

Contents

1	Notions et notations de base de la théorie de graphes	2
1.1	Graphe (non-orienté)	2
1.2	Graphe orienté	3
1.3	Isomorphisme de graphes	4
1.4	Cheminement	4
1.5	Connexité	6
1.6	Distances	7
2	Représentations de graphes	7
2.1	Matrice d'adjacence	7
2.2	Matrice d'incidence	8
2.3	Listes d'adjacence	9
3	Complexité des algorithmes	9
3.1	Notation des classes de complexité	9
4	Algorithmes d'exploration de graphes	10
4.1	Parcours en largeur	10
4.2	Parcours en profondeur	12
4.2.1	Première application de parcours en profondeur : tri topologique	14
4.2.2	Deuxième application de parcours en profondeur : com- posantes fortement connexes	15
5	Arbres couvrants de poids minimum	17
5.1	Énoncé du problème	17
5.2	Algorithmes	18
6	Calcul de distances	18
6.1	Graphes non-orientés	20
6.2	Graphes orientés sans circuit	21
6.3	Graphes orientés avec circuits	21
6.4	Plus courts chemins entre toutes paires de sommets	23
7	Optimisation de flot	24

Algorithmique de graphes

František Kardoš

September 2017

1 Notions et notations de base de la théorie de graphes

(à mettre à jour régulièrement au fur et à mesure)

1.1 Graphe (non-orienté)

En théorie des graphes, un graphe G est une paire $G = (V, E)$ d'ensembles finis, où

- $V = V(G)$ est l'ensemble (non-vide) des *sommets* du graphe G , et
- $E = E(G)$ est l'ensemble des *arêtes* du graphe G , où

chaque arête est une paire de sommets.

Par exemple, si x et y sont des sommets, la paire $e = \{x, y\}$ peut être une arête du graphe G , auquel cas elle est notée simplement $e = xy$.

Si $e = xy$ est une arête d'un graphe G , on dit que l'arête e relie les sommets x et y ; les sommets x et y sont les *extrémités* de l'arête e . Si une arête reliant les sommets x et y existe, on dit que les sommets x et y sont *voisins* (l'un de l'autre), et qu'ils sont *adjacents*. Si une arête $e = xy$ relie le sommet x à un sommet y , on dit aussi que l'arête e est *incidente* à x (et à y).

Une arête reliant un sommet à lui-même est une *boucle*. Deux (ou plusieurs) arêtes sont dites *parallèles*, si elles relient la même paire de sommets. Une *arête multiple* est un ensemble d'arêtes parallèles.

Un graphe sans boucles ni arêtes multiples est dit *simple*.

Le *degré* $\deg(v)$ d'un sommet v d'un graphe G est le nombre d'arêtes de G incidentes à v (une boucle compte pour deux incidences).

Dans un graphe simple, le degré d'un sommet v est égale au nombre de voisins de v .

Un sommet de degré 0 est dit *isolé*.

Dans un graphe simple, un sommet sans voisin est isolé. Observer que dans un graphe non-simple, un sommet sans voisin peut avoir un degré supérieur à 0, s'il est incident à une (ou plusieurs) boucle.

Observation 1 *La somme des degrés de sommets d'un graphe est égale à deux fois le nombre d'arêtes, elle est donc paire.*

Preuve. Considérons le nombre total d'incidences (de couples sommet-arête incidente) dans un graphe donné. Pour chaque sommet du graphe, son degré est égale au nombre d'incidences auxquelles il participe. Le nombre total d'incidences est donc égal à la somme des degrés de tous les sommets du graphe donné. D'autre part, chaque arête contribue à exactement deux incidences – celles avec ses extrémités. Le nombre total d'incidences est donc égal à deux fois le nombre d'arêtes du graphe donné. \square

Corrolaire 1 *Le nombre de sommets de degré impair est toujours pair.*

Preuve. Exercice du cours. \square

1.2 Graphe orienté

En théorie des graphes, un graphe orienté G est une paire $G = (V, A)$ d'ensembles finis, où

- $V = V(G)$ est l'ensemble (non-vide) des *sommets* du graphe G , et
- $A = A(G)$ est l'ensemble des *arcs* du graphe G , où

chaque arc est un couple de sommets.

Par exemple, si x et y sont des sommets, le couple $a = (x, y)$ peut être un arc du graphe orienté G , auquel cas elle il est noté simplement $a = xy$. L'ordre des sommets est important, un arc xy est différent d'un arc yx .

Si $a = xy$ est un arc d'un graphe orienté G , on dit que l'arc a relie le sommet x au sommet y ; le sommet x est le pied (la queue, l'origine) de l'arc a , le sommet y est la tête (la destination) de l'arc a . Si un arc xy existe, on dit que le sommet x est un *voisin entrant* ou *prédécesseur* de y , on dit aussi que le sommet y est un *voisin sortant* ou *successeur* de x . Pareil, l'arc $a = xy$ est un *arc sortant* de x et un *arc entrant* à y .

Le *degré entrant* $\deg^-(v)$ d'un sommet v d'un graphe orienté G est le nombre d'arcs de G entrant à v (dirigés vers v ; nombre de têtes à v). Le *degré sortant* $\deg^+(v)$ d'un sommet v d'un graphe orienté G est le nombre d'arcs de G sortant de v (nombre de pieds à v).

Observation 2 *Dans un graphe orienté G , la somme des degrés entrants est égale à la somme des degrés sortants.*

Preuve. Il suffit d'observer que chaque arc d'un graphe donné contribue une fois à chacune des deux sommes. Elles sont alors toutes les deux égales au nombre d'arcs du graphe. \square

1.3 Isomorphisme de graphes

De façon informelle, si deux graphes ont le même nombre de sommets et leurs sommets sont reliés entre eux de la même façon, ils sont considérés comme deux représentations de la même situation. D'où la définition suivante :

Soient $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$ deux graphes simples. Un *isomorphisme* entre G_1 et G_2 (de G_1 vers G_2) est une bijection $\varphi : V_1 \rightarrow V_2$ telle que

$$uv \in E_1 \iff \varphi(u)\varphi(v) \in E_2 \quad \forall uv \in E_1.$$

Deux graphes sont dits *isomorphes* s'il existe un isomorphisme entre eux (dans n'importe quel sens).

Il est facile d'observer que s'il existe un isomorphisme entre G_1 et G_2 , alors il existe un isomorphisme entre G_2 et G_1 – la bijection réciproque en est un.

Observer que cette définition d'isomorphisme de graphes n'est pas correcte pour les graphes avec des arêtes multiples. Par exemple, les graphes $(\{x, y\}, \{xy\})$ et $(\{u, v\}, \{uv, uv, uv\})$ seraient isomorphes, alors qu'ils n'ont pas le même nombre d'arêtes.

Exercices de cours: Proposer une définition d'isomorphisme pour les graphes (non-orienté), tenant compte des boucles et des arêtes multiples.

1.4 Cheminement

Dans un graphe (non-orienté), une chaîne est une suite finie d'arêtes consécutives. Plus formellement, une *chaîne* de u à v est une suite de sommets et d'arêtes

$$u_0, e_1, u_1, \dots, e_k, u_k$$

avec les propriétés suivantes:

- $\forall i = 1, \dots, k$ l'arête e_i relie les sommets u_{i-1} et u_i ;
- $u_0 = u$ et $u_k = v$.

La *longueur* d'une telle chaîne est égale à k , le nombre d'arêtes dans la chaîne (y compris les répétitions).

Une chaîne est dite *simple* si elle ne passe pas deux fois par la même arête.

Une chaîne est dite *élémentaire* si elle ne passe pas deux fois par le même sommet.

Observation 3 *Toute chaîne élémentaire est simple. Toute chaîne (pas forcément élémentaire ou simple) peut être raccourci en une chaîne élémentaire.*

Preuve. Soit $P = u_0, e_1, u_1, \dots, e_k, u_k$ une chaîne dans un graphe G . Si elle n'est pas élémentaire, il y a au moins un sommet qui est visité au moins deux fois par la chaîne P . Tant qu'il en existe au moins un tel sommet, on répète la procédure suivante :

Soit u un sommet qui est visité (au moins) deux fois par P . Soient i et j ($i < j$) deux indices représentant la première et la dernière apparition de

u le long de P : $u_i = u_j = u$. On peut couper et supprimer la sous-chaîne $(u_i), e_{i+1}, u_{i+1}, \dots, e_j, (u_j)$ et concaténer les deux sous-chaînes restant. Ainsi on obtient une chaîne P' contenu dans P avec moins de sommets répétés, pour laquelle on peut redémarrer la procédure si nécessaire.

On fini forcément par éliminer toutes les répétition, donc la chaîne obtenu est enfin élémentaire. \square

Lorsqu'on parle d'une chaîne dans un graphe simple, parfois on ne spécifie pas les arêtes de la chaîne, puisque (par simplicité) une suite de sommets d'une chaîne détermine les arêtes les reliant l'un au suivant.

Une chaîne de longueur 0 ne contient aucune arête, elle est considérée triviale.

Une chaîne non-triviale reliant un sommet à lui-même, passant une fois au maximum par chaque sommet (sauf le sommet de départ et le sommet d'arrivée), est appelée un *cycle*.

Lorsqu'on parle d'un cycle, on ne spécifie presque jamais le sommet initial (et terminal) ce celui-ci. Par contre, le même ensemble de sommets peut être traversé par plusieurs cycles différents.

Observer que toute boucle est un cycle de longueur 1 et que toute paire d'arêtes parallèles est un cycle de longueur 2. Observer que dans un graphe simple, la longueur de tout cycle est au moins 3.

Dans un graphe orienté, un chemin est une suite finie d'arcs consécutifs. Plus formellement, un *chemin* de u à v est une suite de sommets et d'arcs

$$u_0, e_1, u_1, \dots, e_k, u_k$$

avec les propriétés suivantes:

- $\forall i = 1, \dots, k$ l'arc e_i est un arc de u_{i-1} à u_i ;
- $u_0 = u$ et $u_k = v$.

La *longueur* d'un tel chemin est égale à k , le nombre d'arcs dans le chemin (y compris les répétitions).

Un chemin est dit *simple* s'il ne passe pas deux fois par le même arc.

Un chemin est dit *élémentaire* s'il ne passe pas deux fois par le même sommet.

Observation 4 *Tout chemin élémentaire est simple. Tout chemin (pas forcément élémentaire ou simple) peut être raccourci en un chemin élémentaire.*

Preuve. De la même façon que dans le cas de chaînes dans les graphes non-orienté, on élimine les parts inutiles. \square

Un chemin de longueur 0 ne contient aucun arc, il est considéré trivial.

Un chemin non-trivial reliant un sommet à lui-même, passant une fois au maximum par chaque sommet (sauf le sommet de départ et le sommet d'arrivée), est appelé un *circuit*.

Lorsqu'on parle d'un circuit, on ne spécifie presque jamais le sommet initial (et terminal) ce celui-ci. Par contre, le même ensemble de sommets peut être traversé par plusieurs cycles différents.

Observer que toute boucle est un circuit de longueur 1 et que toute paire d'arcs opposés est un circuit de longueur 2. Observer que dans un graphe orienté simple, la longueur de tout circuit est au moins 3.

1.5 Connexité

Un graphe (non-orienté) est dit *connexe*, si pour toute paire de sommets u et v il existe une chaîne les reliant.

Proposition 1 *Soit G un graphe. Les énoncés suivants sont équivalents:*

1. G est connexe;
2. Pour toute paire de sommets u et v il existe une chaîne élémentaire les reliant;
3. Il existe un sommet u tel que pour tout autre sommet v il existe une chaîne entre u et v .

Preuve. Il suffit de vérifier que $2 \implies 3 \implies 1 \implies 2$. Chacune des trois implications est assez directe. Exercice de cours. \square

Une *composante connexe* d'un graphe G est un sous-graphe de G connexe, et maximal par rapport à la connexité. Ici H est un sous-graphe de G si $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$; un sous-graphe connexe de G est maximal par rapport à la connexité si on ne peut pas y ajouter d'éléments de G (ni sommet, ni arête) sans rompre la connexité.

Tout graphe connexe possède une et une seule composante connexe – c'est le graphe entier lui-même. Si un graphe n'est pas connexe, il se décompose en un nombre fini de composantes connexes (au moins deux).

Un graphe est un *arbre* s'il est connexe et sans cycle. Un sous-graphe H d'un graphe G est un *arbre couvrant* de G si H est un arbre et $V(H) = V(G)$.

Proposition 2 *Soit G un graphe à n sommets et m arêtes. Les énoncés suivants sont équivalents:*

1. G est un arbre, c-à-d il est connexe et sans cycle;
2. G est connexe et $m = n - 1$;
3. G est sans cycle et $m = n - 1$;
4. G est connexe, mais en supprimant n'importe quelle arête il perd la connexité;
5. G est sans cycle, mais en ajoutant une nouvelle arête entre n'importe quelle paire de sommets, il perd la propriété d'être sans cycle.

1.6 Distances

Soient u et v deux sommets d'un graphe G . La *distance* entre u et v (la distance de u à v) est la longueur de la plus courte chaîne les reliant, s'il en existe au moins une; s'il n'y a pas de chaîne reliant u et v , la distance entre u et v est infinie.

Il est facile à vérifier que la distance entre deux sommets d'un graphe est finie si et seulement si ils appartiennent à la même composante connexe du graphe donné.

La distance dans les graphes telle qu'elle est définie a bien les propriétés classiques d'une distance en générale:

Proposition 3 Soit G un graphe.

1. $\text{dist}(u, v) \geq 0 \quad \forall u, v \in V(G); \text{dist}(u, v) = 0 \iff u = v.$
2. $\text{dist}(u, v) = \text{dist}(v, u) \quad \forall u, v \in V(G).$
3. $\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w) \quad \forall u, v, w \in V(G).$

Preuve. Exercice de cours.

2 Représentations de graphes

2.1 Matrice d'adjacence

Soit G un graphe (non-orienté). Soit $V(G) = \{v_1, v_2, \dots, v_n\}$. La *matrice d'adjacence* de G est une matrice $A = (a_{ij})$ de taille $n \times n$ définie par

$$a_{ij} = \text{le nombre d'arêtes reliant } v_i \text{ et } v_j,$$

où (dans le cas où $i = j$) chaque boucle est compté deux fois.

Si le graphe est simple, alors la matrice est binaire, et on a $a_{ij} = 1$ si les sommets v_i et v_j sont adjacents, sinon $a_{ij} = 0$.

Comme chaque incidence de chaque arête incidente à un sommet donné v_i est compté dans un et un seul élément de la ligne i de la matrice, on a bien

$$\text{deg}(v_i) = \sum_{j=1}^n a_{ij} \quad \forall i = 1, \dots, n.$$

La matrice d'adjacence d'un graphe non-orienté est toujours symétrique.

Soit G un graphe orienté. Soit $V(G) = \{v_1, v_2, \dots, v_n\}$. La matrice d'adjacence $A = (a_{ij})$ est une matrice de taille $n \times n$ définie par

$$a_{ij} = \text{le nombre d'arcs de } v_i \text{ à } v_j.$$

Si le graphe n'a pas d'arcs parallèles, alors la matrice est binaire, et on a $a_{ij} = 1$ si l'arc de v_i à v_j existe, sinon $a_{ij} = 0$.

Comme chaque arc sortant d'un sommet donné v_i est compté dans un et un seul élément de la ligne i de la matrice, on a bien

$$\deg^+(v_i) = \sum_{j=1}^n a_{ij} \quad \forall i = 1, \dots, n,$$

c-à-d la somme de la ligne i de la matrice d'adjacence est égale au degré sortant de sommet v_i .

Pareil, comme chaque arc entrant d'un sommet donné v_j est compté dans un et un seul élément de la colonne j de la matrice, on a bien

$$\deg^-(v_j) = \sum_{i=1}^n a_{ij} \quad \forall j = 1, \dots, n,$$

c-à-d la somme de la colonne j de la matrice d'adjacence est égale au degré entrant de sommet v_j .

2.2 Matrice d'incidence

Soit G un graphe (non-orienté) sans boucle. Soit $V(G) = \{v_1, v_2, \dots, v_n\}$ et $E(G) = \{e_1, e_2, \dots, e_m\}$. La *matrice d'incidence* de G est une matrice $B = (b_{ij})$ de taille $n \times m$ définie par

$$b_{ij} = \begin{cases} 1 & \text{si } e_j \text{ est incidente à } v_i, \\ 0 & \text{sinon;} \end{cases}$$

Si jamais G a une boucle e_j incidente à un sommet v_i , on pose $b_{ij} = 2$ (la boucle est double incidente au sommet).

Comme chaque arête incidente à un sommet donné v_i est compté dans un et un seul élément de la ligne i de la matrice, on a bien

$$\deg(v_i) = \sum_{j=1}^m b_{ij} \quad \forall i = 1, \dots, n.$$

Soit G un graphe orienté sans boucle. Soit $V(G) = \{v_1, v_2, \dots, v_n\}$ et $E(G) = \{a_1, a_2, \dots, a_m\}$. La *matrice d'incidence* de G est une matrice $B = (b_{ij})$ de taille $n \times m$ définie par

$$b_{ij} = \begin{cases} 1 & \text{si } a_j \text{ est un arc entrant de } v_i, \\ -1 & \text{si } a_j \text{ est un arc sortant de } v_i, \\ 0 & \text{sinon;} \end{cases}$$

Dans la littérature, parfois les rôles de 1 et -1 sont inversés.

La notion de matrice d'incidence n'est pas bien définie pour les graphes orientés avec des boucles.

2.3 Listes d'adjacence

En toute simplicité, une *liste d'adjacence* $Adj(v)$ d'un sommet v d'un graphe (orienté) G est la liste des voisins (sortants) de v dans G . Si un graphe est représenté par les listes d'adjacence, une liste d'adjacence est donnée pour tout sommet du graphe en question.

3 Complexité des algorithmes

La complexité d'un algorithme est (une borne inférieure sur) le nombre d'opérations élémentaires à effectuer (au pire cas) pendant l'exécution de l'algorithme, exprimé en fonction de la taille de l'entrée de l'algorithme.

Comme paramètres pour la complexité des algorithmes sur les graphes, sont utilisés le plus souvent n (le nombre des sommets) et/ou m (le nombre des arêtes/arcs).

3.1 Notation des classes de complexité

Une fonction $T(n)$ est en $O(f(n))$, si l'ordre de croissance de $f(n)$ est une borne supérieure sur l'ordre de croissance de $T(n)$. Plus précisément, la limite

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)}$$

existe et elle est finie. Autrement dit, il existe des constants c et n_0 tels que

$$\forall n \geq n_0, \quad T(n) \leq c \cdot f(n).$$

Par abus de notation, on écrit dans ce cas-là $T(n) = O(f(n))$, même s'il serait plus correct d'écrire $T(n) \in O(f(n))$.

Une fonction $T(n)$ est en $o(f(n))$, si l'ordre de croissance de $T(n)$ est strictement plus petit que celui de $f(n)$. Plus précisément,

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = 0.$$

Une fonction $T(n)$ est en $\Omega(f(n))$, si l'ordre de croissance de $f(n)$ est une borne inférieure sur l'ordre de croissance de $T(n)$. Plus précisément,

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} > 0.$$

Autrement dit, il existe des constants c et n_0 tels que

$$\forall n \geq n_0, \quad T(n) \geq c \cdot f(n).$$

Finalement, une fonction $T(n)$ est en $\Theta(f(n))$, si l'ordre de croissance de $f(n)$ est asymptotiquement le même que celui de $T(n)$. Plus précisément,

$$0 < \lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} < \infty.$$

Autrement dit, il existe des constantes c_1 , c_2 et n_0 tels que

$$\forall n \geq n_0, \quad c_1 \cdot f(n) \geq T(n) \geq c_2 \cdot f(n).$$

4 Algorithmes d'exploration de graphes

Il y a deux algorithmes principaux d'exploration des graphes: le parcours en largeur et le parcours en profondeur.

4.1 Parcours en largeur

Il s'agit d'une procédure qui, étant donné un graphe G et un sommet de départ s , visite chaque sommet accessible depuis s dans l'ordre croissant de distance depuis s , et explore les arêtes incidentes pour découvrir d'autres sommets un par un.

Cet algorithme peut modéliser par exemple la diffusion d'un message dans un réseau, où les noeuds du réseau respectent le protocole suivant: dès que je reçois le message pour la première fois, je le retransmets tout de suite à tous mes voisins; si jamais je reçois le (même) message de nouveau, je ne le transmets plus.

Algorithm 1 Parcours en largeur PL(G, s)

```

1: couleur( $s$ )  $\leftarrow$  GRIS
2:  $d(s) \leftarrow 0$ 
3: pere( $s$ )  $\leftarrow$  NIL
4:  $F \leftarrow \{s\}$ 
5: for all  $v \in V(G) \setminus s$  do
6:   couleur( $v$ )  $\leftarrow$  BLANC
7:    $d(v) \leftarrow \infty$ 
8:   pere( $v$ )  $\leftarrow$  NIL
9: end for
10: while  $F$  non vide do
11:    $v \leftarrow \text{tete}(F)$ 
12:   for all  $w \in \text{Adj}(v)$  do
13:     if couleur( $w$ ) = BLANC then
14:       couleur( $w$ )  $\leftarrow$  GRIS
15:        $d(w) \leftarrow d(v) + 1$ 
16:       pere( $w$ )  $\leftarrow$   $v$ 
17:       Enfiler( $F, w$ )
18:     end if
19:   end for
20:   Defiler( $F$ )
21:   couleur( $v$ )  $\leftarrow$  NOIR
22: end while
```

L'algorithme $PL(G, s)$ utilise une structure de données auxiliaire F qui est une file (FIFO). Les tableaux $pere()$, $d()$, et $couleur()$ sont remplis progressivement au cours de l'exécution de l'algorithme.

Les couleurs sont utilisées pour mettre en évidence les trois états dans lesquels un sommet peut se trouver:

BLANC – pour tout sommet qui n'a pas encore été enfilé

GRIS – pour tout sommet qui est actuellement dans la file

NOIR – pour tout sommet (au moins une fois) défilé

Un sommet est enfilé soit la ligne 4 (s'il s'agit du sommet de départ s) soit la ligne 17. Observer que le sommet enfilé est toujours blanc, et que sa couleur passe du blanc au gris au moment où il est enfilé. Pareil, les lignes 20 et 21 indiquent que la couleur d'un sommet passe du gris au noir si et seulement si le sommet est défilé.

Ceci dit, il est clair que tout sommet du graphe donné est enfilé une fois au maximum. Un sommet est dit *découvert* à partir du moment où il est colorié gris. Un sommet est dit *visité* à partir du moment où il est colorié noir. La valeur de $d(v)$ (et aussi de $pere(v)$) est calculée uniquement au moment de découverte de v . Soyons encore plus précis:

Proposition 4 *Soit G un graphe et s un sommet de G . Pendant l'exécution de $PP(G, s)$, un sommet v est découvert (et puis visité) si et seulement si v est accessible depuis le sommet de départ s .*

Les sommets sont découverts dans l'ordre croissant par rapport à la distance depuis s . D'ailleurs, la valeur $d(v)$ calculée pendant l'exécution de $PP(G, s)$ est la distance entre s et v .

Preuve. Par récurrence par rapport à la (vraie) distance $dist(s, v)$.

Si $dist(s, v) = 0$, alors $v = s$, et v est le premier sommet enfilé. La valeur $d(s)$ est mise à $0 = dist(v, s)$ pendant l'initialisation (ligne 2). Tous les voisins de s (tous les sommets avec $dist(s, v) = 1$) sont découverts (et donc enfilé) pendant la visite de s . D'autres sommets sont donc enfilé encore plus tard.

Supposons les énoncés vrais pour tous les sommets à distance au plus $k \geq 0$. Soit w un sommet de G tel que $dist(s, w) = k + 1$. Alors, il n'existe pas de chaîne de longueur (au plus) k reliant s et w , mais il existe (au moins) une chaîne de longueur $k + 1$ les reliant. Pour chaque voisin v de w , on a soit $dist(s, v) = k$, soit $dist(s, v) = k + 1$, soit $dist(s, v) = k + 2$.

Pour chaque chaîne de longueur $k + 1$ entre s et w , l'avant dernier sommet est un sommet v voisin de w tel que $dist(s, v) = k$. Par l'hypothèse de récurrence, le sommet v est découvert par $PL(G, s)$ avant w , et $d(v) = dist(s, v)$.

Parmi toutes les chaînes de longueur $k + 1$ entre s et w considérons celle où le sommet v (l'avant dernier) est découvert le premier. Pendant la visite de v (lignes 11-18), le sommet w est découvert (lignes 13-17), et on a (ligne 15)

$$d(w) = d(v) + 1 = k + 1 = dist(s, w).$$

Pour conclure, chaque sommet w avec $\text{dist}(s, w) = k + 1$ est enfilé pendant la visite d'un sommet v avec $\text{dist}(s, v) = k$, par conséquent, tous les sommets à distance $k + 1$ sont enfilés un par un pendant que tous les sommets à distance k sont défilés un par un.

Les sommets inaccessibles depuis s sont exactement ceux avec $\text{dist}(s, v) = \infty$; un tel sommet n'est jamais découvert par $PL(G, s)$. \square

Proposition 5 *Soit G un graphe et s un sommet de G . Soit C la composante connexe de G contenant s . Après une exécution de $PP(G, s)$, considérons l'ensemble d'arêtes*

$$E = \{(u, \text{pere}(u)) \mid u \in V(C), u \neq s\}.$$

Le graphe $H = (V(C), E)$ est un arbre couvrant de C .

Preuve. Puisque la composante connexe C contient exactement tous les sommets visités par un parcours en largeur à partir de s , pour tous les sommets de C (sauf s) le père existe, donc l'ensemble E est bien défini.

Pour chaque sommet $v \neq s$, la suite d'arêtes

$$(v, \text{pere}(v)), (\text{pere}(v), \text{pere}(\text{pere}(v))), \dots, (\text{pere}(\dots(\text{pere}(v))\dots), s)$$

est une chaîne de longueur $\text{dist}(v, s)$ reliant v à s . D'après Proposition 1, le graphe H est connexe.

En plus, $|E| = |V(C)| - 1$ puisque, pour chaque sommet de C sauf s , il y a une et une seule arête dans E . D'après Proposition 2, H est un arbre.

Le fait que H est un arbre couvrant de C suit directement par définition. \square

4.2 Parcours en profondeur

Il s'agit d'un algorithme d'exploration d'un graphe, qui simule la recherche d'un noeud précis dans un réseau, avec un seul agent de recherche disponible. La recherche respecte la règle suivante: Après avoir exploré un noeud, je ne continue jamais par des arêtes/des arcs qui m'amènent à un noeud déjà exploré. Je ne fais marche arrière par l'arête/l'arc qui m'a permis de découvrir le noeud actuel que s'il n'y a plus rien à explorer (s'il n'y a plus d'arêtes/d'arcs que je n'ai pas essayés).

Contrairement au parcours en largeur, le sommet de départ n'est pas un paramètre de l'algorithme de parcours en profondeur: Tant qu'il reste des sommets blancs (non explorés), il redémarre de nouveau avec un nouveau sommet de départ.

L'algorithme utilise une structure de données de type LIFO – une pile. Dans la première version de l'algorithme $PP(G)$ la pile n'est pas explicitée, elle est implémentée sous forme d'une pile d'appels à une fonction récursive $\text{VisiterPP}(v)$.

Il est cependant possible de transformer l'algorithme de façon à ce que la pile soit explicite et il n'y ait pas de récursivité: À chaque étape, soit le dernier

Algorithm 2 Parcours en profondeur $PP(G)$

```
1: for all  $v \in V(G)$  do
2:    $couleur(v) \leftarrow BLANC$ 
3:    $pere(v) \leftarrow NIL$ 
4: end for
5:  $temps \leftarrow 0$ 
6: for all  $v \in V(G)$  do
7:   if  $couleur(v) = BLANC$  then
8:      $VisiterPP(v)$ 
9:   end if
10: end for
```

Algorithm 3 $VisiterPP(v)$

```
1:  $d(v) \leftarrow temps \leftarrow temps + 1$ 
2:  $couleur(v) \leftarrow GRIS$ 
3: for all  $w \in Adj(v)$  do
4:   if  $couleur(w) = BLANC$  then
5:      $pere(w) \leftarrow v$ 
6:      $VisiterPP(w)$ 
7:   end if
8: end for
9:  $couleur(v) \leftarrow NOIR$ 
10:  $f(v) \leftarrow temps \leftarrow temps + 1$ 
```

sommet de la pile a encore un voisin (sortant) non-découvert (blanc), celui-ci est empilé dans ce cas-là; soit le dernier sommet de la pile n'a plus de voisin (sortant) blanc, on le dépile alors.

Exercice de cours: Donner la version non-réursive de la fonction $VisiterPP(v)$.

L'algorithme $PP(G)$ utilise un compteur de temps pour mémoriser les dates de début et de fin de visite de chaque sommet. Puisqu'un sommet est colorié gris lorsqu'il est empilé et noir lorsqu'il est dépilé, et que la condition nécessaire pour être empilé est d'être colorié blanc, tout sommet d'un graphe donné est empilé, visité et dépilé une et une seule fois.

Dans un graphe orienté, tout arc uv peut se classer dans une des quatre classes suivantes :

- Un arc de liaison est un arc reliant un sommet u à son fils v . (Un sommet v est un fils de u si et seulement si $u = pere(v)$.) Au moment où l'arc uv est exploré, u est en tête de la pile (donc gris) et v est blanc. C'est v le prochain sommet à être visité.
- Un arc de retour est un arc reliant un sommet u à un de ses ancêtres. (Un sommet v est un ancêtre de u si c'est le père de u ou le père d'un autre ancêtre de u .) Au moment où l'arc uv est exploré, u est en tête de la pile (donc gris), et v est aussi empilé (donc gris), mais plus tôt que u .

- Un arc avant est un arc reliant un sommet u à un de ses descendants, mais pas un fils. (Un sommet v est un descendant de u si et seulement si u est un ancêtre de v .) Au moment où l'arc uv est exploré, u est en tête de la pile (donc gris), et v est déjà noir, d'ailleurs, la visite de v s'est déroulée pendant la visite de u .
- Tout autre arc est un arc transverse. Un arc transverse est un arc reliant un sommet u à un autre sommet v , tel qu'au moment où l'arc uv est exploré, u est en tête de la pile (donc gris), et v est déjà noir, comme la visite de v s'est terminé avant le début de la visite de u .

Dans un graphe non-orienté, toute arête est soit une arête de liaison (une arête de découverte), soit une arête de retour. Il n'existe pas d'arêtes avant ni arêtes transverses dans un graphe non-orienté.

On peut définir l'intervalle de la durée de la visite d'un sommet u comme $I(u) = [d(u), f(u)]$ pour tout $u \in V(G)$.

Puisque la pile est une structure de type LIFO, les intervalles des sommets ne peuvent pas se chevaucher:

Proposition 6 *Soient $u, v \in V(G)$. Alors soit $I(u)$ et $I(v)$ sont disjoint, soit l'un est inclus dans l'autre. Plus précisément, si u est un descendant (y compris un fils) de v , alors on a $I(u) \subset I(v)$; si uv est un arc transverse, alors $I(u) \cap I(v) = \emptyset$.*

4.2.1 Première application de parcours en profondeur : tri topologique

Soit G un graphe orienté. Un *tri topologique* de G est une numérotation (une permutation) des sommets $V(G) = \{v_1, v_2, \dots, v_n\}$ telle que si $v_i v_j \in E(G)$, alors $i < j$. Autrement dit, si les sommets sont représentés sur une ligne, tout arc va de gauche à droite.

Théorème 1 *Soit G un graphe orienté. Il existe un tri topologique de G si et seulement si G est sans circuit.*

Preuve. Montrons d'abord que l'absence de circuit est une condition nécessaire pour l'existence d'un tri topologique: Soit G un graphe orienté est $V(G) = \{v_1, v_2, \dots, v_n\}$ un tri topologique de G . Supposons qu'il existe un circuit C dans G . Pour chaque sommet de C , il y a un arc entrant et un arc sortant le reliant à un autre sommet de C . Donc, pour le sommet le plus à droite de C il y a forcément un arc sortant vers un sommet plus à gauche, ce qui contredit la définition d'un tri topologique, donc absurde.

Montrons maintenant que l'absence de circuit est une condition suffisante pour l'existence d'un tri topologique: Dans un graphe G sans circuit, nous allons construire/définir un tri topologique de la façon suivante: Ordonner les sommets de G par rapport à l'ordre décroissant de la fin de visite lors d'un parcours en profondeur de G .

Il reste à montrer qu'effectivement, cet ordre est un tri topologique: Il suffit de montrer que pour tout arc uv de G , on a $f(u) > f(v)$. Considérons tous les quatre types d'arcs dans G :

- Si uv est un arc de liaison ou un arc avant, alors $I(v) \subseteq I(u)$. Plus précisément, $d(u) < d(v) < f(v) < f(u)$, d'où l'inégalité demandée.
- Si uv est un arc transeverse, alors les intervalles de u et de v sont disjoints. Plus précisément, la visite de v d'est déroulée avant la visite de u , c-à-d $d(v) < f(v) < d(u) < f(u)$, d'où l'inégalité demandée.
- Il ne peut pas y être d'arc de retour dans un graphe sans circuit: Si uv était un arc de retour, alors le chemin de v à u composé d'arcs de liaison justifiant que v est un ancêtre de u , ensemble avec l'arc uv formerait un circuit dans G , ce qui est absurde.

Pour conclure, tout arc de G est en accord avec l'ordre défini par le parcours en profondeur, cet ordre est alors un tri topologique de G . \square

4.2.2 Deuxième application de parcours en profondeur : composantes fortement connexes

Avant d'énoncer l'algorithme, voici quelques propriétés de composantes fortement connexes par rapport à un parcours en profondeur:

Soit G un graphe orienté. Le graphe des composantes fortement connexes de G est un graphe orienté $CFC(G)$ dont les sommets sont les composantes fortement connexes de G ; il y a un arc de C_1 à C_2 dans $CFC(G)$ si et seulement s'il existe un arc v_1v_2 dans G avec $v_1 \in C_1$ et $v_2 \in C_2$.

Observation 5 *Soit G un graphe orienté. Le graphe $CFC(G)$ n'admet pas de circuit.*

Preuve. Il suffit d'observer (par définition), que s'il y avait un circuit dans $CFC(G)$, alors l'union des composantes fortement connexes composant le circuit serait un sous-graphe de G fortement connexe les contenant toutes, ce qui contredirait la maximalité de chacune d'elles. \square

Théorème 2 (du chemin blanc) *Soit v un sommet d'un graphe orienté G . Lors d'un $PP(G)$, un sommet u est découvert et visité pendant la visite de v (on a $I(u) \subset I(v)$; u est un descendant de v) si et seulement si au moment début de visite de v , le sommet u est accessible depuis v par un chemin composé uniquement par des sommets blancs à ce moment-là.*

Preuve. On peut démontrer le théorème du chemin blanc par récurrence inverse par rapport à la profondeur de la pile: Supposons l'énoncé vrai pour tous les sommets de la profondeur supérieur à p . Soit v un sommet de profondeur égale à p . Il est facile à observer que tous les descendants de v sont empilés pendant que v reste empilé, ils sont donc tous de profondeur supérieur à p .

Il est clair que v a (au moins) un descendant ssi il a au moins un fils, c-à-d il y a (au moins) un sommet blanc accessible depuis v au moment de début de sa visite.

Soit u un descendant de v . Si u est un fils de v , alors au moment de début de la visite de v , il est forcément blanc. Si u est un descendant d'un fils de v , disons v' , alors, par l'hypothèse de récurrence, u est accessible par un chemin blanc depuis v' au moment de début de la visite de v' . Comme la visite de v' commence plus tard que la visite de v , au moment de début de la visite de v ce chemin-ci de v' à u est blanc aussi (y compris le sommet v').

Soit u un sommet accessible depuis v par un chemin blanc au moment de début de la visite de v . Parmi tous les chemins blancs de v à u , considérons celui où le successeur direct de v , disons v' , est visité le plus tôt. Ceci veut dire que au moment de début de la visite de v' , il existe toujours un chemin blanc de v' à u , et donc, par l'hypothèse de récurrence, u est un descendant de v . Mais v' n'est rien d'autre qu'un fils de v , donc u est un descendant d'un fils de v , d'où u est un descendant de v . \square

Comme une conséquence directe du théorème du chemin blanc on obtient l'énoncé suivant :

Observation 6 *Soit C une composante fortement connexe de G et soit v le sommet de C empilé en premier. Alors tous les autres sommets de C sont des descendants de C . Du coup, v est également le sommet de C dépilé en dernier.*

Observation 7 *Soient C_1 et C_2 deux composantes fortement connexes telles qu'il existe un chemin de C_1 à C_2 dans $CFC(G)$. Alors C_2 est visité entièrement pendant la visite du premier sommet de C_1 .*

Observation 8 *Soient C_1 et C_2 deux composantes fortement connexes telles qu'aucune d'elles n'est accessible depuis l'autre dans $CFC(G)$. Alors les intervalles de visite de C_1 et C_2 sont disjoints.*

Voici l'algorithme qui calcule les composantes fortement connexes d'un graphe orienté donné G .

Algorithm 4 $CFC(G)$

- 1: Exécuter $PP(G)$ et trier les sommets selon un ordre décroissant de f
 - 2: Calculer G^{-1}
 - 3: Exécuter $PP(G^{-1})$
 - 4: Retourner les arborescences obtenues
-

Il est utile d'observer que le $PP(G)$ et le fait de trier les sommets selon un ordre décroissant de la fin de visite induit un tri topologique de $CFC(G)$: Si chaque composante fortement connexe C est représentée par son sommet v_C visité en premier, on a bien

$$C_1 C_2 \in E(CFC(G)) \implies f(v_{C_1}) > f(v_{C_2}).$$

Soit C une composante connexe de G ; soit v le sommet de C empilé en premier. Pour tout autre sommet u de C , il existe dans G un chemin de u à v , donc, il existe un chemin de v à u dans G^{-1} . Comme v est le sommet dépilé en dernier lors de $PP(G)$ et que les sommets sont triés selon un ordre décroissant de f , c'est v qui est empilé en premier lors de $PP(G^{-1})$. En plus, comme les autres sommets de C sont accessibles depuis v dans G^{-1} , ils deviennent des descendants de v , s'il sont tous blancs au moment de début de la visite de v par $PP(G^{-1})$.

En effet, comme l'ordre des composantes fortement connexe est un tri topologique de $CFC(G)$, tous les arcs de G^{-1} sortant d'une composante donnée C' vont vers des composantes visitées avant C' . Ceci dit, aucun sommet de C ne peut être découvert depuis un sommet d'une autre composante C' .

5 Arbres couvrants de poids minimum

5.1 Énoncé du problème

Étant donné un graphe connexe G avec une pondération $w : E(G) \rightarrow \mathbb{R}^+$, on cherche un arbre couvrant de G (un arbre sur la totalité des sommets de G) de poids minimum. (Le poids d'un graphe est la somme de poids de ses arêtes.)

Pour tout arbre couvrant d'un graphe donné G le nombre d'arêtes est toujours de même – il est égale à $|V(G)| - 1$. La vraie question ici est plutôt comment sélectionner les arêtes afin de construire un arbre couvrant le moins cher possible.

Les algorithmes de construction d'arbres couvrants sont basés sur les caractérisations suivantes des solutions optimales:

Proposition 7 *Soit C un cycle dans G . Si e est une arête de C de poids strictement plus grand que les autres arêtes du cycle, alors cette arête n'est pas dans l'arbre couvrant de poids minimum.*

Autrement dit, si T^ est une solution optimale (un arbre couvrant de poids minimum) et $e_0 = uv$ est une arête de $E(G) \setminus E(T^*)$, alors pour toute arête e de l'unique chaîne reliant u et v dans T^* , on a $w(e) \leq w(e_0)$.*

Preuve. Par l'absurde. Supposons que la chaîne reliant les extrémités de e_0 dans une solution optimale T^* contient une arête e telle que $w(e) > w(e_0)$. En échangeant l'arête e pour l'arête e_0 on obtient un autre arbre couvrant $T = (V(G), (E(T) \setminus e) \cup e_0)$ de poids total inférieur à celui de T^* , ce qui contredit l'optimalité de T^* . \square

Proposition 8 *Soit T^* un arbre couvrant de poids minimum d'un graphe G . Soit (X, Y) une partition des sommets de G . Notons $E(X, Y)$ l'ensemble des arêtes de G ayant une extrémité dans X et l'autre dans Y . Alors T^* contient au moins une arête de $E(X, Y)$. Plus précisément, parmi les arêtes de $E(X, Y)$, T^* contient toujours la moins chère.*

Preuve. N'importe quel arbre couvrant de G doit contenir au moins une arête de $E(X, Y)$, sinon, T ne serait pas connexe. Supposons qu'un arbre couvrant de poids minimum T^* ne contient pas l'arête la moins chère de $E(X, Y)$, disons $e = uv$. Puisque u et v sont chacun dans une part différente, la chaîne reliant u et v dans T^* passe par au moins une autre arête de $E(X, Y)$, disons e_0 . Comme e est l'arête la moins chère de $E(X, Y)$, on a $w(e) < w(e_0)$. Or, en échangeant e_0 pour e , on peut construire un autre arbre couvrant de G encore moins cher que T^* , ce qui contredit l'optimalité de T^* . \square

5.2 Algorithmes

Le premier algorithme (celui de Kruskal) construit un arbre couvrant à partir d'un graphe sans arête, et à chaque étape il ajoute une arête en diminuant le nombre de composantes connexes. Les arêtes sont considérées dans un ordre croissant de leurs poids, afin de privilégier les arêtes les moins chères.

Algorithm 5 Kruskal(G, w)

```

1: trier les arêtes de  $G$  par poids croissant dans  $L$ 
2: for all  $u \in V(G)$  do
3:    $cc(u) \leftarrow u$ 
4: end for
5: while  $L$  non vide do
6:   soit  $(u, v)$  la tête de  $L$ 
7:   if  $cc(u) \neq cc(v)$  then
8:     ajouter l'arête  $(u, v)$  à l'arbre
9:     for all  $x \in V(G)$  do
10:      if  $cc(x) = cc(v)$  then
11:         $cc(x) \leftarrow cc(u)$ 
12:      end if
13:    end for
14:   end if
15:   Défiler  $(u, v)$ 
16: end while

```

Le deuxième algorithme fait croître un arbre depuis un sommet de départ r (la racine), et puis à chaque étape, il ajoute une arête de poids minimum reliant l'arbre en construction à un nouveau sommet. Deux versions différentes (mais équivalentes) sont proposées.

6 Calcul de distances

Dans cette section nous allons présenter plusieurs algorithmes de calcul de distances dans les graphes, non-orientés ou orientés. Il y a des algorithmes qui calculent les distances depuis un sommet de départ fixe, et aussi des algorithmes

Algorithm 6 Prim(G, w, r)

```
1: for all sommet  $v$  do
2:    $dist_T(v) \leftarrow \infty$ 
3:   if  $v$  voisin de  $s$  then
4:      $dist_T(v) \leftarrow w(r, v)$ 
5:      $cible(v) \leftarrow r$ 
6:   end if
7: end for
8:  $dist_T(r) \leftarrow 0$ 
9: for all  $i$  de 1 à  $n - 1$  do
10:  sélectionner le sommet  $v$  tel que  $dist_T(v) > 0$  minimum
11:  ajouter l'arête  $(v, cible(v))$  à l'arbre
12:   $dist_T(v) \leftarrow 0$ 
13:  for all voisin  $x$  de  $v$  do
14:    if  $dist_T(x) > w(v, x)$  then
15:       $dist_T(x) \leftarrow w(v, x)$ 
16:       $cible(x) \leftarrow v$ 
17:    end if
18:  end for
19: end for
```

Algorithm 7 Prim(G, w, r)

```
1:  $F \leftarrow FILE\_PRIORITE(V(G), cle)$ 
2: for all  $v$  de  $V(G)$  do
3:    $cle(v) \leftarrow \infty$ 
4: end for
5:  $cle(r) \leftarrow 0$ 
6:  $pere(r) \leftarrow NIL$ 
7: while  $F \neq \emptyset$  do
8:    $u \leftarrow EXTRAIRE\_MIN(F)$ 
9:   for all  $v \in Adj(u)$  do
10:    if  $v \in F$  et  $w(u, v) < cle(v)$  then
11:       $pere(v) \leftarrow u$ 
12:       $cle(v) \leftarrow w(u, v)$ 
13:    end if
14:  end for
15: end while
```

qui calculent les distances entre n'importe quelle paire de sommets en même temps.

Dans un graphe (orienté) pondéré, la *distance* entre des sommets u et v est définie comme la plus petite longueur d'une chaîne (d'un chemin) reliant u et v ; la longueur d'une chaîne d'arêtes (d'un chemin d'arcs) étant définie comme la somme des poids de ses arêtes (de ses arcs).

6.1 Graphes non-orientés

Nous allons commencer par l'algorithme de Dijkstra, un algorithme employé essentiellement pour calculer les distances dans les graphes non-orientés, mais qui marche aussi bien dans les graphes orientés, autant que les poids sur les arcs sont tous positifs.

L'algorithme de Dijkstra peut être considéré comme une généralisation de l'algorithme de parcours en largeur.

Algorithm 8 Dijkstra(G, w, s)

```

1: for all  $v$  de  $V(G)$  do
2:    $d(v) \leftarrow \infty$ 
3:    $pere(v) \leftarrow NIL$ 
4:    $couleur(v) \leftarrow BLANC$ 
5: end for
6:  $d(s) \leftarrow 0$ 
7:  $F \leftarrow FILE\_PRIORITE(\{s\}, d)$ 
8: while  $F \neq \emptyset$  do
9:    $pivot \leftarrow EXTRAIRE\_MIN(F)$ 
10:  for all  $e = (pivot, v)$  arc sortant de  $pivot$  do
11:    if  $couleur(v) = BLANC$  then
12:      if  $d(v) = \infty$  then
13:         $INSERER(F, v)$ 
14:      end if
15:      if  $d[v] > d[pivot] + w(e)$  then
16:         $d[v] \leftarrow d[pivot] + w(e)$ 
17:         $pere[v] \leftarrow pivot$ 
18:      end if
19:    end if
20:  end for
21:   $couleur[pivot] \leftarrow NOIR$ 
22: end while

```

Un sommet v est colorié *BLANC* tant que la valeur de $d[v]$ risque d'être modifiée (diminuée) encore. Un sommet est colorié *NOIR* si la valeur de $d[v]$ est la plus petite parmi les sommet blancs, donc elle est considérée comme définitive.

Il est facile à observer qu'une valeur d est attribuée à $d[v]$ si et seulement s'il existe une chaîne (un chemin) reliant s et v de longueur égale à d . Par

conséquent, le contenu final du tableau $d[]$ correspond bien à les distances $d(s, v)$: Montrons, par récurrence par rapport à l'ordre croissant de $d(s, v)$, que $d[v] = d(s, v)$ pour tout $v \in V(G)$:

(Initialisation) Le sommet avec $dist(s, v)$ la plus petite possible est le sommet de départ s , pour lequel on a $dist(s, s) = d[s] = 0$.

(Propagation) Soit v un sommet à distance $d_v = dist(s, v)$. La longueur de toute chaîne reliant s et u est supérieur ou égale à d_v , en plus, il existe une chaîne les reliant de longueur exactement d_v . Soit C une chaîne optimale reliant s et v , soit u l'avant dernier sommet de C (le prédécesseur de v dans C). Il est clair que $dist(s, u) < dist(s, v)$. Par l'hypothèse de récurrence, on a $d[u] = dist(s, u)$.

Considérons maintenant la valeur de $d[v]$. Au moment où u est le sommet pivot (au plus tard), $d[v]$ devient $d[u] + w(uv) = dist(s, u) + w(uv) = d_v$. Montrons maintenant que la valeur de $d[v]$ ne peut pas être inférieur à cela. Supposons le contraire. Si $d[v] < d_v$, alors il existerait un sommet u tel que

$$d_v > d[v] = d[u] + w(uv)$$

ce qui impliquerait l'existence d'une chaîne (d'un chemin) reliant s et v en passant par u de longueur totale inférieur à d_v , ce qui contredirait la définition de distance.

6.2 Graphes orientés sans circuit

L'algorithme de Dijkstra n'est pas garanti correct si dans un graphe orienté il y a des arcs de poids négatif.

Si dans un graphe orienté il y a des arcs de poids négatifs, la définition de distance a un sens seulement si le graphe ne contient aucun *circuit absorbant*, c-à-d circuit de poids total négatif. Si jamais un graphe orienté admet un circuit absorbant, en reprenant ce circuit répétitivement on peut construire des chemins de poids sans borne inférieure.

Un des cas où on peut garantir non-existence de circuits absorbants est le cas de graphes orientés sans circuit. Dans ce cas-là, il suffit d'appliquer le tri topologique et dans la suite, calculer les distances dans l'ordre correspondant. C'est exactement ce que fait l'algorithme de Bellmann, présenté ici avec deux versions.

La deuxième version de l'algorithme de Bellmann fait le travail de trier les sommets en même temps que le calcul des distances; il utilise un tableau supplémentaire $npred[]$ pour garder le nombre de prédécesseurs d'un sommets qui n'ont pas encore été traités par l'algorithme. La distance d'un sommet est déclarée définitive si tous ses prédécesseurs ont été considérés.

6.3 Graphes orientés avec circuits

L'algorithme suivant (celui de Ford) permet de détecter l'existence d'un circuit absorbant, et dans le cas où il n'y a pas, de calculer les distances à partir d'un sommet de départ s donné, dans un graphe orienté donné avec les arcs pondérés.

Algorithm 9 Bellman-court(G, w, s)

```
1: TriTopologique( $G$ )
2: Soit  $v_1, \dots, v_n$  l'ordre topologique calculé à l'étape 1
3: for  $i$  de 1 à  $n$  do
4:    $d[v_i] \leftarrow \infty$ 
5:    $pere[v_i] \leftarrow NIL$ 
6: end for
7:  $d[s] \leftarrow 0$ 
8: Soit  $j$  l'indice de  $s$  dans l'ordre topologique calculé à l'étape 1
9: for  $i$  de  $j$  à  $n - 1$  do
10:  for  $u \in Adj(v_i)$  do
11:    if  $d[u] > d[v_i] + w(v_i, u)$  then
12:       $d[u] \leftarrow d[v_i] + w[v_i, u]$ 
13:       $pere[u] \leftarrow v_i$ 
14:    end if
15:  end for
16: end for
```

Algorithm 10 Bellman-long(G, w, s)

```
1: for all  $v$  de  $V(G)$  do
2:    $d[v] \leftarrow \infty$ 
3:    $pere[v] \leftarrow NIL$ 
4:    $npred[v] \leftarrow deg^-[v]$ 
5: end for
6:  $d[s] \leftarrow 0$ 
7: INSERER_FILE( $F, s$ )
8: while  $F$  non vide do
9:    $u \leftarrow TETE\_FILE(F)$ 
10:  DEFILER( $F$ )
11:  for  $v \in Adj(u)$  do
12:    if  $d[v] > d[u] + w(u, v)$  then
13:       $d[v] \leftarrow d[u] + w[u, v]$ 
14:       $pere[v] \leftarrow u$ 
15:    end if
16:     $npred(v) \leftarrow npred[v] - 1$ 
17:    if  $npred(v) = 0$  then
18:      INSERER_FILE( $F, v$ )
19:    end if
20:  end for
21: end while
```

L'algorithme de Ford renvoie VRAI si le graphe G donné est sans circuit absorbant, et renvoie FAUX sinon. La partie essentielle de l'algorithme est une boucle, où, pour $i = 1, \dots, n - 1$, après l'étape numéro i , la valeur de $d[v]$ est la longueur du plus court chemin de s à v qui emprunt au maximum i arcs.

Si jamais il existe un sommet v tel qu'il existe un chemin de s à v empruntant au moins n arcs, tel qui est de longueur inférieur à la distance calculée après l'étape numéro $n - 1$, ce chemin-ci doit contenir un circuit absorbant, car il ne peut pas être élémentaire. D'où le critère implémenté sous forme de test à la fin de l'algorithme.

Algorithm 11 Ford(G, w, s)

```

1: for all  $v$  de  $V(G)$  do
2:    $d[v] \leftarrow \infty$ 
3:    $pere[v] \leftarrow NIL$ 
4: end for
5:  $d[s] \leftarrow 0$ 
6: for  $i$  de 1 à  $n - 1$  do
7:   for tout arc  $e = (u, v) \in E(G)$  do
8:     if  $d[v] > d[u] + w(u, v)$  then
9:        $d[v] \leftarrow d[u] + w(u, v)$ 
10:       $pere[v] \leftarrow u$ 
11:    end if
12:  end for
13: end for
14: for tout arc  $e = (u, v) \in E(G)$  do
15:   if  $d[v] > d[u] + w(u, v)$  then
16:    return FAUX
17:   end if
18: end for
19: return VRAI

```

6.4 Plus courts chemins entre toutes paires de sommets

Nous allons présenter un algorithme (celui de Floyd) calculant les distances entre toutes paires de sommets en même temps.

Après l'exécution de l'algorithme, la matrice $W_{i,j}$ contiendra la plus courte distance entre le sommet i et le sommet j , et la matrice $Pred_{i,j}$ le sommet prédécesseur de j sur un plus court chemin de i à j .

L'algorithme calcule les distances en n étapes. Après l'étape k ($k = 1, \dots, n$), la matrice $W_{i,j}$ contient la longueur du plus court chemin entre le sommet i et le sommet j , ayant comme sommets intermédiaires des sommets numéro au plus k . À l'étape k , l'algorithme choisit entre le chemin actuel de i à j (qui ne passe que par des sommets numéro inférieur à k) et la concaténation des chemins de i à k et de k à j (ne passant que par des sommets numéro inférieur à k) celui qui

est le plus court. Si c'est celui qui passe par k , l'information sur le prédécesseur de j dans un chemin optimal de i à j est mise à jour – le prédécesseur dans le chemin de k à j est privilégié.

Algorithm 12 Floyd(G, w)

```

1: for  $i$  de 1 à  $n$  do
2:   for  $j$  de 1 à  $n$  do
3:     if  $i = j$  then
4:        $W(i, j) \leftarrow 0$ 
5:        $Pred(i, j) \leftarrow i$ 
6:     else if  $ij \in E(G)$  then
7:        $W(i, j) \leftarrow w(i, j)$ 
8:        $Pred(i, j) \leftarrow i$ 
9:     else
10:       $W(i, j) \leftarrow \infty$ 
11:       $Pred(i, j) \leftarrow NIL$ 
12:    end if
13:  end for
14: end for
15: for  $k$  de 1 à  $n$  do
16:   for  $i$  de 1 à  $n$  do
17:     for  $j$  de 1 à  $n$  do
18:       if  $W(i, k) + W(k, j) < W(i, j)$  then
19:          $W(i, j) \leftarrow W(i, k) + W(k, j)$ 
20:          $Pred(i, j) \leftarrow Pred(k, j)$ 
21:       end if
22:     end for
23:   end for
24: end for

```

7 Optimisation de flot

Un *réseau de flot* $(G, s, t, c())$ est un graphe orienté G , dans lequel on spécifie deux sommets spéciaux : un sommet source s (sommet sans arc entrant) et un sommet destination (puits) t (sommet sans arc sortant); tout arc e est doté d'une capacité $c(e)$.

Un *flot* sur un réseau de flot $(G, s, t, c())$ est une fonction $f : E(G) \rightarrow \mathbb{R}^+$ qui attribue une quantité de flot à chaque arc de graphe G , de façon à ce que

- les flots sur les arcs ne dépassent pas les capacités :

$$\forall e \in E(G) \quad f(e) \leq c(e);$$

- pour tout sommet interne de réseau (autre que la source et la destination),

la quantité de flot entrant est égale à la quantité de flot sortant :

$$\forall v \in V(G), v \neq s, v \neq t \quad \sum_{uv \in E(G)} f(uv) = \sum_{vw \in E(G)} f(vw).$$

Si f est un flot sur un graphe G , alors le sommet s est le seul sommet où le flot entre dans le graphe, et le sommet t est le seul sommet où le flot sort du graphe. Puisque le flot n'est ni créé ni détruit ailleurs, la quantité de flot qui apparaît au sommet s doit être égale à la quantité de flot qui disparaît au sommet t .

Proposition 9 *Soit f un flot sur un réseau de flot $(G, s, t, c())$. Alors*

$$\sum_{sw \in E(G)} f(sw) = \sum_{ut \in E(G)} f(ut).$$

Preuve.

$$\sum_{e \in E(G)} f(e) = \sum_{v \in V(G)} \sum_{vw \in E(G)} f(vw) = \sum_{sw \in E(G)} f(sw) + \sum_{\substack{v \in V(G) \\ v \neq s \\ v \neq t}} \sum_{vw \in E(G)} f(vw),$$

mais aussi

$$\sum_{e \in E(G)} f(e) = \sum_{v \in V(G)} \sum_{uv \in E(G)} f(uv) = \sum_{ut \in E(G)} f(ut) + \sum_{\substack{v \in V(G) \\ v \neq s \\ v \neq t}} \sum_{uv \in E(G)} f(uv).$$

D'où

$$\sum_{sw \in E(G)} f(sw) + \sum_{\substack{v \in V(G) \\ v \neq s \\ v \neq t}} \sum_{vw \in E(G)} f(vw) = \sum_{ut \in E(G)} f(ut) + \sum_{\substack{v \in V(G) \\ v \neq s \\ v \neq t}} \sum_{uv \in E(G)} f(uv),$$

et comme $\sum_{vw \in E(G)} f(vw) = \sum_{uv \in E(G)} f(uv)$ pour tout $v \in V(G)$, $v \neq s$, $v \neq t$,

l'égalité demandée en suit directement. \square

La valeur décrivant à la fois la quantité totale de flot sortant de la source et de flot entrant en destination, s'appelle la *valeur* du flot f , et est notée

$$|f| = \sum_{sw \in E(G)} f(sw) = \sum_{ut \in E(G)} f(ut).$$

C'est la valeur de flot que l'on va essayer de maximiser.

L'idée de base de l'algorithme est la suivante : Tant qu'il existe au moins une amélioration possible, effectuer en une. Pour tester s'il existe une amélioration d'un flot actuel, une procédure de marquage est employée.

Initialement, le seul sommet marquée est la source s . Par la suite, tous les sommets marqués sont visités un par un, et le marquage est propagé dans le graphe en respectant les règles suivantes :

- Si vw est un arc sortant d'un sommet v marqué, et que $f(vw) < c(vw)$ (l'arc vw n'est pas encore saturé – il est possible d'augmenter le flot sur l'arc), marquer le sommet w (si ce n'est pas encore le cas).
- Si uv est un arc entrant d'un sommet v marqué, et que $f(uv) > 0$ (il est possible de diminuer le flot sur l'arc), marquer le sommet u (si ce n'est pas encore le cas).

Il existe une amélioration possible d'un flot actuel si et seulement si à la fin de la procédure de marquage, le sommet destination t est marqué.

Si pendant la procédure de marquage on garde en plus pour chaque sommet v l'arc a permis de marquer le sommet v , au cas où une amélioration existe – où t est marqué – on peut retracer une chaîne améliorante de s à t .

Cette chaîne peut être composée des arcs dans le bon sens et des arcs dans le sens opposé. Tout arc dans le bon sens permet d'augmenter le flot actuel sur l'arc d'un $\varepsilon > 0$, et tout arc dans le sens opposé permet de diminuer le flot actuel sur l'arc d'un $\varepsilon > 0$. Si cette opération (augmentation ou diminution) est effectuée sur chaque arc de la chaîne en même temps, on obtiendra un nouveau flot (les deux conditions définissant un flot sont conservées) de valeur globale augmentée de ε .

Comme pas d'amélioration ε on peut choisir le minimum parmi la plus petite différence $c(e) - f(e)$ pour les arcs dans le bon sens et la plus petite valeur $f(e)$ pour les arcs dans le sens opposé, de la chaîne améliorante.

Soit (G, s, t, c) un réseau de flot. Une s, t -coupe est une partition (S, T) des sommets de G avec $s \in S$ et $t \in T$. La capacité d'une telle coupe, notée $c(S, T)$, est définie comme la somme des capacités de tous les arcs reliant un sommet de S à un sommet de T :

$$c(S, T) = \sum_{\substack{uv \in E(G) \\ u \in S, v \in T}} c(uv)$$

Observer que les arcs dans le sens opposé (les arcs de T à S) ne sont pas comptés.

Proposition 10 *Soit f un flot dans un réseau de flot (G, s, t, c) et soit (S, T) une s, t -coupe. Alors $|f| \leq c(S, T)$.*

Preuve. Il suffit de considérer les arcs entre S et T . On a

$$|f| = \sum_{\substack{uv \in E(G) \\ u \in S, v \in T}} f(uv) - \sum_{\substack{vu \in E(G) \\ u \in S, v \in T}} f(uv) \leq \sum_{\substack{uv \in E(G) \\ u \in S, v \in T}} f(uv) \leq \sum_{\substack{uv \in E(G) \\ u \in S, v \in T}} c(uv) = c(S, T).$$

Autrement dit, s'il existe une s, t -coupe de capacité c , on ne peut transporter plus que c . \square

Theorème 3 (Ford-Fulkerson) Soit (G, s, t, c) un réseau de flot. Un flot f est optimal (maximal) si et seulement s'il existe une s, t -coupe (S, T) tel que $|f| = c(S, T)$. Une telle coupe est donc forcément minimale. A la sortie de l'algorithme de Ford-Fulkerson, une coupe minimale est définie par le marquage de sommets.

Algorithm 13 Ford-Fulkerson(G, s, t, c)

```

1: Marquage()
2: while le sommet destination  $t$  est marqué do
3:   Améliorer()
4:   Marquage()
5: end while

```

Algorithm 14 Marquage()

```

1: for chaque sommet  $v$  do
2:   démarquer( $v$ )
3: end for
4: marquer( $s$ )
5: enfiler( $s$ )
6: while file non vide do
7:    $u \leftarrow$  tête de la file
8:   for chaque voisin sortant  $v$  de  $u$  non-marqué do
9:     if  $f(uv) < c(uv)$  then
10:      marquer( $v$ )
11:      enfiler( $v$ )
12:       $\pi(v) \leftarrow uv$ 
13:     end if
14:   end for
15:   for chaque voisin entrant  $v$  de  $u$  non-marqué do
16:     if  $f(vu) > 0$  then
17:      marquer( $v$ )
18:      enfiler( $v$ )
19:       $\pi(v) \leftarrow vu$ 
20:     end if
21:   end for
22:   défiler( $u$ )
23: end while

```

Algorithm 15 Améliorer()

```
1:  $v \leftarrow t$ 
2:  $val \leftarrow \infty$ 
3: while  $v \neq s$  do
4:   if  $\pi(v) = uv$  then
5:      $val \leftarrow \min\{val, c(uv) - f(uv)\}$ 
6:   end if
7:   if  $\pi(v) = vu$  then
8:      $val \leftarrow \min\{val, f(uv)\}$ 
9:   end if
10:   $v \leftarrow u$ 
11: end while
12:  $v \leftarrow t$ 
13: while  $v \neq s$  do
14:   if  $\pi(v) = uv$  then
15:      $f(uv) \leftarrow f(uv) + val$ 
16:   end if
17:   if  $\pi(v) = vu$  then
18:      $f(vu) \leftarrow f(vu) - val$ 
19:   end if
20:   $v \leftarrow u$ 
21: end while
```
