Application de la programmation dynamique aux plus courts chemins

Quentin Fortier

December 1, 2022

Plus courts chemins

Dans ce cours, on considère seulement des graphes orientés.

Définition

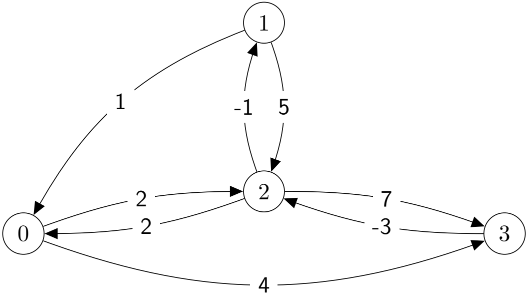
Un graphe **pondéré** est un graphe $\overrightarrow{G} = (V, \overrightarrow{E})$ muni d'une fonction de poids $w : \overrightarrow{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

Il est pratique de définir $w(u, v) = \infty$ s'il n'y a pas d'arête entre u et v.
En Python, on peut utiliser `float("inf")` pour représenter $+\infty$.

Pour représenter un graphe pondéré, on utilisera ici une **matrice d'adjacence pondéré**, contenant $w(u, v)$ sur la ligne u, colonne v.

Plus courts chemins

Exemple de graphe représenté par matrice d'adjacence pondérée :



$$\begin{pmatrix} 0 & \infty & 2 & 4 \\ 1 & 0 & 5 & \infty \\ 2 & -1 & 0 & 7 \\ \infty & \infty & -3 & 0 \end{pmatrix}$$

```python
G = [
    [0, float("inf"), 2, 4],
    [1, 0, 5, float("inf")],
    [2, -1, 0, 7],
    [float("inf"), float("inf"), -3, 0]
]
```

Plus courts chemins

Vous avez déjà vu plusieurs algorithmes pour trouver des plus courts chemins :

- **Parcours en largeur** (BFS), si tous les poids sont égaux (distance = nombre d'arêtes).
- **Dijkstra**, si tous les poids sont positifs.

Plus courts chemins

Nous allons voir 3 algorithmes supplémentaires de plus courts chemins, par programmation dynamique ($n = |V|$, $p = |\vec{E}|$) :

Algorithme	**Condition**	**Complexité**
Parcours en largeur	Tous les poids égaux (distance $=$ nombre d'arêtes)	O$(n + p)$
Dijkstra	Poids positifs	O$(p \log(n))$
Bellman-Ford		O(np)
Floyd-Warshall		O(n^3)
Prog. dyn. sur graphe acyclique	Graphe acyclique	O$(n + p)$

Floyd-Warshall trouve toutes les distances entre deux sommets quelconques, contrairement aux autres algorithmes (qui calculent les distances depuis un sommet de départ).

Lemme

Soit $\vec{G} = (V, \vec{E})$ un graphe orienté et $u, v \in V$. Alors :

$$d(u, v) = \min_{(w,v) \in \vec{E}} d(u, w) + w(w, u)$$

Preuve :
Soit C un plus court chemin de u à v et w le prédecesseur de v dans C.

Soit C' la partie de C allant de u à w.
Alors C' est un plus court chemin de u à w (sinon il existerait un chemin C'' plus court et $C'' + (w, v)$ serait plus court que C : absurde).
Donc $d(u, v) = d(u, w) + w(w, v)$.

Lemme

Soit $\vec{E} = (V, \vec{E})$ un graphe orienté et $u, v \in V$. Alors :

$$d(u,v) = \min_{(w,v)\in\vec{E}} d(u,w) + w(w,u)$$

<u>Problème</u> : on ne se ramène pas à des sous-problèmes plus petits...

Il faut introduire un paramètre pour avoir une équation de récurrence exploitable :

- **Bellman-Ford** : nombre d'arêtes
- **Floyd-Warshall** : numéros des sommets que l'on peut utiliser

Ou ajouter une condition sur le graphe :

- **Graphe acyclique** : un tri topologique donne un ordre dans lequel traiter les sommets.

Algorithme de Bellman-Ford

Lemme

Soit $d_k(v)$ le poids minimum d'un chemin de r à v **utilisant au plus** k **arêtes**. Alors :

$$d_{k+1}(v) = \min_{(u,v) \in E} d_k(u) + w(u, v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k+1$ arêtes.
Soit u le prédecesseur de v dans C.
Alors le sous-chemin de C de r à u est un plus court chemin utilisant au plus k arêtes (s'il y avait un chemin plus court que C', on pourrait le remplacer dans C ce qui contredirait la minimalité de C).

Remarque : c'est une propriété de **sous-structure optimale** (un sous-chemin d'un plus court chemin est aussi un plus court chemin).

Algorithme de Bellman-Ford

Parcourir tous les sommets puis tous les arcs (`u`, `v`) entrants dans `v` revient à parcourir tous les arcs du graphe :

Algorithme de Bellman-Ford

Entrée : $\overrightarrow{G} = (V, \overrightarrow{E})$ pondéré par w et $s \in V$.
Sortie : d tel que $d[v]$ soit la distance de s à v.

Pour $u \in V$ **Faire**
  Pour $k \in [\![0, n-2]\!]$ **Faire**
    Si $v = s$ **Alors** $d[s][k] = 0$
    Sinon $d[v][k] = \infty$

Pour $k \in [\![0, n-2]\!]$ **Faire**
  Pour $(u, v) \in \overrightarrow{E}$ **Faire**
    $d[v][k+1] = \texttt{min}(d[v][k+1], d[u][k] + w(u, v))$

Algorithme de Bellman-Ford

Comme on a juste besoin de stocker $d[...][k-1]$ pour calculer $d[...][k]$:

Algorithme de Bellman-Ford

Entrée : $\vec{G} = (V, \vec{E})$ pondéré par w et $s \in V$.
Sortie : d tel que $d[v]$ soit la distance de s à v.

$d[s] = 0$
Pour $v \in V - s$ **Faire**
  $d[v] = \infty$

Pour $k \in [\![0, n-2]\!]$ **Faire**
  Pour $(u, v) \in \vec{E}$ **Faire**
    $d[v] = \texttt{min}(d[v], d[u] + w(u, v))$

Algorithme de Bellman-Ford

Algorithme de Bellman-Ford

python

```python
def bellman(g, r):
    n = len(g)
    d = [float("inf")]*n
    d[r] = 0
    for k in range(n - 2):
        for u in range(n):
            for v in range(len(g[u])):
                d[v] = min(d[v], d[u] + g[u][v])
    return d
```

Algorithme de Floyd-Warshall

Initialiser $d_0(u, v) \leftarrow g[u][v]$ si $(u, v) \in \overrightarrow{E}$.
∞ sinon.

Pour `k` $= 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :
            $d_{k+1}(u, v) \leftarrow \texttt{min}(d_k(u, v), d_k(u, k) + d_k(k, v))$

On peut utiliser un tableau `d` à 3 dimensions pour stocker $d_k(u, v)$ dans `d[u][v][k]`.

On a en fait juste besoin de d_k pour calculer d_{k+1} : on peut donc utiliser une matrice `d` telle que `d[u][v]` contient le dernier $d_k(u, v)$ calculé (ça marche car $d_{k+1}(u, k) = d_k(u, k)$).

Algorithme de Floyd-Warshall

On utilise une matrice `d` telle que `d[u][v]` contienne le dernier $d_k(u, v)$ calculé :

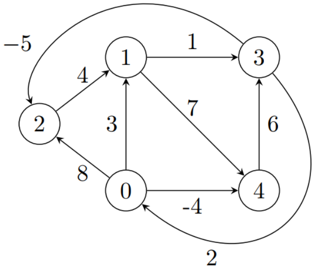
Algorithme de Floyd-Warshall

```
d = copie de g

Pour k = 0 à n − 1 :
    Pour tout sommet u :
        Pour tout sommet v :
            d[u][v] = min(d[u][v], d[u][k] + d[k][v])
```

Complexité : $\boxed{O(n^3)}$

Algorithme de Floyd-Warshall



$$A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Matrice d'adjacence

$$\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Matrice des distances renvoyée par Floyd-Warshall

Algorithme de Floyd-Warshall

On suppose que le graphe g est représenté par matrice d'adjacence pondérée :

Algorithme de Floyd-Warshall

python

```python
import copy

def floyd_warshall(g):
    n = len(g)
    d = copy.deepcopy(g) # pour éviter de modifier g
    for k in range(n):
        for u in range(n):
            for v in range(n):
                d[u][v] = min(d[u][v], d[u][k] + d[k][v])
    return d
```

Algorithme de Floyd-Warshall

Initialiser `d[u][v]` $\leftarrow g[u][v]$ si $(u, v) \in \vec{E}$, ∞ sinon.

Pour `k` $= 0$ à $n - 1$:
Pour tout sommet u :
Pour tout sommet v :

```
d[u][v] = min(d[u][v], d[u][k] + d[k][v])
```

Question

Comment connaître un plus court chemin de n'importe quel sommet u à n'importe quel un autre v?

Utiliser une matrice `pere` telle que `pere[u][v]` est le prédécesseur de `v` dans un plus court chemin de `u` à `v`.

Algorithme de Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas).
Soit $p_k(u, v)$ un prédécesseur de v dans un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$.
Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

- Si C n'utilise pas k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, v)$$

$$p_{k+1}(u, v) = p_k(u, v)$$

- Si C utilise k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, k) + d_k(k, v)$$

$$p_{k+1}(u, v) = p_k(k, v)$$

Algorithme de Floyd-Warshall

Initialiser `d[u][v] = w(u, v)` si (u, v) $\in \vec{E}$, ∞ sinon.
Initialiser `pere[u][v] = u`, $\forall$ u, v $\in V$.

Pour `k` $= 0$ à $n-1$:
    Pour tout sommet u :
        Pour tout sommet v :
            Si `d[u][v] > d[u][k] + d[k][v]` :
                `d[u][v] = d[u][k] + d[k][v]`
                `pere[u][v] = pere[k][v]`

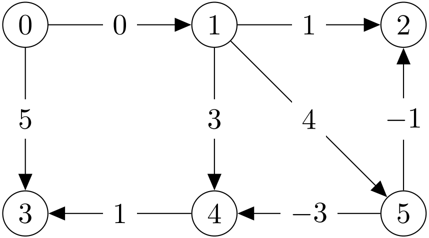
On obtient une matrice `pere` telle que `pere[u][v]` est le prédécesseur de `v` dans un plus court chemin de `u` à `v`.

Graphe acyclique

Définition

On appelle **graphe acyclique** un graphe qui ne contient pas de cycle.

Exemple de graphe acyclique :



Graphe acyclique : Tri topologique

Définition

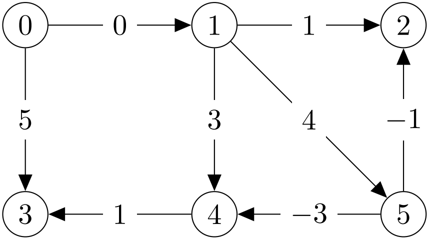
Un **tri topologique** d'un graphe $\vec{G}$ est un ordre des sommets de $\vec{G}$ telle que si $(u, v) \in \vec{E}$, alors u est avant v dans l'ordre.

Lemme

G est acyclique $\iff$ il existe un tri topologique de G

Question

Trouver un tri topologique du graphe ci-dessous.

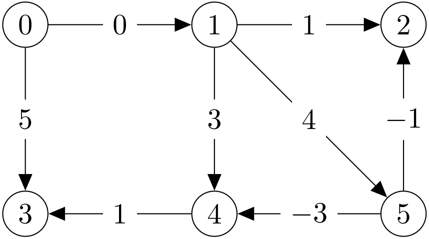


Graphe acyclique : Tri topologique

Pour l'implémentation, on utilisera une liste d'adjacence où un arc $u \xrightarrow{w} v$ est représenté par un tuple (v, w).

Question

Donner la représentation du graphe ci-dessous.



Graphe acyclique : Tri topologique

Théorème (admis)

Soit $\vec{G}$ un graphe acyclique.
La liste des sommets de $\vec{G}$ dans un parcours en profondeur postfixe (obtenue par `dfs_postfixe` ci-dessous) est un tri topologique de G.

Tri topologique

python

```python
def dfs_postfixe(G, s):
    seen = [False]*len(G)
    L = []
    def aux(G, v):
        if not seen[v]:
            seen[v] = True
            for w in G[v]:
                aux(G, w)
            L.append(v)
    aux(G, s)
    return L[::-1] # inverse la liste
```

Graphe acyclique : Tri topologique

Soit $\overrightarrow{G} = (V, \overrightarrow{E})$ un graphe orienté acyclique et $s \in V$.

Il est possible de trouver les distances depuis s dans G en complexité linéaire :

- Trouver un tri topologique de $\overrightarrow{G}$.
- Pour chaque sommet v dans l'ordre topologique, calculer $d(s, v)$ par récurrence (programmation dynamique).

Graphe acyclique : Programmation dynamique

On utilise alors la formule de récurrence démontrée précédement :

Théorème

$$d(s, v) = \mathtt{min}_{(u,v)\in \vec{E}} d(s, u) + w(u, v)$$

Graphe acyclique : Programmation dynamique

Distances dans un graphe acyclique $\vec{G}$

Calculer un tri topologique de $\vec{G}$.
Pour chaque sommet v dans l'ordre topologique :

$$d[v] = \mathtt{min}_{(u,v)\in\vec{E}} d[v] + w(u,v)$$

Cependant, il est plus pratique de mettre à jour les successeurs plutôt que de considérer les prédécesseurs. D'où l'algorithme équivalent suivant :

Distances dans un graphe acyclique $\vec{G}$

Calculer un tri topologique de $\vec{G}$.
Pour chaque sommet u dans l'ordre topologique :
Pour chaque arc $(u,v) \in \vec{E}$:

$$d[v] = \mathtt{min}(d[v], d[u] + w(u,v))$$

Graphe acyclique : Programmation dynamique

Distances dans un graphe acyclique

python

```python
def distances_acyclic(G, s):
    n = len(G)
    d = [float('inf')]*n
    d[s] = 0
    for u in dfs_postfixe(G, s):
        for v in G[u]:
            d[v] = min(d[v], d[u] + w(u, v))
    return d
```