Algorithmique II

Vlady Ravelomanana

IRIF – UMR CNRS 8243 - Paris 7
vlad@liafa.univ-paris-diderot.fr

Master M1BIB
— **Algos de Graphes** —

Programme du cours

- Parcours de graphes
  (en profondeur d'abord – DFS / en largeur – BFS)
- Algorithme de plus court(s) chemin(s)
- Arbres couvrant de poids minimum.

Compétences visées

Avoir les briques de base pour élaborer des algorithmes dans les graphes.

Parcours en profondeur (DFS)

L'algorithme de **parcours en profondeur** ou **DFS** pour *Depth First Search* est un algorithme de parcours d'arbre et de graphe qui s'écrit **récursivement**.

explorer(graphe G, sommet s, liste L)
Entrée : Un graphe G et un sommet s de ce graphe.
Sortie : Une liste L des sommets de ce graphe.
$L :=$ concaténer(L, s)
marquer s
pour tout sommet t voisin de s **faire**
    si t n'est pas marqué **alors**
        $L :=$ concaténer(L, t)
        marquer t
        explorer(G, t, L)
    fin si
fin pour

Applications

L'algo. trouve l'ensemble de sommets accessibles depuis le sommet s : on peut l'utiliser pour calculer les **composantes connexes**, pour tester s'il existe un chemin entre s et t

Parcours en largeur (BFS)

Le principe consiste à visiter d'abord à partir d'une source s ses plus proches voisins (distance 1 de la source). Puis d'explorer ces voisins un par un. On utilise une **file**.

explorer(graphe G, sommet s)
Entrée : Un graphe G et un sommet s de ce graphe.
Sortie : Une liste L des sommets de ce graphe.
f := CréerFile() ; f.enfiler(s) ; marquer(s)
tant que f est non vide **faire**
    $s = f$.défiler();
    résultat = concaténer(résultat, s)
    pour tout voisin t de s dans G **faire**
        si t non marqué **alors**
            f.enfiler(t) ; marquer(t)
        fin si
    fin pour
fin tant que
retourner résultat

Applications

L'algorithme de parcours en largeur permet de calculer les distances de tous les sommets depuis un sommet source dans un graphe non pondéré. Il peut aussi servir à déterminer si un graphe est connexe.

L'algo de Dijkstra (1959)

Cet algorithme sert à trouver le **plus court chemin** d'un sommet s à un sommet t mais il est **plus général** car on peut demander à avoir la liste des plus courts chemins de s à tous les sommets du graphe. A noter que le graphe peut être **orienté**.

Principe :

- En entrée, nous avons un graphe orienté pondéré par des réels positifs et un sommet source.
- On construit progressivement un **sous-graphe** dans lequel sont classés les différents sommets par ordre croissant de **distance** par rapport à s. Cette distance est la **somme** des poids des arcs empruntés.
- Au début, les distances sont toutes infinies sauf pour s qui est à distance 0. Le sous-graphe est vide.
- A chaque itération, on choisit un sommet **de distance minimale n'appartenant pas encore au sous-graphe**. On le rajoute au sous-graphe. On met alors à jour les distances des sommets voisins de celui qui vient d'être rajouté : c'est le minimum entre la distance d'avant et celle obtenue en rajoutant le poids de l'arc entre le sommet voisin et le sommet ajouté.

Pseudo-code Dijkstra

Dijkstra(graphe $G = (V, E)$, sommet s)
Entrée : Un graphe G et un sommet s de ce graphe.
Sortie : La liste des distances de tous les sommets de G à partir de s.

```
dist[s] = 0
pour tout sommet v dans V − {s} faire
    dist[s] = +∞
fin pour
S := ∅ ; Q := V (⋆ S est l'ensemble des sommets déjà visité(s) ⋆)
tant que Q ≠ ∅ faire
    u := mindistance(Q, dist) (⋆ u est l'élément de Q de plus petite distance ⋆)
    S := S ⋃ {u}
    pour tout voisin v de u faire
        si dist[v] > dist[u] + poids(u, v) alors
            dist[v] = dist[u] + poids(u, v)
        fin si
    fin pour
fin tant que
retourner dist
```

Complexité et applications de DIJKSTRA

En considérant Q (voir algo.) comme une **file de priorité** implémenter en **tas**, la complexité de l'algorithme est en $O((|A| + |S|) \times \log(|S|))$.

Applications

- Calcul d'intinéraires :
  arcs = distance, temps, coût, consommation carburant, prix, ...
- protocoles de routages (recherche de parcours efficaces, ...)

Remarque(s) :

- L'algorithme de DIJKSTRA est basé sur un parcours en largeur.
- L'algorithme ne s'applique pas aux graphes avec des arcs de **poids négatifs**.

L'algo de Floyd-Warshall (1959)

Décrit par Bernard Roy en 1959 :) c'est un algorithme qui determine les distances des plus courts chemins entre toutes les paires de sommets dans un **graphe orienté et pondéré** **avec des poids négatifs mais sans que le graphe ne possède de circuit de poids strictement négatif.** La complexité de l'algorithme est en temps $O(n^3)$ pour n sommets.

Principe :

- On note $W_{i,j}^k$ le poids minimal d'un **chemin du sommet i au sommet j** n'empruntant que des sommets intermédiaires dans $\{1, 2, 3, \cdots, k\}$ s'il en existe sinon c'est $+\infty$.
- On note W^k la matrice $W^k = [W_{i,j}^k]$, W^0 étant la **matrice d'adjacence** du graphe de départ.
- Si p est un chemin entre i et j de poids minimal dont les sommets intermédiaires sont dans $\{1, \cdots, k\}$ alors :
  1. soit p n'emprunte pas k
  2. soit p emprunte k exactement une fois et donc p est la concaténation de deux chemins de i à k puis de k à j avec des sommets intermédiaires dans $\{1, 2, \cdots, k-1\}$ d'où la récurrence :

$$W_{i,j}^k = \min(W_{i,j}^{k-1}, W_{i,k}^{k-1} + W_{k,j}^{k-1})$$

Pseudo-code de Floyd-Warshall

FloydWarshall(graphe G)
Entrée : (graphe G donné par sa matrice d'adjacence M)
Sortie : W^n la matrice des distances entre tous les sommets.
W^0 := matrice d'adjacence de G
pour k de 1 à n **faire**
    pour i de 1 à n **faire**
        pour j de 1 à n **faire**
            $W^k_{i,j} := \min(W^{k-1}_{i,j},\ W^{k-1}_{i,k} + W^{k-1}_{k,j})$
        fin pour
    fin pour
fin pour
retourner W^n

Complexité et applications de Floyd-Warshall

On lit directement sa complexité en temps : $O(n^3)$. On peut optimiser l'espace en ne mémorisant qu'une matrice $n \times n$ (i.e. W^k est remplacé par W) : complexité en espace en $O(n^2)$.

Applications

- Plus courts chemins notamment dans les graphes orientés non pondérés (poids unitaire pour chaque arc
- Calcul de la fermeture transitive d'un graphe orienté (on rajoute un arc entre x et y ssi il existe un chemin orienté de x à y) : en remplaçant

$$W_{i,j} := \min(W_{i,j},\ W_{i,k} + W_{k,j})$$

par

$$C[i,j] = C[i,j] \text{ OR } (C[i,k] \text{ AND } C[k,j])\,.$$

Remarque(s) :

Attention : Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de circuit de poids strictement négatif !

Algo de Bellman-Ford

Découvert en 1956 (Ford) redécouvert en 1958 (Bellman) puis en 1959 (Moore), cet algorithme calcule les plus courts chemins depuis un sommet source donné dans un graphe orienté pondéré. Il a la **particularité** d'autoriser la présence d'arcs de poids négatif et permet de détecter l'existence d'un **circuit absorbant**, c'est-à-dire de poids total strictement négatif, accessible depuis le sommet source.

Principe :

- On note $d[t,\, k]$ la distance du sommet source s à t avec un chemin d'au plus k arcs.
- On a donc $d[t,\, 0] = +\infty$ pour $t \neq s$ et $d[s,\, 0] = 0$
- Puis

$$d[t, k] = \min(d[t, k-1], \min_{\text{arc}(u,t)} (d[u, k-1] + \text{poids}(u, t)))$$

Pseudo-code de Bellman-Ford

BellmanFord(graphe G, sommet s)
Entrée : (graphe G et sommet de départ s)
Sortie : distance de s à tous les sommets de G.
pour chaque sommet u de G **faire**
    $d[u] := +\infty$
fin pour
$d[s] := 0$
pour k de 1 à $n-1$ **faire**
    pour chaque arc $(u,\ t)$ de G **faire**
        $d[t] := \min(d[t],\ d[u] + \text{poids}(u,\ t))$
    fin pour
fin pour
retourner d

Comme il y a un cycle de poids négatif ssi un nouveau tour de boucle fera diminuer la distance alors la détection d'un cycle de poids négatif se fait à la fin de l'algo comme suit :

pour chaque arc $(u,\ t)$ de G **faire**
    si $d[u] + \text{poids}(u,\ t) < d[t]$ **alors**
        retourner VRAI
    fin si
fin pour
retourner FAUX