Distance dans les Graphes

Scribes : Julien PROVILLARD Lionel RIEG

14 Mars 2007

1 Définitions et variantes

La distance se définit pour des graphes $G = (V, E)$ orientés ou non, pondérés ou non (les poids sont alors placés soit sur les arcs, soit sur les arêtes). On définit habituellement une fonction de poids $\omega : E \to \mathbb{R}$ qu'on peut naturellement étendre en $\omega : V \times V \to \mathbb{R}$ par $+\infty$ pour tous les couples $(x, y) \in V \times V \setminus E$.

Définition 1.1 *Soit $G = (V, E)$ un graphe pondéré par la fonction de poids ω. Soit $(x, y) \in V^2$, On définit* $\mathrm{dist}(x, y)$ *par :*

$$\mathrm{dist}(x, y) = \min_{\substack{(x=x_1,\ldots,x_k=y) \\ \text{chemin de } x \text{ à } y \text{ dans } G}} \sum_{1 \leq i < k} \omega(x_i, x_{i+1})$$

Remarque 1.1 *Soit $G = (V, E)$ un graphe pondéré. On a :*

$$\forall (x, y) \in V^2, \mathrm{dist}(x, y) > -\infty$$

si et seulement s'il n'existe pas de cycle ou de circuit de poids strictement négatif dans G.

2 Plus court chemin à origine unique.

Les algorithmes que nous allons voir sont définis pour des graphes orientés mais s'adaptent sans problèmes aux graphes non orientés.

FIG. 1 – Passage d'un graphe non orienté à un graphe orienté

2.1 Graphes orientés avec poids positifs ou nuls.

Algorithme de Dijkstra

Il est basé sur le même principe que l'algorithme de Prim (pour la recherche d'arbres couvrants de poids minimal dans un graphe).

Dans l'algorithme de Prim, on effetue des mises à jour du type

$$\forall y \in N(x),\ c(y) \leftarrow \min(c(y), \omega(x, y))$$

Dans l'algorithme de Dijkstra, les mises à jour sont du type

$$\forall y \in N(x),\ c(y) \leftarrow \min(c(y), c(x) + \omega(x, y))$$

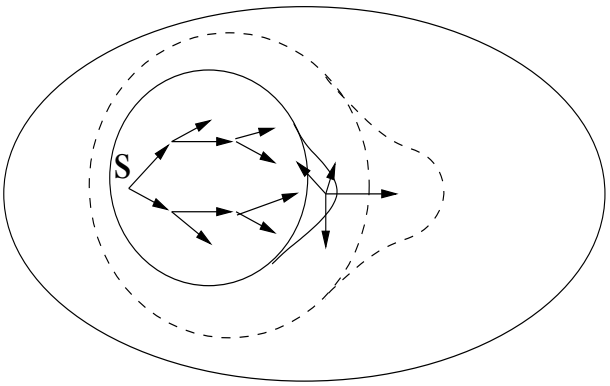

FIG. 2 – Une étape de Dijkstra

Complexités possibles :
- $\mathcal{O}(m + n \cdot log(n))$ avec les tas de Fibonacci
- $\mathcal{O}(m \cdot log(n))$ avec les tas binaires
- $\mathcal{O}(n^2)$ avec les listes ou les tableaux

Remarque 2.1 *Si ω est constante et positive (typiquement 1), on peut faire un parcours en largeur (BFS) en $\mathcal{O}(n + m)$.*

2.2 Graphes orientés acycliques (*ie* sans circuits) et fonction de poids quelconque.

On s'intéresse à la reconnaissance de graphes orientés sans circuits.

Lemme 2.1 *Si $G = (V, E)$ est orienté et sans circuit, alors il existe $x \in V$ tel que* $\deg^-(x) = |N^-(x)| = 0$.

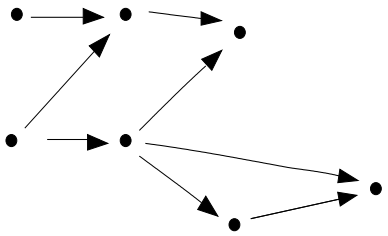

FIG. 3 – Un graphe orienté sans cycle

Démonstration :

Soit $G = (V, E)$ un graphe orienté sans circuit.

Considérons un chemin $\mu = (x_1, \ldots, x_k)$ de G de longueur maximum[1] (il existe car G est fini et sans cycle).

Supposons $\deg^-(x_1) > 0$, d'où $\exists\, y \in N^-(x_1)$.

Ou bien $y \notin \{x_2, \ldots, x_k\}$, ce qui est absurde car $(y, x_1, \ldots, x_k)$ serait un chemin prolongeant μ.

Ou bien $\exists\, i \in \{2, \ldots, k\}$ tel que $y = x_i$, ce qui est absurde car on aurait un chemin $(y, x_1, \ldots, x_i)$ donc $(x_1, \ldots, x_i)$ formerait un cycle alors que G est supposé sans cycle.

Au final, $\deg^-(x_1) = 0$.

[1] *ie* qui ne peut être prolongé. À ne pas confondre avec *maximal* qui signifie qu'il n'existe pas de chemin de longueur plus grande.

Propriété 2.1 *Un graphe orienté G est sans circuit si et seulement s'il existe $x \in V$ tel que* $\deg^-(x) = 0$ *et* $G - \{x\}$ *est sans circuit.*

On en déduit le principe d'algorithme suivant :

tant que $\exists\, x \in V$ tel que $\deg^-(x) = 0$, faire $G \leftarrow G - \{x\}$
Si $G = \emptyset$, alors G est sans circuit et sinon G contient un circuit.

L'implémentation se fait en temps (et en espace) $\mathcal{O}(n + m)$ à partir des listes d'adjacence.

Algorithme 1 Algorithme de reconnaissance de graphes orientés sans circuits

Entrée: Un graphe orienté $G = (V, E)$.

```
1:  SommetsRestants ← n
    # Calculer les deg⁻(x)
2:  pour chaque x de V , faire
3:    deg⁻(x) ← 0
4:  pour chaque x de V , faire
5:    pour chaque y de N⁺(x) , faire
6:      deg⁻(y) ← deg⁻(y) + 1
    # Repérer les sommets de degré sortant nul
7:  ATraiter ← ∅
8:  pour chaque x de V , faire
9:    si deg⁻(x) = 0 , alors
10:     ATraiter ← ATraiter ∪ {x}
    # Corps de l'algorithme
11: tant que ATraiter ≠ ∅, , faire
12:   prendre prendre x dans ATraiter
13:   SommetsRestants ← SommetsRestants − 1
14:   pour chaque y de N⁺(x) , faire
15:     deg⁻(y) ← deg⁻(y) − 1
16:     si deg⁻(y) = 0 , alors
17:       ATraiter ← ATraiter ∪ {y}
    # Test sur le nombre de sommets restants
18: si SommetsRestants = 0 , alors
19:   retourner "G est sans circuit"
20: sinon
21:   retourner "G contient un circuit"
```

Structures de données supplémentaires nécessaires pour l'algorithme :
- Un tableau pour stocker les $\deg^-(x)$
- Une liste pour *ATraiter*
- Un entier pour *SommetsRestants*

Définition 2.1 *Un tri topologique est une numérotation des sommets, c'est à dire une bijection* σ *entre* V *et* $\{1, \ldots, n\}$ *telle que* $(x, y) \in E \Rightarrow \sigma(x) < \sigma(y)$.

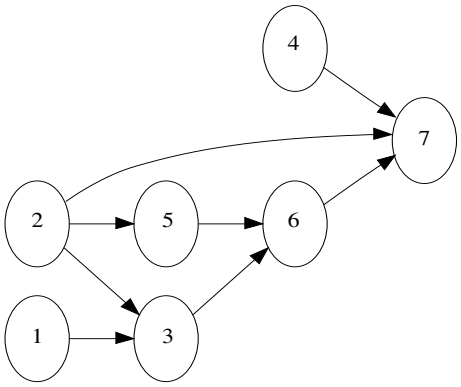

FIG. 4 – Un graphe numéroté selon un tri topologique

Un tri topologique représente un ordre rationnel de traitement des sommets dans un graphe orienté sans circuit.

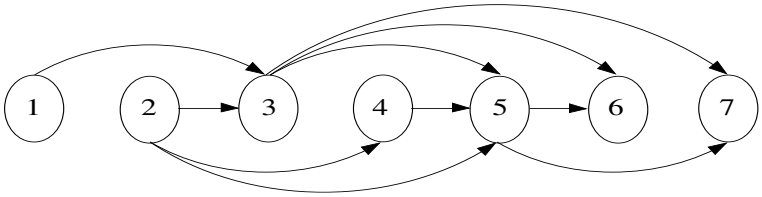

FIG. 5 – Représentation linéaire d'un tri topologique sur un graphe

Il est facile d'adapter l'algorithme de reconnaissance des graphes sans cycle pour qu'il calcule un tri topologique sans surplus de complexité.

Algorithme 2 Distance à un sommet unique s dans un graphe sans cycle

Entrée: Un graphe $G = (V, E)$

Sortie: Un tableau dist où dist(i) sera la distance de s au sommet numéroté par i dans le tri topologique

 # Initialisation

1: calculer un tri topologique σ de G
2: **pour** i **variant de** 1 **à** n , **faire**
3: dist(i) $\leftarrow +\infty$
4: dist($\sigma(s)$) $\leftarrow 0$

 # Corps de l'algorithme

5: **pour** i **variant de** 1 **à** n , **faire**
6: $x \leftarrow \sigma^{-1}(i)$ *# $x \leftarrow$ sommet n°i*
7: **pour chaque** y **de** $N^+(x)$, **faire**
8: **si** dist(y) > dist(x) + $\omega(x, y)$ **, alors**
9: dist(y) $\leftarrow$ dist(x) + $\omega(x, y)$
10: **retourner** dist

2.3 Graphe orienté avec poids réels sans circuit de poids strictement négatif

Algorithme 3 Algorithme de Bellman-Ford

Entrée: Un graphe $G = (V, E)$ et un sommet particulier s
Sortie: Les distances au sommet s

 # Initialisation
pour chaque x **de** V , **faire**
  $\text{dist}(x) \leftarrow +\infty$
$\text{dist}(s) \leftarrow 0$
tant que une distance est modifié par la passe précédente , **faire**
  pour chaque (x, y) **de** E , **faire**
    si $\text{dist}(x) + \omega(x, y) < \text{dist}(y)$ **, alors**
      $\text{dist}(y) \leftarrow \text{dist}(x) + \omega(x, y)$
retourner dist

On fait des passes dans la boucle `while` tant que des valeurs sont modifiés. (On sait qu'au bout d'au plus n passes, l'algorithme s'arêtera).

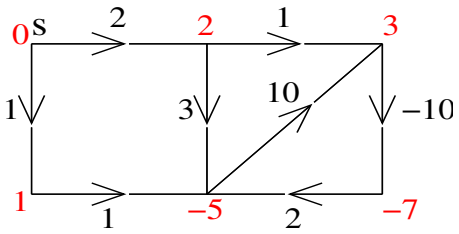

FIG. 6 – Un graphe pour lequel le sommet de poids -5 n'obtient pas sa valeur en deux étapes

Sa compléxité est en $\mathcal{O}(m \cdot n)$.

Correction de l'algorithme :
On montre la correction à l'aide de ces trois invariants :
- (I_1) : $\forall x, \text{dist}(x)$ contient le poids d'un chemin de s à x.
- (I_2) : Un sommet qui a une bonne valeur n'en change plus.
- (I_3) : Si un sommet n'a pas la bonne valeur, après une passe au moins un sommet a reçu sa bonne valeur.

Démonstration :
- (I_1) : À tout instant de l'algorithme et pour tout x de V, $\text{dist}(x)$ est la distance d'un chemin de s à x. Donc on a (I_1).
- (I_2) : $\forall x \in V$, $\text{dist}(x)$ est décroissant. Donc $(I_1) \Rightarrow (I_2)$.
- (I_3) : Si un sommet x n'a pas la bonne valeur, on considère un chemin $\mu = (s = x_1, \ldots, x_k = x)$ optimal pour aller de s à x (c'est posible car G ne contient pas de circuit de poids négatif). On note x_i le premier sommet de μ dont la valeur n'est pas juste ($i \geq 2$ car $x_i = s$). On a alors $\text{dist}(x_i) > \text{dist}(s, x_i)$. Or $\mu' = (s, \ldots, x_i)$ est optimal puisque μ l'est. Lors de la prochaine passe, on aura donc $\text{dist}(x_i) > \text{dist}(s, x_i) = \text{dist}(x_{i-1}) + \omega(x_{i-1}, x_i)$ (par optimalité de μ') et à la fin de cette passe x_i aura la bonne valeur.

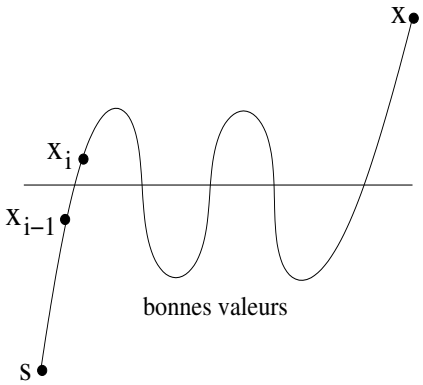

FIG. 7 – démonstration de I3

Cet algorithme est de plus un algorithme stabilisant car on peut corriger une valeur corrompue (seulement dans le bon sens, *ie* si elle a une valeur trop grande) par uniquement des mises à jour locales de chaque noeud.

3 Plus courte distance pour tout couple de sommets

Idée :

On calcule $d_{ij}^{(k)}$, le poids minimum d'un chemin de i à j de longueur k, par programmation dynamique.

$$\begin{cases} \forall k \geq 2,\ d_{ij}^{(k)} = \min_{1 \leq l \leq n}(d_{il}^{(k-1)} + d_{lj}^{(1)}) \\ d_{ij}^{(1)} = \omega(i,j) \end{cases}$$

La complexité de ce calcul est $\mathcal{O}(n^4)$.

Remarque 3.1 *Ce calcul correspond au calcul de* $[d_{ij}^{(k)}]_{i,j} = [d_{ij}^{(k-1)}]_{i,j} * [d_{ij}^{(1)}]_{i,j}$ *qui est une multiplication de matrices dans le semi-anneau*[2] $(\min, +)$.

Avec cette optique, la complexité est alors celle du produit matriciel, en $\mathcal{O}(n^3 \cdot ln(n))$.

Autre idée :

On calcule $d_{ij}^{(k)}$, le poids minimum d'un chemin de i à j dont les sommets intermédiaires sont dans $\{1, \ldots, k\}$, par programmation dynamique :

$$\begin{cases} d_{ij}^{(0)} = \omega(i,j) \\ \forall k \geq 1, d_{ij}^{(k)} = \min(d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, d_{ij}^{(k-1)}) \end{cases}$$

La complexité de cette méthode est $\mathcal{O}(n^3)$.

[2] *ie* un anneau où les éléments n'ont pas nécessairement d'opposé par la loi + qui est ici min