 Algorithme de Prim

Algorithme 7 : Algorithme de Prim

début
- $(X, Y) \longleftarrow (\varnothing, \varnothing)$
- $T \longleftarrow \{s\}$
- **pour** k *de 1 à* $n - 1$ **faire**
  - $e = \{u, v\}$: arête de coût minimum de $\omega(T)$
  - $(X, Y) \longleftarrow (X \cup \{e\}, Y)$
  - $T \longleftarrow T \cup \{v\}$
- **retourner** X

Complexité (Algorithme de Prim)**.** En utilisant un tas : complexité en $\mathcal{O}((n + m)\log(n))$.

4.3 Algorithme de Kruskal

Algorithme 8 : Algorithme de Kruskal

début
- $(X, Y) \longleftarrow (\varnothing, \varnothing)$
- `Trier`(A)
- **pour** k *de 1 à* m **faire**
  - $a_k : \{u_k, v_k\}$
  - **si** u_k *et* v_k *sont dans la même CFC* **alors**
    - $(X, Y) \longleftarrow (X, Y \cup \{a_k\})$
  - **sinon**
    - $(X, Y) \longleftarrow (X \cup \{a_k\}, Y)$
- **retourner** X

Complexité (Algorithme de Kruskal)**.** En utilisant `Union` et `Find`, fonctions en $\mathcal{O}(\log(n))$, alors la complexité de l'algorithme de Kruskal est $\mathcal{O}(m\log(m) + m\log(n)) = \mathcal{O}(m\log(m))$.