Graphes #3 : Plus-court-chemins

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Recherche de plus-court-chemins non pondérés

Objectif

Soit :

- $G = (S, A)$ un graphe (éventuellement orienté)
- v un sommet de G

Pour tout $w \neq v$ accessible depuis v, on veut trouver **un** plus-court-chemin non pondéré depuis v (un chemin $v = s_0 \to s_1 \to \cdots \to s_n = w$ minimisant n parmi tous les chemins $v \rightsquigarrow w$)

- Sans perte de généralité, le chemin recherché est simple, élémentaire, ouvert

Résolution par BFS

Un parcours en largeur d'origine v visite les sommets accessibles depuis v par distance non pondérée croissante, et les chemins reconstruits post-parcours sont des plus-court-chemins

- Par définition d'un parcours en largeur

BFS : implémentation typique (rappel)

- On maintient (par ex. dans une file ou deux piles) un ensemble des sommets *non encore visités* pour lesquels il existe un chemin de longueur i ou $i+1$ depuis la source
- On visite (au besoin) les sommets quand on les extrait de l'ensemble
- Un sommet extrait à i est à distance au plus i
- La première fois qu'un sommet est inséré, il l'est à sa distance minimale (par récurrence sur i)

Bilan

Un problème simple

- Peut se résoudre par un algorithme « de base », en temps linéaire en la taille d'une représentation par tableau de liste d'adjacence
- Peut aussi se résoudre par un algorithme plus compliqué et moins efficace (*cf. infra*)
- Dans le cas d'une représentation par matrice d'adjacence, peut aussi se résoudre par un algorithme moins efficace mais pas forcément beaucoup plus compliqué (*cf. infra2*)

Réfléchissez bien à quel algorithme utiliser pour résoudre un tel problème !

Table des matières

Recherche de plus-court-chemins pondérés

Objectif

Soit :

- $G = (S, A, P)$ un graphe orienté de pondération d'arcs P
- v un sommet de G

On veut trouver un plus-court-chemin pondéré depuis v vers tout sommet w accessible depuis v (un chemin $v = s_0 \to s_1 \to \cdots \to s_n = w$ minimisant $\sum_{i=0}^{n-1} P(s_i, s_{i+1})$ parmi tous les chemins $v \rightsquigarrow w$)

Remarque

La pondération n'est pas forcément positive (même si on le supposera pour l'algorithme que l'on s'apprête à décrire), mais le problème devient mal posé si le graphe possède un cycle de poids strictement négatif

- Dans ce dernier cas, on souhaite généralement détecter la présence de tels cycles

Algorithme « de Dijkstra »

Algorithme « de Dijkstra » : parcours général avec une file de priorité *min*

Résout le SSSP pondéré pour des graphes « représentés par tableau de listes d'adjacence », à pondération positive (et dans une certaine variante, à pondération générale en l'absence de cycles de pondération négative 🙈)

Principe

- On maintient une file de priorité qui stocke des sommets non encore visités avec un **majorant** de leur (plus-courte) distance à l'origine
  - Ou plutôt des couples *(distance, arcs)* où la distance est la longueur du chemin à l'origine obtenu en branchant cet arc à l'arborescence du parcours
- Tant que cette file n'est pas vide, on en extrait un sommet de majorant le plus petit, on le visite si pas encore vu, et l'on insère (éventuellement, éventuellement à nouveau) ses voisins dont on connaît maintenant un (nouveau) majorant de la distance
- On va montrer que lorsqu'un sommet extrait n'est pas encore visité, le majorant de la distance est la vraie (plus-courte) distance

Exemple d'implémentation en OCaml

```
type graph = (int * int) list array (* voisin, poids de l'arc *)
let sssp g v =
    let n = Array.length g in
    let dist = Array.make n (-1, -1) in (* infini, pas de parent *)
    let pq = MinPq.create () in
    let push_cond d p (w, dpw) =
        if fst dist.(w) = -1 then MinPq.add pq (d + dpw, p, w) in
    let rec loop
        = function
        | None -> dist
        | Some (d, p, w) -> if fst dist.(w) = -1 then begin
                                dist.(w) <- d, p ;
                                List.iter (push_cond d w) g.(w)
                            end ;
                            loop (MinPq.pop_min pq) in
    loop (Some (0, -1, v))
```

Exemple

Correction

Lemme

Les sommets sont visités par distance à v croissante, avec une distance égale à celle d'un plus-court-chemin vers v

Et puisque `sssp` est un parcours depuis v :

Corollaire

L'arborescence du parcours donne des plus-court-chemins depuis v vers tous les sommets qui lui sont accessibles

Preuve du lemme (esquisse)

Esquisse de preuve

Par récurrence sur le nombre de sommets déjà visités

- Cas de base : vrai pour le premier sommet visité (*viz.* v, à distance 0 de lui-même, minimal en l'absence de poids négatifs)
- Cas récursif : quand on visite un sommet w depuis un arc (p, w) avec priorité d, tout autre chemin $v \rightsquigarrow w$ (différent de $v \rightsquigarrow p \to w$) ou $v \rightsquigarrow w'$ vers w' pas encore visité doit passer par un arc encore dans la file de priorité, donc moins prioritaire, donc induisant un chemin plus long (avec égalité possible) ; les « différents » cas sont :
  - $v \rightsquigarrow p' \to x$
  - $v \rightsquigarrow p' \to p'' \rightsquigarrow x$

  avec p' (éventuellement égal à v) déjà visité, et (p', x) ou (p', p'') des arcs présents dans la file de priorité avec une priorité d'. Dans les deux cas, la longueur du chemin est minorée par $d' \geq d$.

  Remarque : cela serait faux en présence d'arcs de poids négatif

Analyse de coût

On note S_v, A_v les sommets, arcs accessibles dans un parcours depuis v

- Chaque arc $\in A_v$ est inséré au plus une fois dans la file de priorité
- Et donc extrait au plus une fois
- On effectue donc au plus $\#A_v$ opérations sur une file de priorité d'au plus $\#A_v$ éléments
  - Coût total $O(\#A_v \log \#A_v) = O(\#A_v \log \#S_v)$ pour une file efficace (par exemple implémentée avec un tas binaire)
- On effectue par ailleurs $O(\#S)$ opérations (optimisable en $O(\#S_v)$)
- Ce qui donne un coût total en $O(\#A_v \log \#S_v + \#S)$

Dans le cas (habituel) où $\#A = \Omega(\#S)$, le coût de l'algorithme « de Dijkstra » implémenté avec une file de priorité efficace est un $O(\#A \log \#S)$ (améliorable)

Table des matières

Plus-court-chemins pondérés entre toute paire de sommets (APSP)

Objectif

Soit $G = (S, A)$ un graphe orienté à arc pondérés, on veut trouver des plus-court-chemins pondérés entre *chaque paire de sommets* v, w de G

Résolution

- Itérer un algorithme de résolution du SSSP
  - Pour l'algorithme « de Dijkstra » tel qu'implémenté ci-dessus, donne un coût $O(\#S\#A \log \#S)$
- Algorithme dédié ?

Algorithme « de Floyd-Warshall »

Algorithme « de Floyd-Warshall »

Résout le APSP pour des graphes « représentés par matrice d'adjacence »

- Gère correctement la présence d'arcs de poids négatif en l'absence de cycle de poids strictement négatif
- Permet de détecter la présence de tels cycles (*cf.* TD)

Principe

On construit itérativement une matrice des longueurs des plus-court-chemins entre toutes paires de sommets **pour des chemins qui ne peuvent que passer par un certain sous-ensemble** (bien choisi, qui croît strictement à chaque itération) **des sommets**

Ce n'est pas un algorithme de parcours de graphe

FW *bis* : un invariant

Un peu plus de détails

On définit une matrice des distances $\mathbf{D}$ telle qu'après $\#S$ itérations d'une boucle à définir, $\mathbf{D}_{v,w}$ contient la longueur d'un plus-court-chemin $v \rightsquigarrow w$

- On va écrire la boucle indexée par k de sorte qu'elle satisfasse l'invariant de boucle que :

  > $\mathbf{D}_{v,w}$ est la longueur d'un plus-court-chemin $v \to [\![k]\!] \to w$ **dont les éventuels sommets intermédiaires sont tous** $< k$

- Cet invariant est satisfait avant entrée dans la boucle ($k = 0$) pour $\mathbf{D}$ initialisée à la matrice d'adjacence du graphe : tout sommet v est à distance 0 de lui-même et $\mathbf{D}_{v,w}$ de w pour les chemins sans aucun sommet intermédiaire

La correction totale est une conséquence immédiate de l'invariant en fin de la dernière itération ($k = \#S$)

FW *ter* : maintenir l'invariant

On note $\mathbf{D}^k$, (resp. $\mathbf{D}^{k+1}$) la matrice des distances au début (resp. à la fin) de la $k^{\text{ième}}$ itération

- Il faut possiblement changer toutes les entrées de $\mathbf{D}^{k+1}$ (par rapport à $\mathbf{D}^k$) pour préserver l'invariant
- Pour chaque v, w, Deux cas possibles :
  - Un plus-court-chemin $v \to [\![k]\!] \to w$ est aussi un plus-court-chemin $v \to [\![k+1]\!] \to w$, dans ce cas $\mathbf{D}^{k+1}_{v,w} = \mathbf{D}^k_{v,w}$ par l'invariant
  - Il existe un chemin $v \to [\![k+1]\!] \to w$ plus court ; *en l'absence de cycles de poids* < 0, celui-ci est nécessairement élémentaire, et donc de la forme $v \to [\![k]\!] \to k \to [\![k]\!] \to w$, et par l'invariant de longueur $\mathbf{D}^k_{v,k} + \mathbf{D}^k_{k,w}$
- On peut donc préserver l'invariant si pour chaque v, w l'on prend :

$$\mathbf{D}^{k+1}_{v,w} = \min \mathbf{D}^k_{v,w}, (\mathbf{D}^k_{v,k} + \mathbf{D}^{k,w})$$

FW *quater* : remarques

- Aucune attention particulière n'est nécessaire pour gérer les arcs de poids négatif en l'absence de cycles de poids strictement négatif ; dans ce dernier cas, on peut détecter leur présence en inspectant le résultat calculé (ou même en cours d'exécution), *cf.* TD
- On n'a pour l'instant que calculé les distances, mais on peut facilement reconstruire des plus-court-chemins si à chaque mise à jour de **D** l'on mémorise l'arc utilisé, *cf.* TP
- La mise à jour de **D** peut se faire *en place*, *cf.* TP
- Le même algorithme peut être utilisé pour résoudre d'autres problèmes en changeant le semi-anneau sous-jacent (ici $(\min, +)$), *cf.* TP
- Cet algorithme peut-être vu comme un exemple d'algorithme de programmation dynamique, *cf.* bientôt

Exemple d'implémentation en C

Pour un graphe représenté par matrice d'adjacence pondérée ; on exploite ici le fait que la mise-à-jour de d est faisable en place (pas *a priori* évident)

```
double **apspfw(size_t n, double **a) {
    double **d = copy_a(n, a);
    for (size_t k = 0; k < n; k++) { // K.I.J.
        for (size_t i = 0; i < n; i++) {
            for (size_t j = 0; j < n; j++) {
                double npl = d[i][k] + d[k][j];
                d[i][j] = npl < d[i][j] ? npl : d[i][j];
            }
        }
    }
    return d;
}
```

Analyse de coût

Le coût de l'algorithme « de Floyd-Warshall » est un $O(\#S^3)$ évident