Graphes

Informatique MP2I

Victor, ROBERT

1. Notions élementaires

1.1. Graphes orientés

Définition

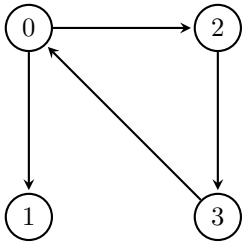
Un *graphe orienté* G est un couple (V, E) avec
- V un ensemble de *sommets* non vide
- $E \subseteq V^2$ un ensemble d'*arcs*

Un arc (u, v) va du sommet u au sommet v.

E ne contient pas de couples (u, u) (pas de boucles)

Exemple

Si $V = \{0, 1, 2, 3\}$ et $E = \{(0; 1), (0; 2), (1; 2), (2; 3), (3; 0)\}$



Remarque

Si (u, v) est un arc alors u est un prédecesseur de v et v est un successeur de u.

Remarque

On peut ajouter des étiquettes sur les arcs.

1.2. Graphes non orientés

Définition

Un *graphe non orienté* est un couple G de deux ensembles V et E.
- V est un ensemble de sommets
- E est un ensemble d'*arêtes* avec E ensemble de parties de V de taille 2

On considère que E ne contient pas d'âretes de la forme $\{x, x\}$

Remarque

Un graphe non orienté $G = (V, E)$ correspond toujours à un graphe orienté $G' = (V', E')$ avec

- $V = V'$
- $\forall \{x, y\} \in E, \quad (x, y) \in E'$ et $(y, x) \in E'$

Quelques familles de graphes non orientés

- Les graphes complets: notés K_n, il contient toutes les arêtes possibles
- Les graphes cycles: notés C_n, les arêtes sont les $\{i, i+1 \mod n\}$ pour tout $i \in V$

1.3. Représentation en machine

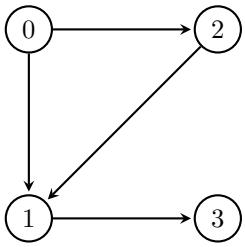
Pour simplifier on considère toujours que $V = [\![0, n-1]\!]$.

1.3.1. En utilisant une matrice d'adjacence

La matrice d'adjacence est une matrice $n \times n$ contient soit des 0 et des 1, soit des booléens.

$$A_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \text{ ou } \{i,j\} \in E \\ 0 & \text{sinon} \end{cases}$$

Exemple



On a la matrice d'adjacence:

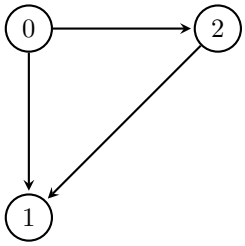
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

1.3.2. En utilisant une liste d'adjacence

On représente le graphe par un tableau de taille n qui contient des listes.

Si $(x, y) \in E$ alors y est dans la liste $\tau.(x)$

Si $\{x, y\} \in E$ alors y est dans la liste $\tau.(x)$ et x est dans la liste $\tau.(y)$



1.4. Degré, Densité

Définition

Dans un graphe non orienté, le *degré* d'un noeud x, noté $\deg(x)$ est son nombre de voisins.

Dans un graphe orienté, on distingue de le *degré entrant*

$$d_- = \Big|\{y|(y, x) \in E\}\Big|$$

du *degré sortant*

$$d_+ = \Big|\{y|(x, y) \in E\}\Big|$$

On peut aussi définir le *degré total* $d(x) = d_+(x) + d_-(x)$

Remarque

Dans un graphe non orienté à n sommets, on a:

$$\forall x \in V, \quad 0 \leq \deg(x) \leq n - 1$$

Si le graphe est orienté:

$$\forall x \in V, \quad 0 \leq d(x) \leq 2n - 2$$

Propriété

Dans un graphe orienté $G = (V, E)$, on a:

$$|E| \leq |V| \cdot \Big(|V| - 1\Big)$$

Preuve

Méthode 1 Par dénombrement des 2-listes sans répétitions.

Méthode 2

$$|E| = \sum_{x \in V} d_+(x) \leq \sum_{x \in V} (|V| - 1) = |V| \cdot \Big(|V| - 1\Big)$$

Remarque

Pour compter les arcs, on somme les degrés sortants (ça fonctionne aussi avec les degrés entrants). Si on somme les degrés entrants les les degrés sortants, c'est à dire les degrés totaux, on obtient le double du nombre d'arêtes.

Propriété

Dans un graphe non orienté $G = (V, E)$, on a:

$$|E| \leq \frac{|V| \cdot \Big(|V| - 1\Big)}{2}$$

Preuve

Méthode 1 Par dénombrement des 2-combinaisons.

Méthode 2 On a:

$$2|E| \leq \sum_{x \in V} d(x) \leq \sum_{x \in V} |V| \cdot \Big(|V| - 1\Big) \implies |E| \leq \frac{|V| \cdot \Big(|V| - 1\Big)}{2}$$

Remarque

On pourra considérer que le nombre d'arcs ou d'arêtes est dans le pire cas de l'ordre de grandeur de $|V|^2$.

Le nombre d'arêtes ou d'arcs peut varier entre 0 et les bornes établies.

Définition

On dira qu'un graphe est *creux* si son nombre d'arêtes est très petit devant le maximum.

Un graphe qui est proche d'avoir $\frac{|V|(|V|-1)}{2}$ arêtes est dit *dense*.

Remarque

L'efficacité de la représentation d'un graphe est affecté par sa densité.

Une matrice d'adjacence est de taille $|V| \cdot |V|$. Cela entraine souvent que les algorithmes sont en $O(|V|^2)$ temporellement.

Une liste d'adjacence contient au total $2|E|$ valeurs (graphe non orienté) ou $|E|$ valeurs (graphe orienté). La complexité temporelle pourra être linéaire en $O(|V|)$.

1.5. Sous-graphes

Définition

Soit $G = (V, E)$ un graphe. Un *sous-graphe* $G' = (V', E')$ est un graphe tel que $V' \subseteq V$ et $E' \subseteq E$ avec aucun arête dont l'une des extrémités n'est pas dans V'.

Définition

Le *sous-graphe induit* par $V' \subseteq V$ est le graphe $G' = (V', E')$ avec:

$$E' = \Big\{(x, y) \in E \text{ ou } \{x, y\} \in E \mid x \in V' \text{ et } y \in V'\Big\}$$

C'est en gros le sous-graphes tel que les arêtes de V' sont en contact.

Exercices

Combien y-a-t'il de sous-graphes induits possibles pour un graphe $G = (V, E)$?

Autant de sous-graphes induits que de parties de V donc $2^{|V|}$

Encadrer le nombre de sous-graphes de G non orienté.
Le graphe complet à

$$\sum_{1 \leq k \leq n} \binom{n}{k} 2^{\frac{k(k-1)}{2}}$$

2. Accessiblité

Un problème principal dans les graphes est *"trouver un chemin entre x et y"* qui admet de nombreuses variantes comme *"trouver un plus court chemin"* ou *"compter le nombre de chemins"*.

2.1. Chemins

Définition

Dans un graphe $G = (V, E)$ un chemin de longueur n est une suite $(z_0, ..., z_n)$ de n+1 sommets tels que

$$\forall i \in [\![0, n-1]\!],\ (z_i, z_{i+1}) \in E$$

.
z_0 et z_n sont *les extrémités* du chemin.

Remarque

Un chemin de longueur n contient n+1 sommets et n arêtes/arcs.

Un chemin dont les extrémités sont z_0 et z_n est un chemin de z_0 à z_n.

Définition

Un chemin est dit
- *élementaire* s'il ne pase pas deux fois par le même sommet (à part les extrémités)
- *simple* s'il ne passe pas deux fois par la même arête (ou le même arc)

Définition

Dans un graphe non-orienté, un cycle est un chemin simple, de longueur supérieure ou égale à 1, dont les deux extrémités sont les mêmes.

Définition

Dans un graphe orienté, un circuit est un chemin dont les extrémités sont les mêmes.

Remarque

Un cycle ou un circuit est dit élementaire si aucun sommet n'apparait deux fois sauf l'extrémité.

Propriété

Il existe un nombre fini de chemins élementaires.
Il existe un nombre fini de chemins simples.

Preuve

On note $n = |V|$

Alors la longueur maximale d'un chemin élementaire est n. Supposons par l'absure l'existence d'un chemin de longueur $m > n$. Mais alors il existe $z_0, ..., z_m$ n+1 sommets tous différents sauf z_0 et z_n. Or $m > n$, c'est absurde.

Dans le graphe complet à n noeuds, $0 \rightsquigarrow 1 \rightsquigarrow ... \rightsquigarrow n-1 \rightsquigarrow 0$ est un chemin élementaire de longueur n. Il n'est pas possible de faire une infinité de chemins de longueur bornée par n.

Même argument pour les chemins simples sauf que c'est le nombre d'arêtes qui borne la longueur.

Propriété

Il y a équivalence entre:

(i) Il existe un chemin de x à y
(ii) Il existe un chemin élementaire de x à y
(iii) il existe un plus court chemin de x à y
(iv) il existe un chemin simple de x à y

Preuve

(iv) $\implies$ (i): évident

Montrons (i) $\implies$ (ii).

Soit $z_0 \rightsquigarrow ... \rightsquigarrow z_n$ un chemin de x à y. Donc il existe un sommet qui se répète, i.e. il existe $i \neq j$ tel que $z_i = z_j$ et $(i, j) \neq (0, n)$. On a:

$$z_0 \rightsquigarrow ... \rightsquigarrow \boxed{z_i \rightsquigarrow ... \rightsquigarrow z_{j-1} \rightsquigarrow z_i} \rightsquigarrow z_{j+1} \rightsquigarrow ... \rightsquigarrow z_n$$

Que l'on raccourcis en:

$$z_0 \rightsquigarrow ... \rightsquigarrow \boxed{z_i} \rightsquigarrow z_{j+1} \rightsquigarrow ... \rightsquigarrow z_n$$

On a supprimé une répétition. On recommence jusqu'à ce que le chemin soit élementaire. Ce processus termine, on fait disparaitre 1 répétition et on n'en fait pas apparaitre.

Montrons (ii) $\implies$ (iii).

On considère l'ensemble $\mathcal{A}$ des longueurs des chemins élementaires entre x et y. $\mathcal{A}$ est une partie de $\mathbb{N}$ non vide donc $\mathcal{A}$ admet un minimum. Il existe donc un chemin élementaire de longueur minimale.

Montrons (iii) $\implies$ (iv).

Supposons l'existence d'un plus court chemin entre x et y.
On a montré que ce chemin est élementaire donc il est simple.

2.2. Composantes connexes

Dans cette partie on se place dans un graphe non-orienté.

Définition

Un sommet y est *accessible* à partir de x s'il existe un chemin entre les deux.

Définition

On définit: $\forall (x,y) \in V^2,\ x\mathcal{R}y \iff y$ est accessible à partir de x

On appelle *composantes connexes* les classes d'équivalences de la relation $\mathcal{R}$.

Autrement dit, les composantes connexes sont des parties $\mathcal{C}_1, ..., \mathcal{C}_p \subseteq V$ telles que:

$$V = \bigsqcup_{1 \leq i \leq p} \mathcal{C}_i$$

et:

$$\forall i \in [\![1,p]\!] \quad \forall (x,y) \in \mathcal{C}_i^2, \quad x\mathcal{R}y$$

et:

$$\forall (i,j) \in [\![1,p]\!]^2 \quad \forall (x,y) \in \mathcal{C}_i \times \mathcal{C}_j, \quad x\mathcal{R}y \implies x = y$$

Définition

Un graphe non-orienté est dit connexe si pour toute paire de sommets x et y, il y a un chemin de x à y.

Propriété

G est connexe $\iff$ G n'a que une seule composante connexe.

Remarque

On peut aussi appeller *"composantes connexes"* les sous-graphes induits par $\mathcal{C}_1, ...\mathcal{C}_p$.

Propriété

Le sous-graphe induit par $\mathcal{C}_i$ est connexe.

Propriété

Soit $G = (V, E)$ et soit $(x, y) \in V^2$ tels que $\{x, y\} \notin E$. On s'intéresse à $G' = \left(V, E' := E \cup \left\{\{x, y\}\right\}\right)$

Si x et y étaient dans deux composantes connexes différentes, alors G' a une composante connexe de moins que G (les mêmes que G avec $\mathcal{C}_x$ et $\mathcal{C}_y$ fusionnées

Si x et y sont initialement dans la même composante connexe, alors les composantes connexes de G' sont les mêmes que G.

Si à l'inverse on retire une arêt qui existe (notée $\{x, y\}$) alors on peut:

- Ne rien changer aux composantes connexes
- Soit sciender une des composantes connexes en $\mathcal{C}_x$, $\mathcal{C}_y$

Remarque

Si on retire une arête à G connexe (1 composante connexe), alors G' a 1 ou 2 composantes connexes.

2.3. Composantes fortement connexes

On étudie dans cette partie des graphes orientés.

Définition

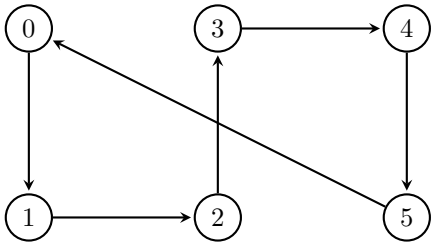
Un graphe $G = (V, E)$ est dit *fortement connnexe* si pour tout couple $(x, y) \in V$ il existe un chemin de x à y.

Remarque

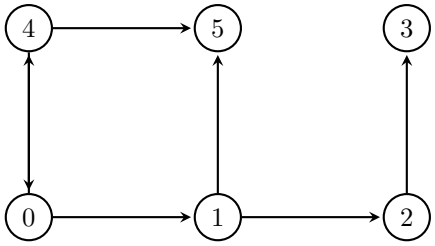
Si G est fortement connexe, alors G' non orienté obtenu en remplaçant tous les arcs par des arêtes est connexe.

Si G non-orienté est connexe alors G' orienté obtenu en remplaçant les arêtes par des arcs dans un sens arbitraire n'est pas toujours fortement connexe.

Exemple



est fortement connexe mais



n'est pas fortement connexe.

Définition

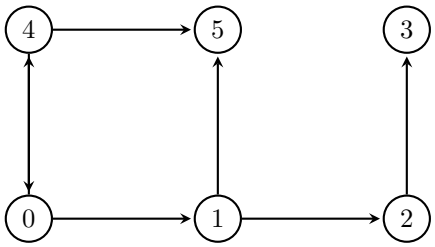
On définit une relation $\mathcal{R}'$ par $x\mathcal{R}'y \iff (x\mathcal{R}y)$ et $(y\mathcal{R}x)$.

$\mathcal{R}'$ est une relation d'équivalence.

Définition

On appelle *composante fortement connexes* les classes d'équivalences de la relation $\mathcal{R}'$.
Autrement dit la composante fortement connexe de x est l'ensemble des sommets y tels qu'on peut aller de x à y et de y à x.

Exemple



La composante fortement connexe de 0 est $\{0, 4\}$, celle de 5 est $\{5\}$.

Remarque

Deux sommets d'appartenant pas à la même composante peuvent être reliés par un arc.

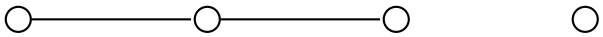
3. Graphes particuliers

3.1. Graphes non-orientés et acycliques

Définition

Un graphe est *acyclique* s'il ne contient pas de cycle.

Exemple



Définition

Un graphe non-orienté, acyclique est connexe est un *arbre*.

Attention Pas la même définition que les arbres du cours sur les arbres. Cependant si on choisit un noeud comme racine, alors il y a une bijection avec les arbres n-aires.

Remarque

On peut dire qu'on graphe non-orienté et acyclique est une forêt (des arbres formés par ses composantes connexes).

Propriété

Pour $G = (V, E)$, on note $n = |V|$ et $p = |E|$ alors les assertions suivantes sont équivalentes:

(i) G est un abre (G acyclique et connexe).
(ii) G acyclique et possède $n - 1$ arêtes.
(iii) G connexe est possède $n - 1$ arêtes.
(iv) Si on retire n'importe quelle arête, G n'est plus connexe.
(v) Pour tous $x, y \in V$ il existe un unique chemin élementaire de x à y
(vi) G est acyclique et rajouter n'importe quelle arête (qui n'existe pas dans E) crée un cycle.

Lemme 1

Si G est connexe, alors $p \geq n-1$

Preuve

Par récurrence sur p (à montrer pour tout n).

Lemme 2

Si G est acyclique, alors $p \leq n-1$.

Preuve

Montrons le par récurrence forte sur p. (pour chaque p on le montre pour tout graphe acyclique peu importe son nombre de noeuds).

Si p est nul:

Soit G acyclique à $n \neq 0$ noeuds et $p = 0$ arêtes. On a tout le temps $0 \leq n-1$. Soit $p \in \mathbb{N}$. Supposons la propriété vraie pour tout $k \leq p$. On veut la montrer pour $p+1$. Soit G un graphe acyclique a n noeuds et $p+1$ arêtes.

Si G a plusieurs composantes connexes non réduites à 1 sommet, on les note $\mathcal{C}_1, ..., \mathcal{C}_m$ avec $m > 1$. On note n_i et p_i le nombre d noeuds et d'âretes du graphe de la composante connexe: il existe au-moins deux p_i différents de 0. Donc toutes les composantes connexes ont moins de p arêtes, elles vérifient donc l'hypothèse de récurrence et elles sont toutes acycliques car G l'est, donc:

$$\forall i \in [\![1, m]\!],\ p_i \leq n_i - 1$$

Or $p+1 = \sum_{i=1}^{m} p_i$ et $n = \sum_{i=1}^{m} n_i$ donc:

$$p+1 = \sum_{1 \leq i \leq m} p_i \leq \sum_{1 \leq i \leq m} n_i - 1 \leq \sum_{1 \leq i \leq m} n_i - m \leq n - m \leq n - 1$$

Si G n'a qu'une seule composante connexe (ou plusieurs mais l'une d'entre elle contient les $p+1$ arêtes. Dans ce cas on prend une arête $\{x, y\}$ de G et on la supprime. On considère G' le graphe G sans $\{x, y\}$ (et si G contenait plusieurs composantes connexes on ne garde que celle qui a $p+1$ arêtes).

Fin de la preuve

On note $n' \leq n$ le nombre de noeuds de G'. G' a p arêtes.

Par l'absurde, si G' est connexe, alors il existe un chemin $x \rightsquigarrow y$. Ce chemin existe aussi dans G. Or dans G il y a une arête supplémentaire entre x et y donc il a un cycle: absurde. Donc G' n'est pas connexe.

Ainsi, G' a deux composantes connexes: $\mathcal{C}_x$ et $\mathcal{C}_y$. En notant n_x, p_x et n_y, p_y leur nombre de noeuds et d'arêtes. On trouve:

$$p = p_x + p_y \overset{H}{\leq} n_x - 1 + n_y - 1 \leq (n_x + n_y) - 2 \leq n - 2$$

donc

$$\boxed{p + 1 \leq n - 1}$$

Remarque

HR sur G' on obitent $p \leq n - 1$ ce qui n'est pas suffisant pour conclure.

Preuve de la propriété

(i) $\Longrightarrow$ (ii): G est un arbre donc acyclique et connexe.
Si on note $m = |V|$ et $p = |E|$ en utilisant le lemme 1, G est connexe donc $p \geq n-1$. En utilisant le lemme 2, G est acyclique donc $p \leq n-1$. Donc $p = n-1$ et G est acyclique: (ii).

(ii) $\Longrightarrow$ (iii): On divise G en ses composantes connexes $\mathcal{G}_1, ..., \mathcal{G}_m$.
Montrons que $m = 1$. Par hypothèse, G est acyclique et $p = n-1$. Alors $\mathcal{G}_1(n_1, p_1), ..., \mathcal{G}_m(n_m, p_m)$ sont acycliques. Par le lemme 2, on a:

$$\forall i \in [\![1, m]\!], \quad p_i \leq n_i - 1$$

Tous les $\mathcal{G}_i$ sont connexes, donc d'après le lemme 1 on a également:

$$\forall i \in [\![1, m]\!], \quad p_i \geq n_i - 1$$

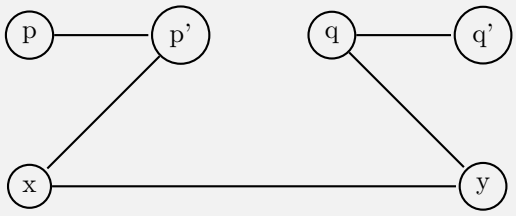
Donc on a $\forall i \in [\![1, m]\!], \quad p_i = n_i - 1$, or

$$n = \sum_{i=1}^{m} n_i \text{ et } p = \sum_{i=1}^{m} p_i = \sum_{i=1}^{m} \Big(n_i - 1\Big) = n - m$$

et on a supposé $p = n-1$ donc $m = 1$. Le graphe est connexe et $p = n-1$ (iii).

(iii) $\Longrightarrow$ (iv): Soit G connexe et $p = n-1$. Si on retire une arête, on obtient un graphe à $p-1$ arêtes et $n-1$ sommets. Or $p-1 < n-1$, donc par la contraposée du lemme 1, le graphe n'est pas connexe.

(iv) $\Longrightarrow$ (v): Supposons que retirer n'importe quelle arête à G le rend non connexe. Supposons par l'absurde que G ne vérifie pas (v), c'est à dire qu'il existe x et y reliés par deux chemins élementaires différents. Il existe une arête $\{p, q\}$ qui est sur un chemin et pas sur l'autre. Si on supprime $\{p, q\}$, le graphe n'est par hypothèse plus connexe. Les seuls sommets qui ont pu être déconnectés sont les paires (p', q') telles que le seul chemin qui les relie passe par (p, q).



Donc aucune paire n'a pu être déconnectée, absurde.

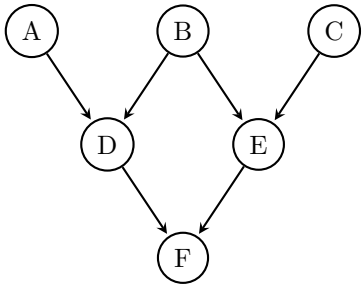
Fin de la preuve

(v) $\implies$ (vi): G est acyclique car un cycle contient x et y, alors il existe deux chemins élementaires entre x et y. Rajoutons $\{x, y\}$ quelconques. Il y avait un chemin élementaire entre x et y. On a crée un cycle.

(vi) $\implies$ (i): On suppose G acyclique et que rajouter n'importe quelle arête crée un cycle. Supposons G non connexe. Soient x et y pas dans la même composante. Si on rajoute l'arête $\{x, y\}$, alors on ne crée pas de cycles: absurde. Donc G est convexe et acyclique par hypothèse. Ainsi G est un arbre (i).

3.2. Graphes orientés acycliques

Dans cette partie, on étudie des graphes orientés sans circuit.



Lemme

Tout graphe orienté acyclique a au moins un sommet de degré sortant nul (feuille) et un sommet de degré entrant nul (racine).

Preuve

Soit $G = (V, E)$ un graphe orienté acyclique, on note $n = |V|$. Soient $x_1 \rightsquigarrow ... \rightsquigarrow x_r$ avec $r > n$ un chemin. Il existe par principe des tiroirs un noeud qui apparait deux fois. Donc il existe $1 \leq i < j \leq r$ tel que $x_i = x_j$. On considère le sous-chemin $x_i \rightsquigarrow ... \rightsquigarrow x_{j-1} \rightsquigarrow x_i$ qui est un circuit. C'est absurde. On en déduit que les chemins dans un graphe acyclique orienté sont de longueur majorée par $n - 1$. Il existe donc un plus grand chemin car l'ensemble des longueurs de chemin dans G est une partie de $\mathbb{N}$ majorée. On note $m \leq n - 1$ la longueur maximale d'un chemin. On peut considérer $x_1 \rightsquigarrow ... \rightsquigarrow x_{m+1}$ un chemin de longueur m. Alors x_1 est une racine car s'il existe $y \rightsquigarrow x_1$, alors le chemin n'est pas maximal, absurde. De la même manière, x_{m+1} est une feuille.

Définition

Un *ordre topologique* sur un graphe orienté acyclique G est un ordre $\prec$ sur ses sommets tel que si (u, v) est un arc, alors $u \prec v$.

Propriété

Un graphe orienté admet un ordre topologique, si, et seulement si il est acyclique.

Preuve

$\boxed{\Rightarrow}$ Soit G un graphe orienté admettant un ordre topologique $x_1 \prec ... \prec x_n$ avec $m = |V|$ et $V = \{x_1, ..., x_n\}$. On suppose par l'absurde l'existence d'un circuit $x_{i_1} \rightsquigarrow x_{i_2} \rightsquigarrow ... \rightsquigarrow x_{i_p} = x_{i_1}$. avec $i_1, ..., i_p - 1 \in [\![1, n]\!]$ distincts deux à deux. Il existe toujours un arc de $x_{i_j} \longrightarrow x_{i_{j+1}}$ pour $j \in [\![1, p-1]\!]$, donc $x_{i_j} \prec x_{i_{j+1}}$ et donc

$$x_{i_1} \prec x_{i_2} \prec ... \prec x_{i_{p-1}} \prec x_{i_1}$$

ce qui est impossible (ordre strict). Finalement, G est acyclique.

$\boxed{\Leftarrow}$ On montre par récurrence sur n le nombre de noeuds que G acyclique à n noeuds admet un ordre topologique.

Cas $n = 1$ l'ordre topologique est trivial, pas de comparaison nécessaire.

Supposons que c'est vrai à $n \in \mathbb{N}$ fixé. On veut la montrer pour $n+1$ On considère G acyclique à $n+1$ noeuds. **En géneral, il faut bien choisir le noeud à retirer.**

Ici G admet une racine d'après le lemme précedent. On nomme x_{n+1} cette racine. On considère G' le graphe G dont on a retiré x_{n+1}. On nomme $x_1, ..., x_n$ les noeuds de G' (de G). G' est acyclique car G l'est et a n noeuds.

On peut lui appliquer l'hypothèse de récurrence. Il existe $x_1 \prec ... \prec x_n$ un ordre topologique sur les noeuds de G' (quitte à renuméroter).
Alors $x_n + 1 \prec x_1 \prec ... \prec x_n$ est un ordre topologique pour G. En effet,

- Aucun arc de la forme (x_i, x_{n+1})
- Pour tout arc de la forme (x_{n+1}, x_j) on a bien $x_{n+1} \prec x_j$.

3.3. Graphes bipartis

Définition

Un *graphe biparti* est un graphe non orienté $G = (V, E)$ tel que V peut être séparée en $V_1 \cup V_2$ avec:

- $V_1 \cap V_2 = \emptyset$
- Si $\{x, y\} \in E$ alors $x \in V_i$ pour $i = 1$ ou 2 et $y \in \bar{V}_i$.

Définition

Un coloriage d'un graphe est l'attribution d'une couleur pour chaque sommet (souvent un numéro) avec la règle que deux voisins ne peuvent même couleur.

Propriété

On peut toujours colorier un graphe, par exemple à donnant à chaque noeud une couleur unique. On s'intéresse souvent à colorier le graphe avec le moins de couleurs possibles.

Définition

On dit qu'un graphe est k-coloriable s'il peut être colorié avec moins de k couleurs.

Remarque

Si un graphe contient un sous-graphe complet à k noeuds, alors il ne peut pas être colorié avec moins de k couleurs.

Propriété

Les graphes 1-coloriables sont les graphes sans arêtes.

Propriété

Les graphes 2-coloriables sont les graphes bipartis.

Preuve

$\boxed{\Leftarrow}$ Si G est bipartis, on peut colorier V_1 en une couleur et V_2 d'une autre.

$\boxed{\Rightarrow}$ Soit G 2-coloriable. Montrons qu'il est biparti. On définit V_1 par les sommets de la première couleur et V_2 par les sommets de la deuxième couleur. On a $V = V_1 \sqcup V_2$.
Si on considère une arête $\{x, y\}$ alors x et y ne sont pas de la même couleur (règle du coloriage) et donc un est dans V_1 et l'autre dans V_2.

4. Parcours de graphes

On veut se déplacer dans le graphe et visiter tous les sommets, en passant par les arêtes. C'est similaire au parcours d'arbres mais il y a des différences:

- il n'y a pas de notion de fils, on utilise par les successeurs ou les voisins
- il n'y a pas de racine
- il y a des cycles donc on peut tourner en rond, il faut se souvenir de là où l'on est passé

Remarque

On peut utiliser un tableau de booléens pour se souvenir de là où l'on est déjà passé.

4.1. Parcours en profondeur

Dans ce parcours, lorsqu'on a terminé d'explorer un noeud, on va en priorité vers ses voisins (vers la profondeur).

Pour étudier le parcours en profondeur on va étudier une version spéciale.

4.1.1. Version récursive

On suppose définis:

- un tableau `couleur` de taille $|V|$, initialisé à `[blanc, ...]` pour tous les sommets (`gris` si en cours d'exploration, `noir` si totalement exploré)
- deux tableaux `d` et `f` initialisés à `[0, ...]` et de taille $|V|$ pour noter de début et la fin de l'exploration d'un sommet
- une variable globale `temps` initialisée à `0`

Implémentation

```
Procédure ParcoursProfondeur(G, s) faire
    Couleur[s] <- gris
    d[s] <- temps

    temps <- temps + 1

    Pour chaque voisin/successeur u de s faire
        Si Couleur[u] = blanc faire
            ParcoursProfondeur(G, u)
        Fin
    Fin

    Couleur[s] <- noir
    f[s] <- temps
    temps <- temps + 1
Fin
```

Remarque

Cette méthode ne peut atteindre tous les sommets que si les graphe (dans le cas où il est non orienté) est connexe. On peut completer le parcours en profondeur pour qu'il aille sur chaque composante connexe.

Implémentation complète

```
Procédure ParcoursProfondeurComplet(G, s) faire
    Couleur := [blanc, ...]
    d, f := [0, ...]
    temps := 0

    Pour chaque s dans V faire
        Si Couleur[s] = blanc faire
            ParcoursProfondeur(G, s)
        Fin
    Fin
Fin
```

4.2.2. Version itérative

On utilise une pile.

Implémentation

```
Procédure ParcoursProfondeurItératif(G, s) faire
    à_voir := (* une pile vide *)
    vus := [false, ...]

    Empiler s sur à_voir

    Tant que à_voir n'est pas vide faire
        Dépiler un sommet s' de à_voir

        Si vus[s'] = false faire
            vus[s'] <- true

            Pour chaque voisin u de s' faire
                Si vus[u] = false, Empiler u sur à_voir
            Fin
        Fin
    Fin
Fin
```

Remarque

Dans le cours récursif, on n'est pas obligé de mettre de mettre `d` et `f` (et `temps`) et on peut utiliser des booléens à la place des couleurs puisqu'on ignore le `gris`.

On peut compléter le parcours en profondeur itératif de la même manière que le récursif.

Propriété

Les 3 procédures terminent:

- Un variant de `ParcoursProfondeur` est le nombre de sommets qui sont blancs (diminue strictement, quantité positive).
- Un variant de `ParcoursProfondeurItératif` est le nombre de sommets `i` tels que `vus[i] = false` ajouté au nombre de sommets dans la piles qui valent déjà `true`
- `ParcoursProfondeurComplet` termine car c'est une boucle pour bien définie.

Propriété

la complexité de `ParcoursProfondeur`, `ParcoursProfondeurComplet` et `ParcoursProfondeurItératif` est en $O(|V| + |E|)$

Preuve

Pour la version récursive: on ne lance au plus qu'un appel par noeud. Un noeud qui a déjà eu un appel est `gris` et on ne lance un appel que pour un noeud blanc.

Pendant chaque appel:

- On fait quelques opérations élementaires
- On fait une boucle sur tous les voisins, ce qui est comme faire une boucle sur toutes les arêtes (ou arcs) dont une extrémité est `s`. Si on somme sur tous les appels (tous les sommets dans le pire cas):

$$\begin{aligned}
&\sum_{s \in V} \Big(C^{st} + |\text{arêtes dont une extrémité est } s| \Big) \\
&= \sum_{s \in V} C^{st} + \sum_{s \in V} |\text{arêtes dont une extrémité est } s| \\
&= |V| \cdot C^{st} + 2|E| \\
&= O(|V| + |E|)
\end{aligned}$$

Remarque

Si on a une matrice d'adjacence, quand on cherche les voisins d'un noeud on parcourt tous les noeuds. La complexité est en $O(|V|)$.

Propriété

Soit $G = (V, E)$ un graphe et soient $u, v \in V$. On a soit:

$$d[u] < d[v] < f[v] < f[u]$$

soit:

$$d[u] < f[u] < d[v] < f[v]$$

Définition

Soit $G = (V, E)$ un graphe et soient $u, v \in V$.
On définit une fonction Π (prédecesseur dans le parcours) de la manière suivante:

$$\Pi\Big(u, v\Big) = u \iff \text{l'appel récursif } PP(G, v) \text{ a été lancé depuis } PP(G, u)$$

On note $u \longrightarrow_\Pi v$ si $\Pi(v) = u$
On note $u \overset{*}{\longrightarrow}_\Pi v$ si il existe un chemin $u_1 \rightsquigarrow_\Pi u_2 \rightsquigarrow_\Pi ... \rightsquigarrow_\Pi v$

Propriété

$$\forall (u, v) \in v^2 \quad u \overset{*}{\longrightarrow}_\pi v \iff d[u] < d[v] < f[v] < f[u]$$

Théroème du Chemin Blanc

Soient $u, v \in V^2$

$u \overset{*}{\longrightarrow}_\Pi v$ si, et seulement si, au temps $d[u]$ il existait un chemin de u à v constitué uniquement de sommets blancs.

Preuve

$\boxed{\Rightarrow}$ Supposons $u \xrightarrow{*}_{\Pi} v$.
Il existe donc un chemin

$$u \rightsquigarrow_{\Pi} u_1 \rightsquigarrow_{\Pi} u_2 \rightsquigarrow_{\Pi} ... \rightsquigarrow_{\Pi} u_{p-1} \rightsquigarrow_{\Pi} v$$

et d'après la propriété:

$$d[u] < d[u_1] < ... < d[u_{p-1}] < d[v] < f[v] < f[u_{n-1}] < f[u]$$

A la date $d[u]$ les sommets $u_1, ..., u_{p-1}, v$ étaient tous blancs car on a commencé à les étudier après u.
$\boxed{\Leftarrow}$ Supposons l'existence à la date $d[u]$ d'un chemin

$$u \rightsquigarrow u_1 \rightsquigarrow u_2 \rightsquigarrow ... \rightsquigarrow u_{p-1} \rightsquigarrow v$$

de sommets tous blancs (sauf u).

Supposons par l'absurde que

$$\neg u \xrightarrow{*}_{\Pi} v$$

.

$$u \rightsquigarrow u_1 \rightsquigarrow ... \rightsquigarrow u_{p-1} \rightsquigarrow v$$

On considère u_i le premier sommet en numéro du chemin tel que $\neg u \xrightarrow{*}_{\Pi} v$.

Pour $j < i$: $u \xrightarrow{*}_{\Pi} u_j$

Pour $j = i$: $\neg u \xrightarrow{*}_{\Pi} u_i$

On considère le plus petit $j < i$ tel qu'il existe une arête/un arc de u_j à u_i.

Si $d[u_i] < d[u_j] < f[u_j] < f[u_i]$. Or on sait que $d[u] < d[u_1]$ car u_i est blanc à $d[u]$. Ainsi

$$d[u] < d[u_j] < f[u_j] < f[u]$$

car $u \xrightarrow{*}_{\Pi} u_j$. . Mais alors

$$\boxed{d[u] < d[u_i]} < d[u_j] < f[u_j] < \boxed{f[u_i] < f[u]}$$

donc

$$d[u_i] < f[u_i] < d[u_i] < f[u]$$

d'où $u \xrightarrow{*}_{\Pi} v$: Absurde

Si $d[u] < f[u_i] < d[u_j] < f[u_j]$ on sait les mêmes choses et ça donne également une absurdité.

Propriété

A la fin de PP sur G et s un sommet est blanc s'il n'est pas atteignable et noir sinon.

4.2. Parcours en largeur

On utilise une file. Son principe est, lors de l'étude d'un sommet, on préfère étudier ses voisins avant de s'inéresser aux voisins de ses voisins.

Implémentation

```
Procédure ParcoursLargeur (G, s) faire
    f := (* une file vide *)

    (* Tous les tableaux sont de taille |V| *)
    vus := [false, ...]
    dist := [inf, ...]
    pi := [-1, ...]

    Enfiler s sur f

    vus[s] <- true
    dist[s] <- 0

    Tant que f n'est pas vide faire
        Défiler un sommet x de f

        Pour chaque voisin y de x faire
            Si vu[y] = false faire
                Enfiler y sur f

                vus[y] <- true
                dist[y] <- dist[x] + 1
                pi[y] <- x
            Fin
        Fin
    Fin
Fin
```

Le tableau `pi` permet de stocker le chemin que l'algorithme a parcouru.

Propriété

Cet algorithme termine.

Preuve

La boucle Pour termine car tout sommet a un nombre fini de voisins.
La boucle Tant que termine, le nombre de sommet qui sont encore à parcourir est un variant.

C'est une quantité positive.
On se place à une itération de la boucle, on note $n_{\mathcal{F}}$ la taille de la file et n_{false} le nombre de `false` au début de cette itération.
On note $m_{\mathcal{F}}$ et m_{false} les mêmes quantités après l'itération.
On note p le nombre de sommets ajoutés à la file après l'itération. On est assurés que durant l'itération, p cases de `vus` sont passées de `false` à `true`. En résumé:

$$m_{\mathcal{F}} = n_{\mathcal{F}} - 1 + p \textbf{ et } m_{false} = n_{false} - p$$

Ainsi $m_{\mathcal{F}} + m_{false} = n_{\mathcal{F}} + n_{false} - 1$. La quantité étudiée a diminuée strictement.

Propriété

La complexité de cet algorithme est en $O(|V| + |E|)$ ou $O(|V|^2)$ pour la représentation par matrice d'adjacence.

Preuve

La justification est similaire à celle pour le parcours en profondeur.

- chaque sommet a subi au plus un nombre constant d'opérations (ajouts, retrait, changer vu)
- chaque arête a été vu au plus 2 fois, car chacune de ses extrémités est vue au plus une fois.

Remarque

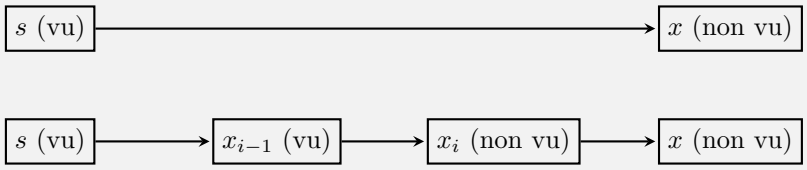
Un sommet peut ne pas être vu si il n'est pas accessible depuis s

Propriété

A la fin d'un parcours en largeur, dans le graphe G, depuis le sommet s, un noeud x est non-vu s'il n'est pas atteignable depuis s et vu sinon.

Preuve

Supposons l'existence de x un sommet atteignable depuis s mais `vu[x] = false` à la fin du parcours.
Soit $s = x_0 \rightsquigarrow ... \rightsquigarrow x_k = x$ un chemin de s à x.



Le sommet x_{i-1} a été vu, donc il a été enfilé. Comme l'algorithme ne s'arrêtermin que si la file est vide, il a été défilé. Donc x_i a été regardé, et à ce moment la `vu[x[i]]` vallait `false` donc x_i a été ajouté à la file et `vu[x[i]]` a été mis à `true`. C'est absure, x ne peut pas exister.
Supposons l'existence de y un sommet non atteignable mais tel que `vu[y] = true` à la fin. On considère y' le premier sommet temporellement parlant à avoir `vu[y'] = true` alors qu'il est non accessible. Le seul moment du code où `vu[y']` peut être mis à `true` est le moment où on explore les voisins d'un sommet défilé. Il existe donc un voisin de y' qu'on appellera y'', qui a été défilé avant que `vu[y']` soit mis à `true`. Or y'' n'est pas atteignable, sinon y' l'est et ceci contredit la définition de y' qui est le premier sommet non atteignable tel que `vu[y'] = true`. Absurde, y' n'existe pas et y non plus.

Définition

On définit la *distance entre deux points s et t* dans un graphe non pondéré (toutes les arêtes sont les mêmes) par:

$$\delta(s,t) = \begin{cases} \infty \text{ si on ne peut pas aller de } s \text{ à } t \\ \min \left\{ \text{ largeurs des chemins des sommets de } s \text{ à } t \right\} \text{ sinon} \end{cases}$$

Propriété

A la fin du parcours en largeur à partir du sommet s, on a pour tout sommet t

- $dist[t] = \delta(s,t)$
- Si $s \neq t$ et que $s \xrightarrow{*} t$ alors il existe un plus court chemin de s à t dont $\left\{\Pi[t], t\right\}$ est la dérnière arête

4.3. Applications

4.3.1. Calculer $\delta(s,t)$ avec s fixé, $\forall t$

On peut le faire avec un parcours en largeur.

4.3.2. Calculer les composantes connexes dans un graphe non orienté

Pour obtenir la composante connexe (liste des sommets) de s, il faut effectuer un parcours et regarder les sommets qui ont été vus.

Pour obtenir toutes les composantes connexes du graphe, on fait un parcours complet.

Remarque

Cette méthode ne permet pas de trouver des composantes fortement connexes.

4.3.3. Détection de cycles

On considère un graphe G non orienté et on veut savoir s'il contient un cycle.

Si lors d'un parcours on trouve un voisin vu qui n'est pas celui dont on vient alors on a un cycle.

Pour tester si un voisin est le sommet dont on viens:

- utiliser un parcours profondeur récursif et mettre en paramètre du parcours le sommet dont on vient.
- utiliser un parcours profondeur itératif en utilisant le tableau Π: si on est du sommet i, le sommet dont on vient est $\Pi[i]$

4.3.4. Détection de cycles sur les graphes orientés

Il peut y avoir un problème avec le parcours en profondeur si il y a un cycle.

On considère le parcours en profondeur récursif, la version qui colorie les noeuds en gris puis en noir.

Les arcs arrières sont les seuls à pouvoir former un circuit. Les arcs arrière se repèrent dans le parcours car lors du parcours, ils mènent à un sommet gris.

Les arcs avant et transveres, eux, mènent vers des sommets noirs. La condition *"pas le sommet dont on est venus"* n'existe plus.

Pour repérer un circuit, on utilise le parcours en profondeur récursif avec blanc/gris/noir. Si lors du parcours on croise un sommet gris, il y a un circuit.

4.3.5. Tris topologiques

Rappel

Pour un graphe orienté acyclique, un tri topologique est un ordre total sur les sommets tel que: si (u, v) est un arc, $u \prec v$.

On réalise le parcours en profondeur récursif avec date de début et date de fin.

Propriété

Les valeurs de `f` forment un tri topologique.

4.4. Plus courts chemins

4.4.1. Poids d'un chemin

Définition

Un graphe pondéré est la donnée d'un graphe $G = (V, E)$ et d'une fonction de poids

$$\omega : E \to P$$

On peut le noter $G = (V, E, \omega)$ et la plupart du temps $P = \mathbb{Z}$ ou $\mathbb{R}$

Remarque

On peut étendre ω à tous les arcs/arêtes, en donnant un poids infini aux arcs/arêtes qui ne sont pas dans E.

Définition

Le poids d'un chemin $c = x_0 \rightsquigarrow ... \rightsquigarrow x_n$ est:

$$\omega(c) = \sum_{i=0}^{n-1} \omega\Big((x_i, x_{i+1})\Big)$$

Définition

Si G n'admet pas de cycle (ou circuit) de poids négatif, on peut définir la plus courte distance entre x et y deux sommets:

$$d(x, y) = \min\{\ \omega(c) \mid c = x \rightsquigarrow ... \rightsquigarrow y\ \}$$

4.4.2. Algorithme de Floyd-Warshall

Cet algorithme calcule tous les $d(x, y)$ pour x et y sommets dans G sans cycles de poids négatif.

Cet algorithme utilise la programmation dynamique.

Soit x et y deux sommets, on note $n = |V|$ et soit $k < n$ (les sommets sont numérotés de 0 jusqu'à $n - 1$)

On note $d_k(x, y)$ la plus courte distance entre x et y sur les chemins qui n'utilisent que des sommets entre 0 et $k - 1$ sauf aux extrémités.

Si ce n'est pas possible d'aller de x à y en utilisant que des sommets de numéro $< k$, $d_k(x, y) = +\infty$.

La formule de récurrence pour x et y quelconques est la suivante:

$$d_0(x, y) = \omega(x, y) \text{ ou } +\infty$$
$$d_{k+1}(x, y) = \min\Big\{d_k(x, k) + d_k(k, y),\ d_k(x, y)\Big\}$$

Remarque

A k fixé, $d_k(x, y)$ est stocké comme une matrice (donc au minimum $O(n^2)$ en mémoire)
Dans cet algorithme, on peut toujours utiliser la matrice d'adjacence puisqu'on ne peut pas faire mieux que $O(n^2)$ en mémoire.
La matrice d'adjacence d'un graphe pondéré est une matrice $m \times n$ dans laquelle on indique les poids de chaque arcs/arêtes.

Implémentation

```
Fonction FloaydWarshallV1(Adj) faire
    d0 := Adj

    Pour k allant de 1 à n faire
        dk := copie(d0)

        Pour x allant de 0 à n - 1 faire
            Pour y allant de 0 à n - 1 faire
                dk[x][y] <- min (
                    d0[x][y],
                    d0[x][k-1] + d0[k-1][y]
                )
            Fin
        Fin

        d0 <- dk
    Fin

    Renvoyer d0
Fin
```

Remarque

On peut aussi le faire avec une seule matrice

Implémentation

```
FloaydWarshallV2(Adj) faire
    d := copie(Adj)

    Pour k allant de 1 à n faire
        Pour x allant de 0 à n - 1 faire
            Pour y allant de 0 à n - 1 faire
                d[x][y] = min (
                    d[x][y],
                    d[x][k-1] + d[k-1][y]
                )
            Fin
        Fin
    Fin

    Renvoyer d
Fin
```

Complexité

En temps: $O(n^3)$

En mémoire: $O(n^2)$

4.4.3. Algorithme de Dijkstra

Cet algorithme calcule les plus courts chemins de s fixé à n'importe quel sommet.

Remarques sur l'algorithme

- Il ne marche qu'avec $\omega \geq 0$ (cas le plus courant)
- Il est adapté à la représentation par listes d'adjacence
- C'est un algorithme de parcours

Pour l'algorithme de Dijkstra, on utilise un tas minimum à chaque étape on parcours le sommet le plus proche connu (en poids de chemin).

Dans la file de priorité, il nous faut deux informations:

- La distance
- Le sommet

Exemple

On applique le pseudo-code suivant:

```
Créer une tas f

Mettre (0, s) sur le tas

Tant que f n'est pas vide faire
    Extraire (d, x)

    Pour chaque voisin y pas vu faire
        Si la distance est meilleure que celle connue faire
            Mettre d + w(x, y) sur tas
        Fin
    Fin
Fin
```

Avoir le même sommet ajouté deux fois au tas n'est pas une bonne chose:

- à cause de la complexité
- parce que si on implémente le tas avec un tableau, il faut qu'on décide de sa taille au début de l'algorithme

On va ajouter une opération supplémentaire aux tas qui permet de modifier la distance qui lui est associée. Ainsi on a plus besoin d'ajouter plusieurs fois le même sommet au tas et donc la taille du tas est majorée par le nombre de sommets. Cette opération est appelée *"relâchement de la distance"*.

Pour implémenter cette opération dans un tas représenté par un tableau:

- trouver l'indice du tableau où se trouve le sommet dont on relâche la distance
- changer la distance si elle est plus petite
- percoler vers le haut afin de garder la structure de tas minimum

Exemple

Pour $n = 10$

$$\Big[(3,1);(4,5);(7,9);(5,3);(12 \overset{(ii)}{\rightarrow} 3,\mathbf{4});(8,7);\_;\_;\_;\_\Big]$$

On veut relâcher le sommet 4 à 3.

(i) on parcourt le tableau: le sommet 4 est dans la case 4
(ii) on change 12 en 3
(iii) on percole

On a alors:

$$\Big[(3,1);(4,3);(7,9);(5,3);(4,6);(8,7);\_;\_;\_;\_\Big]$$

La complexité est $O(n) + O(1) + O(\log n) = O(n)$

On peut faire mieux en se souvenant à chaque instant où se situe chaque sommet. On rajoute au tas un autre tableau qui à la case i indique soit -1 si ne sommet n'est pas dans le tas, soit la case j qui contient le sommet i. Pour l'exemple précedent, le tableau des localisations initial est:

$$\Big[-1;0;-1;3;\boxed{4};-1;1;5;\boxed{-1};2\Big]$$

Le sommet $\boxed{4}$ est localisé en 4 et le sommet $\boxed{8}$ n'est pas dans le tas.

On obtient en $O(1)$ l'information que le sommet 4 à la case 4. Le tableau des localisatoins après relâchement de 4 est:

$$\Big[-1;0;-1;3;\boxed{1};-1;4;5;\boxed{-1};2\Big]$$

Remarque

L'ajoute de la gestion du tableau de localisation ne change pas la complexité des opérations du tas.

L'opération de relâchement est un $O(\log n)$

Implémentation

```
Fonction Dijkstra(G, s) faire
    dist := [inf, ...] (* Taille n *)
    a_visiter := (* Une file de priorité*)

    dist[s] <- 0
    Ajouter (0, s) sur a_visiter

    Tant que a_visiter n'est pas vide faire
        Extraire (d, x) de a_visiter

        Pour chaque voisin y de x faire
            d' := d + w(x, y)

            Si d' < dist[y] faire
                Si dist[y] = inf faire
                    (* Pas dans de le tas *)
                    Ajouter (d', y) à a_visiter
                Sinon faire
                    Relacher la distance de y à d'
                Fin

                dist[y] <- d'
            Fin
        Fin
    Fin

    Renvoyer dist
Fin
```

Terminaison

La boucle *"Pour tout voisin"* termine car tout sommet possède un nombre fini de voisins.

La boucle *"Tant que"* termine, un variant est *"le nombre de sommets i tels que $dist[i] = +\infty$ + /a_visiter/"*

Ainsi l'algorithme termine.

Complexité

Pour un graphe G avec n sommets et p arêtes, alors:

- chacun des sommets subit au plus un ajout au tas, au plus un retrait du tas
- chacune des arêtes est considérée au plus deux fois. A chaque arête qu'on considère on effectue quelques comparaisons, quelques affectations dans un tableau et un relâchement
- il faut aussi compter la création de `a_visiter`

Au total, on a:

$$\boxed{n\log(n) + p \cdot (C^{st} + \log(n)) + O(n) = O((n+p)\log(n))}$$

Correction

Soit $G = (V, E, \omega)$ un graphe pondéré avec $\forall e \in E,\ \omega(e) \geq 0$, alors pour tout $s \in V$ l'appel $Dijkstra(G, s)$ renvoit un tableau $dist$ de taille $|V|$ tel que:

$$\forall x \in V,\ dist[x] = \delta(s, x)$$

On dira qu'un sommet est
- blanc si $dist[x] = \infty$
- gris si x a été ajouté au tas
- noir si x n'est pas dans le tas et que $dist[x] < \infty$

Pour chaque sommet x, on définit:

$$\begin{aligned} d(x) &= \delta(s, x) \\ d_n(x) &= \text{la longueur d'un plus court chemin de } s \text{ à } x \\ &\qquad \text{dont tous les sommets sont noirs} \end{aligned}$$

Boucle Tant que

Pour tout $x \in V$
- $dist[x] = d_n(x)$
- x est noir $\implies d_n(x) = d(x)$

I. Initialement, il n'y a pas de sommets noirs donc

$$d_n(x) = \begin{cases} 0 & \text{si } x = s \\ \infty & \text{sinon} \end{cases}$$

ce qui est l'initialisation $dist$

H. Supposons l'invariant vrai avant une itération de la boucle. On note j le sommet sorti de la file à cette itération.

Remarque

j est le seul sommet qui devient noir, seuls les voisins de j subissent un éventuel changement.

Pour tout sommet $x \neq j$ qui n'est pas voisin de j, la propriété est hérité immédiatement.

Pour le sommet j, on note $d_n(j)$ et $dist[j]$ les valeurs avant l'itération, $d'_n(j)$ et $dist'[j]$ les valeurs après l'itération.

Fin de la correction

On a $dist'[j] = dist[j] \overset{HR}{=} d_n(j)$. De plus $d'_n(j) = d_n(j)$ car avoir j comme nouveau sommet noir ne permet pas de créer un chemin plus court de s à j, donc $dist'[j] = d'_n(j)$.
On doit montrer $d_n(j) = d(j)$. On a $d_n(j) \overset{\text{déf}}{\geq} d(j)$.
Montrons que $d_n(j) \leq d(j)$. Il est suffisant de montrer que

$$\forall \text{chemin } c \text{ de } s \text{ à } j,\ d_n(j) \leq \omega(c)$$

Soit c un chemin de s à j.

- Si c n'a que des sommets noirs (sauf extrémités) alors $\omega(c) \geq d_n(j)$
- Sinon on considère le premier sommet blanc ou gris sur le chemin:

$$s \overset{c'}{\rightsquigarrow} x \overset{c''}{\rightsquigarrow} j$$

.
On sait que $\omega(c') \geq d_n(x)$. En particulier en remplaçant sans perte de généralité c' par un plus court chemin de sommets noirs de s à x, oa $\omega(c') = d_n(x)$. x ne peut pas être blanc car il est à coté d'un sommet noir. Donc x est gris.

x est dans le tas, or j a été sorti du tas, donc:

$$dist[j] \leq dist[x] \iff d_n(j) \leq d_n(x)$$

On rappelle que

$$\omega(c) = \omega(c') + \omega(c'') \geq \omega(c') \geq d_n(x)$$

Donc

$$\omega(c) \geq d_n(x) \geq d_n(j)$$

On a ainsi montré que

$$\boxed{d_n(j) = d(j)}$$

Pour les sommets x voisins de j:

- Si ils sont noirs, rien n e se passe et par HR la propriété est vraie.
- Si ils sont blancs, $dist[x] = \infty$ et $dist[x'] = dist[j] + w(j, x)$

Or $s \overset{\text{noirs}}{\rightsquigarrow} j$ (noir) $\overset{1}{\rightsquigarrow} x$ est un chemin de sommets noirs de s à x, c'est le seul qu'on connait (donc qui existe) et son poids est bien $dist[j] + \omega(j, x)$.

A la fin x est gris et n'a pas besoin de vérifier le deuxième point s'ils sont gris.

Fin de la correction

Lemme

Les chemins noirs de s à x qui passent par j mais tel que j n'est pas l'avant dernier sommet ne sont pas optimaux (on ne les considère pas).

Il nous faut donc étudier uniquement les chemins de la forme

$$s \overset{\text{noirs}}{\rightsquigarrow} j \overset{1}{\rightsquigarrow} x$$

On choisit sans perte de géneralité la partie gauche des chemins la plus courte possible. Les chemins de cette forme sont de taille

$$d_n(j) + \omega(j, x) = dist[j] + \omega(j, x)$$

- Si $dist[j] + \omega(j, x) < dist[x] = d_n(x)$, l'algorithme modifie $dist[x]$ et on a bien $d_n(x) = d_n(j) + \omega(jx)$
- Sinon, ce type de chemin n'est pas plus court qu'un chemin noir ne passant pas par j dopnc $d_n(x) = d'_n(x)$

L'algorithme de change pas $dist[x]$, c'est le comportement attendu.

A la fin de la boucle tant que, les sommets noirs sont tous les sommets accessbiles depuis s sont noirs (on peut faire la même preuve que pour le parcours en largeur).
On conclut que pour tout sommet x accessible, $dist[x] = d(x)$