Plus courts chemins dans les graphes pondérés

Quentin Fortier

Algorithme de Dijkstra

On considère dans ce cours seulement des graphes orientés.

Définition

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

Pour représenter un graphe pondéré, on utilisera ici une liste d'adjacence avec une fonction de poids w.

Algorithme de Dijkstra

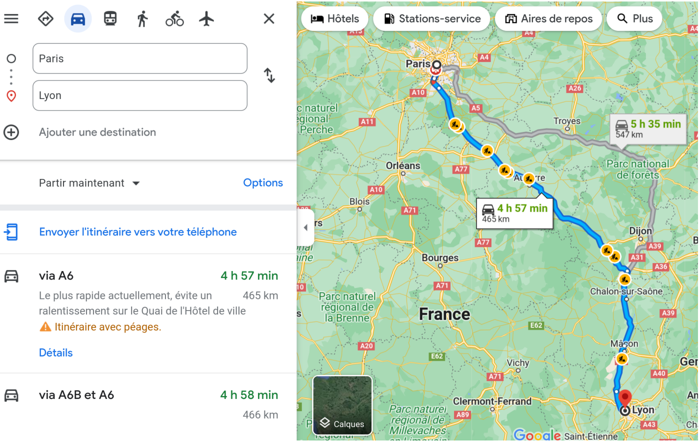
L'algorithme de Dijkstra permet de résoudre le problème suivant :

Problème

Entrée : $\vec{G} = (V, \vec{E})$ un graphe orienté pondéré par des **poids positifs** et $s \in V$.
Sortie : Un tableau contenant $d(s, v)$, pour tout $v \in V$.

Algorithme de Dijkstra

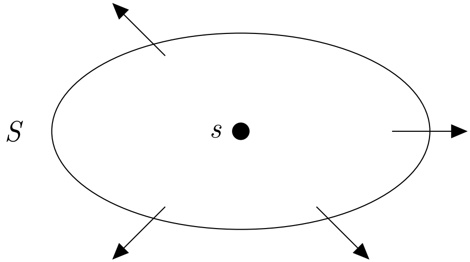
Exemple d'application : trouver le plus court chemin d'une ville à une autre.



Algorithme de Dijkstra

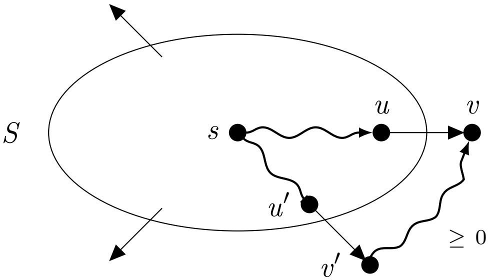
Idée : Calculer les distances par ordre croissant depuis s.

Soit $S \subset V$ l'ensemble des sommets de distance connue.



À chaque étape, on déduit la distance à un sommet de plus (et on l'ajoute à S).

Algorithme de Dijkstra



Soit $(u, v) \in \vec{E}$ tel que $v \notin S$ et $d(s, u) + w(u, v)$ est minimum.
Alors :

$$\boxed{d(s, v) = d(s, u) + w(u, v)}$$

<u>Preuve</u> :

2. Un chemin C de s à v doit sortir de S avec un arc (u', v'). Comme les poids sont ≥ 0 :

$$\text{poids}(C) \geq d(s, u') + w(u', v') \geq d(s, u) + w(u, v)$$

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans q $(= \overline{S})$ et on conserve un tableau `dist` des distances estimées tel que :

1. $\forall v \notin$ `q` : `dist.(v)` $= d(s, v)$.
2. $\forall v \in$ `q` : `dist.(v)` $= \min\limits_{u \notin q} d(s, u) + w(u, v)$.

<u>Algorithme de Dijkstra</u> :

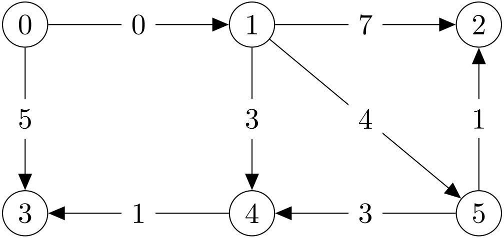
```
Initialement : q contient tous les sommets
               dist.(s) <- 0
               dist.(v) <- float("inf"), si v <> s

Tant que q est non vide:
    Extraire u de q tel que dist.(u) soit minimum
    Pour tout voisin v de u:
        Si dist.(u) + w(u, v) < dist.(v):
            dist.(v) <- dist.(u) + w(u, v)
```

Algorithme de Dijkstra : Exemple

Exercice

Appliquer l'algorithme de Dijkstra depuis $s = 0$ sur le graphe suivant, en mettant `dist.(v)` à côté de chaque sommet `v` :



Algorithme de Dijkstra : Avec une file de priorité

Initialement : `next` contient tous les sommets
`dist.(r) <- 0` et `dist.(v) <- ∞`, $\forall$ `v` $\neq$ `r`

Tant que `next` $\neq \emptyset$:
  Extraire `u` de `next` tel que `dist.(u)` soit minimum
  Pour tout voisin `v` de `u` :
    `dist.(v) <- min dist.(v) (dist.(u) + w u v)`

Complexité si `next` est une **file de priorité min** :

1. n extractions du minimum $\longrightarrow \mathrm{O}(n\log(n))$
2. au plus p mises à jour $\longrightarrow \mathrm{O}(p\log(n))$

Total : $\mathrm{O}(n\log(n)) + \mathrm{O}(p\log(n)) = \boxed{\mathrm{O}(p\log(n))}$.

Algorithme de Dijkstra : Avec une file de priorité

On suppose avoir une file de priorité min avec les fonctions suivantes :

```
(* make () renvoie une file de priorité vide *)
make : unit -> 'a priority_queue

(* add q e p ajout e avec priorité p dans q *)
add : 'a priority_queue -> 'a -> int -> unit

(* empty q détermine si q est vide *)
empty : 'a priority_queue -> bool

(* extract_min q extrait l'élément de q de priorité minimum *)
extract_min : 'a priority_queue -> 'a

(* update q e p met à jour la priorité de e dans q *)
update : 'a priority_queue -> 'a
```

Algorithme de Dijkstra : Avec une file de priorité

```
let dijkstra g w s =
    let n = Array.length g in
    let dist = Array.make n max_int in
    dist.(s) <- 0;
    let q = make () in
    for v = 0 to n - 1 do
        add q v dist.(v)
    done;
    while not empty q do
        let u = extract_min q in
        List.iter (fun v ->
            let d = dist.(u) + w u v in
            if d < dist.(v) then (
                update q v d;
                dist.(v) <- d
            )
        ) g.(u)
    done;
    dist
```

Algorithme de Dijkstra : Avec une file de priorité

Problème : si on veut implémenter la file de priorité avec un tas, la fonction de mise à jour d'un élément dans un tas demande de connaître l'indice de l'élément à modifier.

Il faudrait maintenir un tableau qui donne l'indice (dans le tas) d'un sommet. C'est fastidieux (voir info-llg.fr qui le fait)...

On pourrait utiliser un ABR : pour mettre à jour il suffit d'appeler `del` puis `add`.

Algorithme de Dijkstra : Avec une file de priorité

Ma solution « personnelle » plus simple : ajouter des couples (distance estimée de `v`, `v`) sans jamais mettre à jour les éléments de la FP. On peut donc avoir plusieurs fois le même sommet dans la FP :

Initialement : `q` contient `(0, r)`
`dist.(r) <- 0` et `dist.(v) <-` ∞, $\forall$ `v` $\neq$ `r`

Tant que `q` $\neq \emptyset$:
Extraire `(d, u)` de `q` tel que `d` soit minimum
Si `dist.(u)` $= \infty$:
`dist.(u) <- d`
Pour tout voisin `v` de `u` :
Ajouter `(d + w u v, v)` à `q`

Algorithme de Dijkstra : Avec une file de priorité

```
let dijkstra g w s =
  let n = Array.length g in
  let q = make () in
  let dist = Array.make n max_int in
  add q s 0;
  while not (is_empty q) do
    let d, u = extract_min q in
    if dist.(u) = max_int then (
        dist.(u) <- d;
        List.iter (fun v -> add q (v, d + w u v)) g.(u)
    )
  done;
  dist
```

Remarque : ici je suppose que `extract_min` renvoie un couple (priorité, sommet).

Plus courts chemins avec Dijkstra

Initialement : `next` contient tous les sommets
`dist.(r) <- 0` et `dist.(v) <- ∞`, $\forall$ `v` $\neq$ `r`

Tant que `next` $\neq \emptyset$:
Extraire `u` de `next` tel que `dist.(u)` soit minimum
Pour tout voisin `v` de `u` :
`dist.(v) <- min dist.(v) (dist.(u) + w u v)`

Question

Comment modifier l'algorithme pour connaître les plus courts chemins?

Plus courts chemins avec Dijkstra

Tant que `next` $\neq \emptyset$:

```
    Extraire u de next tel que dist.(u) soit minimum
    Pour tout voisin v de u :
        Si dist.(v) > dist.(u) + w u v :
            dist.(v) <- dist.(u) + w u v
            pere.(v) <- u
```

On peut conserver dans `pere.(v)` le prédécesseur de `v` dans un plus court chemin de `r` à `v`.
`v`, `pere.(v)`, `pere.(pere.(v))`, ... jusqu'à `r` donne un chemin (à l'envers) de `r` à `v`.

Le graphe des pères est un arbre (**un arbre des plus courts chemins**).