Plus courts chemins dans un graphe avec l'algorithme de Dijkstra

Quentin Fortier

Graphe pondéré : Poids

On considère dans ce cours 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.

Graphe pondéré : Poids

On considère dans ce cours 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 $+\infty$.

Graphe pondéré : Poids

On considère dans ce cours seulement des graphes orientés.

Définition

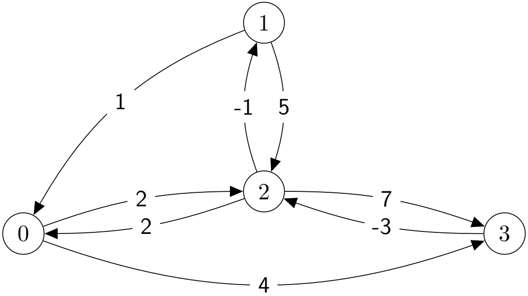
Un graphe **pondéré** est un graphe $\overrightarrow{G} = (V, \overrightarrow{E})$ muni d'une fonction de poids $w : \overrightarrow{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

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 une **matrice d'adjacence pondéré**, contenant $w(u, v)$ sur la ligne u, colonne v.

Graphe pondéré : Poids

Exemple de graphe représenté par matrice d'adjacence pondérée :



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

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

Graphe pondéré : Chemin

Définition

1. Le **poids d'un chemin** C, noté $w(C)$ est la somme des poids de ses arêtes.
2. Un chemin de $u \in V$ à $v \in V$ est un **plus court chemin** s'il n'existe pas de chemin de poids plus petit.

Graphe pondéré : Chemin

Définition

1. Le **poids d'un chemin** C, noté $w(C)$ est la somme des poids de ses arêtes.
2. Un chemin de $u \in V$ à $v \in V$ est un **plus court chemin** s'il n'existe pas de chemin de poids plus petit.

Exercice

Écrire une fonction `poids_chemin(C, G)` qui calcule le poids d'un chemin C (donné par la liste des ses sommets) dans le graphe représenté par la matrice d'adjacence pondéré G.

Graphe pondéré : Distance

Définition

La **distance** $d(u, v)$ est le poids d'un plus court chemin de u à v.
Autrement dit :

$$d(u, v) = \inf\{w(C) \mid C \text{ est un chemin de } u \text{ à } v\}$$

Graphe pondéré : Distance

Définition

La **distance** $d(u, v)$ est le poids d'un plus court chemin de u à v.
Autrement dit :

$$d(u, v) = \inf\{w(C) \mid C \text{ est un chemin de } u \text{ à } v\}$$

Il peut ne pas y avoir de plus court chemin de u à v...

Graphe pondéré : Distance

Définition

La **distance** $d(u, v)$ est le poids d'un plus court chemin de u à v.
Autrement dit :

$$d(u, v) = \inf\{w(C) \mid C \text{ est un chemin de } u \text{ à } v\}$$

Il peut ne pas y avoir de plus court chemin de u à v...

- ... si v n'est pas atteignable depuis u : on pose $d(u, v) = \infty$.

Graphe pondéré : Distance

Définition

La **distance** $d(u, v)$ est le poids d'un plus court chemin de u à v.
Autrement dit :

$$d(u, v) = \inf\{w(C) \mid C \text{ est un chemin de } u \text{ à } v\}$$

Il peut ne pas y avoir de plus court chemin de u à v...

- ... si v n'est pas atteignable depuis u : on pose $d(u, v) = \infty$.
- ... s'il existe un cycle de poids négatif : on pose $d(u, v) = -\infty$.

Graphe pondéré : Distance

Inégalité triangulaire

S'il n'y a pas de cycle de poids négatif :

$$\boxed{d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)}$$

Graphe pondéré : Distance

Inégalité triangulaire

S'il n'y a pas de cycle de poids négatif :

$$\boxed{d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)}$$

Preuve :
Concaténer un plus court chemin de v_1 à v_3 et un plus court chemin de v_3 à v_2 donne un chemin de v_1 à v_2 de poids $d(v_1, v_3) + d(v_3, v_2)$.
Ce poids est donc supérieur au poids $d(v_1, v_2)$ d'un plus court chemin de v_1 à v_2.

Graphe pondéré : Distance

Sous-optimalité

Soit C un plus court chemin de u à v et u', v' deux sommets de C. Alors le sous-chemin de C de u' à v' est aussi un plus court chemin.

Graphe pondéré : Distance

Sous-optimalité

Soit C un plus court chemin de u à v et u', v' deux sommets de C. Alors le sous-chemin de C de u' à v' est aussi un plus court chemin.

Preuve :
Si ce n'était pas le cas on pourrait le remplacer par un chemin plus court pour obtenir un chemin de u à v plus court que C (absurde).

Plus courts chemins

L'objectif du cours est de résoudre le problème suivant :

Problème

Entrée : $\vec{G} = (V, \vec{E})$ un graphe pondéré orienté et $s \in V$.
Sortie : Une liste contenant $d(s, v)$, pour tout $v \in V$.

Plus courts chemins

L'objectif du cours est de résoudre le problème suivant :

Problème

Entrée : $\vec{G} = (V, \vec{E})$ un graphe pondéré orienté et $s \in V$.
Sortie : Une liste contenant $d(s, v)$, pour tout $v \in V$.

Cas particuliers :

- Tous les poids sont égaux :

Plus courts chemins

L'objectif du cours est de résoudre le problème suivant :

Problème

Entrée : $\overrightarrow{G} = (V, \overrightarrow{E})$ un graphe pondéré orienté et $s \in V$.
Sortie : Une liste contenant $d(s, v)$, pour tout $v \in V$.

Cas particuliers :

- Tous les poids sont égaux : **parcours en largeur** depuis s.

Plus courts chemins

L'objectif du cours est de résoudre le problème suivant :

Problème

Entrée : $\vec{G} = (V, \vec{E})$ un graphe pondéré orienté et $s \in V$.
Sortie : Une liste contenant $d(s, v)$, pour tout $v \in V$.

Cas particuliers :

- Tous les poids sont égaux : **parcours en largeur** depuis s.
- Tous les poids sont positifs : **algorithme de Dijkstra** depuis s.

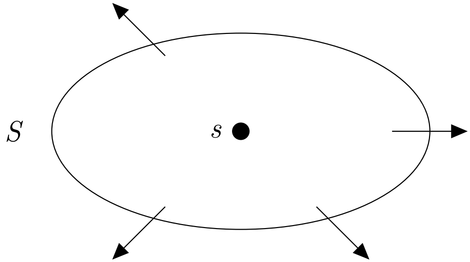
Algorithme de Dijkstra

Idée : Calculer les distances par ordre croissant depuis s.

Algorithme de Dijkstra

Idée : Calculer les distances par ordre croissant depuis s.

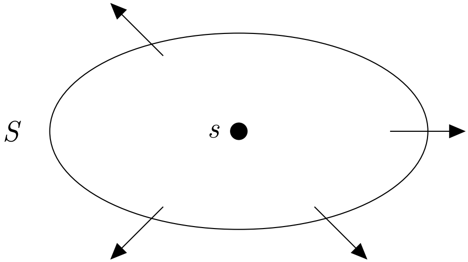
Soit $S \subset V$ l'ensemble des sommets de distance connue.



Algorithme de Dijkstra

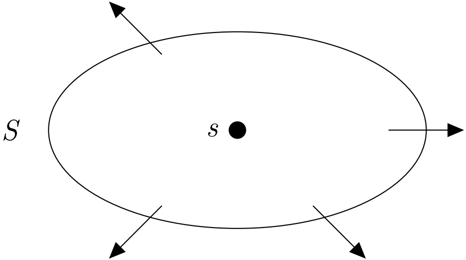
<u>Idée</u> : Calculer les distances par ordre croissant depuis s.

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



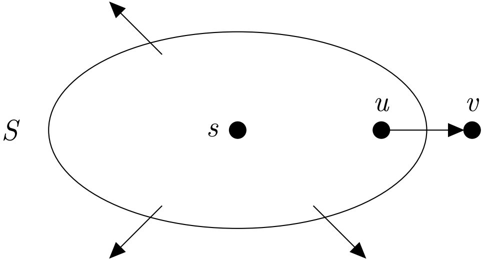
À chaque étape, on déduit la distance à un sommet de plus (et on l'ajoute à S).

Algorithme de Dijkstra



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

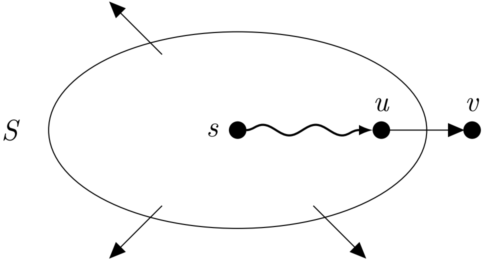
Algorithme de Dijkstra



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

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

Algorithme de Dijkstra



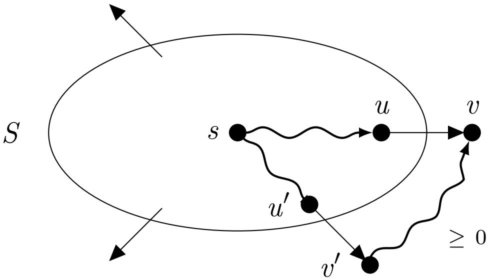
Soit $(u, v) \in \vec{E}$ tel que $v \notin S$ et $d(s, u) + w(u, v)$ est minimum.
Alors :

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

<u>Preuve</u> :

1. Il existe un chemin de longueur $d(s, u) + w(u, v)$.

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]` $= \min\limits_{u \notin q} d(s, u) + w(u, v)$.

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]` $= \min\limits_{u \notin q} d(s, u) + w(u, v)$.

Algorithme de Dijkstra :

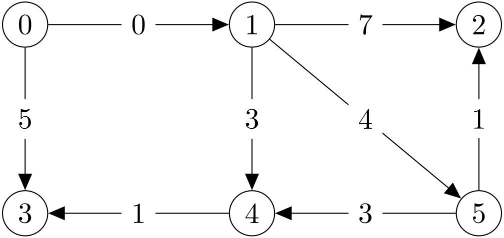
```
Initialement : q contient tous les sommets
               dist[s] = 0
               dist[v] = float("inf"), si v != s

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 : File de priorité

L'algorithme de Dijkstra demande d'extraire le minimum de q, ce qui peut être réalisé efficacement avec une file de priorité :

Définition

Une **file de priorité** est une structure de données permettant d'ajouter un élément et d'obtenir le minimum.

Algorithme de Dijkstra : File de priorité

L'algorithme de Dijkstra demande d'extraire le minimum de q, ce qui peut être réalisé efficacement avec une file de priorité :

Définition

Une **file de priorité** est une structure de données permettant d'ajouter un élément et d'obtenir le minimum.

Il existe plusieurs implémentation de file de priorité, notamment par arbre binaire de recherche ou tas. Voir le cours de MP2I pour plus de détails.

Algorithme de Dijkstra : File de priorité

L'algorithme de Dijkstra demande d'extraire le minimum de q, ce qui peut être réalisé efficacement avec une file de priorité :

Définition

Une **file de priorité** est une structure de données permettant d'ajouter un élément et d'obtenir le minimum.

Il existe plusieurs implémentation de file de priorité, notamment par arbre binaire de recherche ou tas. Voir le cours de MP2I pour plus de détails.

En Python, on utilisera heapq (qui implémente une file de priorité par tas).

Algorithme de Dijkstra : File de priorité

Opérations sur une file de priorité q :

- `heappush(q, e)` : ajoute e à q.
- `heappop(q)` : supprime et renvoie le minimum de q.

```python
import heapq # pour utiliser une file de priorité

q = [] # file de priorité vide
heapq.heappush(q, 3) # ajoute 3 dans q
heapq.heappush(q, 1)
heapq.heappush(q, 6)
heapq.heappop(q) # renvoie 1
heapq.heappop(q) # renvoie 3
```

Algorithme de Dijkstra : File de priorité

Opérations sur une file de priorité q :

- `heappush(q, e)` : ajoute e à q.
- `heappop(q)` : supprime et renvoie le minimum de q.

```
import heapq # pour utiliser une file de priorité

q = [] # file de priorité vide
heapq.heappush(q, 3) # ajoute 3 dans q
heapq.heappush(q, 1)
heapq.heappush(q, 6)
heapq.heappop(q) # renvoie 1
heapq.heappop(q) # renvoie 3
```

Remarque : On peut aussi mettre des couples dans q, et `heappop` extrait le couple dont le 1er élément est minimum.

Algorithme de Dijkstra : File de priorité

Dans l'algorithme de Dijkstra, il faut normalement mettre à jour un élément déjà existant dans la file de priorité... Mais cette opération n'existe pas dans `heapq`.

Algorithme de Dijkstra : File de priorité

Dans l'algorithme de Dijkstra, il faut normalement mettre à jour un élément déjà existant dans la file de priorité... Mais cette opération n'existe pas dans `heapq`.

On peut utiliser une variante de l'algorithme de Dijkstra : ne jamais modifier un élément de `q` mais l'ajouter à nouveau avec une distance estimée mise à jour.
On peut donc avoir plusieurs fois le même sommet dans `q`.

Algorithme de Dijkstra : File de priorité

On utilise une file de priorité `q` contenant des couples (distance estimée de `v`, `v`).

Algorithme de Dijkstra : File de priorité

On utilise une file de priorité `q` contenant des couples (distance estimée de `v`, `v`).

Variante de l'algorithme de Dijkstra :

```
Initialement : q contient (0, s)
               dist[s] = 0
               dist[v] = float("inf"), si v != s

Tant que q est non vide:
    Extraire (d, u) de q tel que d soit minimum
    Si dist[u] == float("inf"):
        dist[u] = d
        Pour tout voisin v de u:
            Ajouter (d + w(u, v), v) dans q
```

Algorithme de Dijkstra : File de priorité

On utilise une file de priorité `q` contenant des couples (distance estimée de `v`, `v`).

Algorithme de Dijkstra : File de priorité

On utilise une file de priorité q contenant des couples (distance estimée de v, v).

Variante de l'algorithme de Dijkstra :

```
import heapq

def dijkstra(G, s): # G : matrice d'adjacence pondérée
    dist = [float("inf")]*len(G)
    q = []
    heapq.heappush(q, (0, s))
    while len(q) > 0:
        d, u = heapq.heappop(q)
        if dist[u] == float("inf"):
            dist[u] = d
            for v in range(len(G)):
                if G[u][v] != float("inf"):
                    heapq.heappush(q, (dist[u] + G[u][v], v))
    return dist
```