Plus courts chemins dans les graphes pondérés

Quentin Fortier

Graphe pondéré

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

Graphe pondéré

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

Le **poids d'un chemin** est la somme des poids de ses arêtes.

Graphe pondéré

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

Le **poids d'un chemin** est la somme des poids de ses arêtes.

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

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

Le **poids d'un chemin** est la somme des poids de ses arêtes.

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.

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

Graphe pondéré

Il est pratique de dire $w(u, v) = \infty$ s'il n'y a pas d'arête entre u et v. En OCaml, on utilise `max_int`. Mais on aura besoin de gérer les dépassements :

```
let sum x y =
  if x = max_int || y = max_int then max_int
  else x + y
```

Distance

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

Distance

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

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

Distance

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

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

Remarque : s'il n'y a pas de cycle de poids ≤ 0, les plus courts chemins sont élémentaires (ils passent au plus une fois sur un sommet) donc sont de longueur au plus $n - 1$.

Propriétés

Inégalité triangulaire

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

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

Propriétés

Inégalité triangulaire

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

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

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.

Propriétés

Inégalité triangulaire

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

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

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.

Sous-optimalité

L'optimalité des sous-problèmes est cruciale pour montrer que certains algorithmes/raisonnements sont corrects, par exemple :

1. pour utiliser la programmation dynamique / diviser pour régner.
2. un sous-arbre d'un ABR (optimal) est un ABR (optimal).
3. un sous-arbre d'un tas est un tas.

Pensez à le mentionner et le justifier si besoin est...

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** :

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** : BFS en O$(n + p)$
2. ...depuis un sommet à tous les autres quand il n'y a **pas de cycle** :

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** : BFS en $\mathrm{O}(n+p)$
2. ...depuis un sommet à tous les autres quand il n'y a **pas de cycle** : tri topologique + prog. dyn. en $\mathrm{O}(n+p)$
3. ...depuis un sommet à tous les autres quand les **poids** sont **positifs** :

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** : BFS en O($n + p$)
2. ...depuis un sommet à tous les autres quand il n'y a **pas de cycle** : tri topologique + prog. dyn. en O($n + p$)
3. ...depuis un sommet à tous les autres quand les **poids** sont **positifs** : Dijkstra en O($p \log(n)$)
4. ...depuis un sommet à tous les autres (et détecter un cycle de poids négatif) :

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** : BFS en O$(n + p)$
2. ...depuis un sommet à tous les autres quand il n'y a **pas de cycle** : tri topologique + prog. dyn. en O$(n + p)$
3. ...depuis un sommet à tous les autres quand les **poids** sont **positifs** : Dijkstra en O$(p \log(n))$
4. ...depuis un sommet à tous les autres (et détecter un cycle de poids négatif) : Bellman-Ford (prog. dyn.) en O(pn)
5. ...**entre tout couple de sommets** (et détecter un cycle de poids négatif) :

Problèmes de plus courts chemins

Pour trouver des plus courts chemins...

1. ...depuis un sommet à tous les autres quand tous les **poids** sont **égaux** : BFS en O($n + p$)
2. ...depuis un sommet à tous les autres quand il n'y a **pas de cycle** : tri topologique + prog. dyn. en O($n + p$)
3. ...depuis un sommet à tous les autres quand les **poids** sont **positifs** : Dijkstra en O($p \log(n)$)
4. ...depuis un sommet à tous les autres (et détecter un cycle de poids négatif) : Bellman-Ford (prog. dyn.) en O(pn)
5. ...**entre tout couple de sommets** (et détecter un cycle de poids négatif) : Floyd-Warshall (prog. dyn.) en O(n^3).

Graphe non-orienté

Nous allons écrire les algorithmes de plus courts chemins pour des graphes orientés, mais ils fonctionnent aussi pour des graphes non-orientés, **à condition que tous les poids soient positifs**.

Graphe non-orienté

Nous allons écrire les algorithmes de plus courts chemins pour des graphes orientés, mais ils fonctionnent aussi pour des graphes non-orientés, **à condition que tous les poids soient positifs**.

Soit G un graphe non-orienté et $\overrightarrow{G}$ obtenu à partir de G en remplaçant chaque arête $\{u, v\}$ par deux arcs de même poids (u, v) et (v, u) :

Graphe non-orienté

Nous allons écrire les algorithmes de plus courts chemins pour des graphes orientés, mais ils fonctionnent aussi pour des graphes non-orientés, **à condition que tous les poids soient positifs**.

Soit G un graphe non-orienté et $\vec{G}$ obtenu à partir de G en remplaçant chaque arête $\{u, v\}$ par deux arcs de même poids (u, v) et (v, u) :

1. Si les poids de G sont ≥ 0, les distances entre sommets sont les mêmes dans G et $\vec{G}$.
2. G et $\vec{G}$ ont même matrice d'adjacence et même liste d'adjacence.

Algorithme de Dijkstra

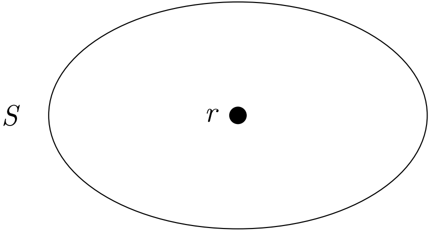
But : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.

Algorithme de Dijkstra

<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.

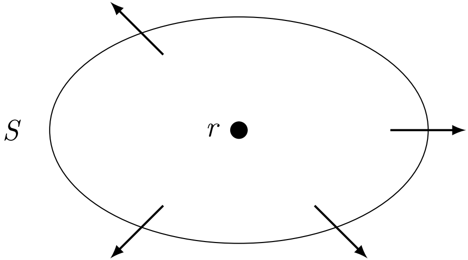
Algorithme de Dijkstra

<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



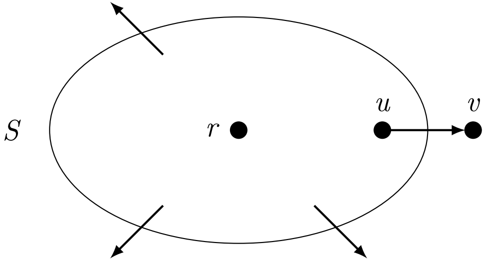
Algorithme de Dijkstra

<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



Algorithme de Dijkstra

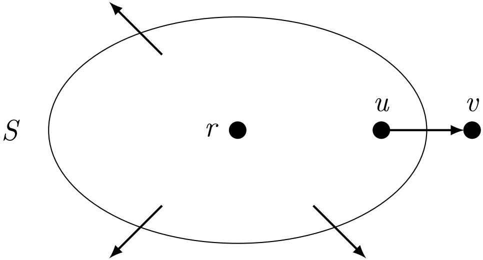
<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



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

Algorithme de Dijkstra

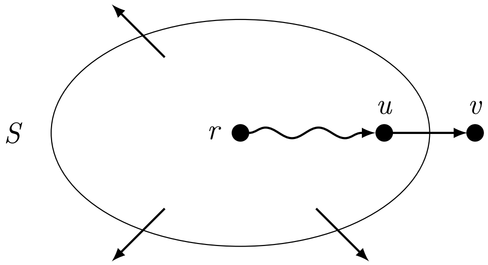
<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



Soit $(u, v) \in \vec{E}$ tel que $v \notin S$ et $d(r, u) + w(u, v)$ soit minimum.
Alors, si tous les poids sont ≥ 0 : $\boxed{d(r, v) = d(r, u) + w(u, v)}$.

Algorithme de Dijkstra

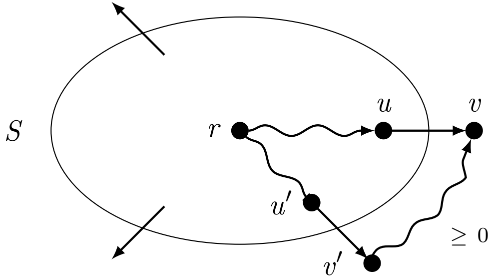
<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



En effet : 1) Il existe un chemin de longueur $d(r, u) + w(u, v)$.

Algorithme de Dijkstra

<u>But</u> : calculer $d(r, v)$, $\forall v \in V$ dans $\vec{G} = (V, \vec{E})$ où les poids sont ≥ 0.
<u>Idée</u> : calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



En effet : 1) Il existe un chemin de longueur $d(r, u) + w(u, v)$.
2) Un chemin C de r à v doit sortir de S avec un arc (u', v'). Comme les poids sont ≥ 0, poids(C) $\geq d(r, u') + w(u', v') \geq d(r, u) + w(u, v)$.

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans `next` et on conserve un tableau `dist` tel que :

1. $\forall v \notin \texttt{next} : \texttt{dist.(v)} = d(r, v)$.
2. $\forall v \in \texttt{next} : \texttt{dist.(v)} = \min_{u \notin next} d(r, u) + w(u, v)$.

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans `next` et on conserve un tableau `dist` tel que :

1. $\forall v \notin$ `next` : `dist.(v)` $= d(r, v)$.
2. $\forall v \in$ `next` : `dist.(v)` $= \min_{u \notin next} d(r, u) + w(u, v)$.

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)
```

Remarque : ne marche pas avec des poids négatifs.

Implémentation de Dijkstra avec une file de priorité

Initialement : `next` contient tous les sommets
`dist.(r) <- 0` et `dist.(v) <- ∞`, ∀ `v` ≠ `r`

Tant que `next` ≠ ∅ :

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** :

Implémentation 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 $\longrightarrow \mathsf{O}(n \log(n))$
2. au plus p mises à jour $\longrightarrow \mathsf{O}(p \log(n))$

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

Implémentation de Dijkstra avec une file de priorité

Problème : la fonction de mise à jour d'un élément dans un tas demande de connaître l'indice de l'élément à modifier.

```
let update tas i new_e =
  let prev_e = tas.t.(i) in
  tas.t.(i) <- new_e;
  if prev_e < new_e then monter tas i
  else descendre tas i;;
```

Implémentation de Dijkstra avec une file de priorité

Problème : la fonction de mise à jour d'un élément dans un tas demande de connaître l'indice de l'élément à modifier.

```
let update tas i new_e =
  let prev_e = tas.t.(i) in
  tas.t.(i) <- new_e;
  if prev_e < new_e then monter tas i
  else descendre tas i;;
```

Il faudrait maintenir un tableau qui donne l'indice (dans le tas) d'un sommet. C'est fastidieux ...

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

Implémentation 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 :

Implémentation 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`

Implémentation 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.

```
let dijkstra g w r =
  let n = Array.length g in
  let q = empty () in (* file de priorité vide *)
  let dist = Array.make n max_int in (* dist.(v) va être d(r, v) *
  add q (0, r);
  while not (is_empty q) do
    let d, u = take_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
```

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 ≠ ∅ :
      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`.

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**).

Plus courts chemins avec Dijkstra

On stocke des triplets (distance estimé de `v`, `v`, père de `v`) dans la FP :

```
let dijkstra g w r =
  let n = Array.length g in
  let q = empty () in
  let pere = Array.make n (-1) in
  add q (0, r, r);
  while not (is_empty q) do
    let d, u, p = take_min q in
    if pere.(u) = max_int then (
      pere.(u) <- p;
      List.iter (fun v -> add q (sum d (w u v)), v, u) g.(u)
    )
  done;
  pere
```

(On n'a plus besoin de `dist`)

Plus courts chemins entre toutes paires de sommets

On peut aussi utiliser de la programmation dynamique :

1. Bellman-Ford pour trouver les plus courts chemins depuis une racine fixée
2. Floyd-Warshall pour trouver tous les plus courts chemins

Ces algorithmes marchent même s'il y a des poids négatifs, contrairement à Dijkstra.

	Dijkstra	Bellman-Ford	Floyd-Warshall
Complexité	O$(p\log(n))$	O(np)	O(n^3)
Contraine	Poids positifs	Aucune	Aucune

Programmation dynamique

Notre problème est P = « trouver $d(u, v)$, pour tous sommets u, v ».

Programmation dynamique

Notre problème est P $=$ « trouver $d(u,v)$, pour tous sommets u, v ».

Programmation dynamique

Notre problème est P $=$ « trouver $d(u,v)$, pour tous sommets u, v ».

On peut écrire l'équation :

$$d(u,v) = \min_{v'} d(u,v') + w(v',u)$$

Programmation dynamique

Notre problème est P = « trouver $d(u, v)$, pour tous sommets u, v ».

On peut écrire l'équation :

$$d(u, v) = \min_{v'} d(u, v') + w(v', u)$$

On ne se ramène pas à des sous-problèmes plus petits !
Il faut introduire un paramètre :

1. Bellman : nombre d'arêtes
2. Floyd-Warshall : sommets que l'on peut utiliser

Bellman-Ford

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

Bellman-Ford

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

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

Bellman-Ford

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

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

<u>Preuve</u> : soit C un plus court chemin de r à v utilisant au plus $k+1$ arêtes.

Bellman-Ford

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

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

<u>Preuve</u> : soit C un plus court chemin de r à v utilisant au plus $k+1$ arêtes.
Soit u le prédecesseur de v dans C.

Bellman-Ford

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

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

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

Bellman-Ford

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

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

<u>Preuve</u> : 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).

Bellman-Ford

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

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

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).

Bellman-Ford

On va utiliser un tableau `d[v][k]` pour stocker $d_k(v)$.

Algorithme de Bellman-Ford

`d[r] <- 0`
Pour `v` $\neq$ `r` :
    Pour `k` $= 0$ à $n-2$:
        `d[v][k] <-` ∞

Pour $k = 0$ à $n-2$:
    Pour tout sommet `v` :
        Pour tout arc (`u`, `v`) entrant dans `v` :
            Si `d[u][k] + w(u, v) < d[v][k + 1]` :
                `d[v][k + 1]` $\leftarrow$ `d[u][k] + w(u, v)`

Bellman-Ford

On va utiliser un tableau `d[v][k]` pour stocker $d_k(v)$.

Algorithme de Bellman-Ford

d[r] <- 0
Pour v $\neq$ r :
    Pour k $=0$ à $n-2$:
        d[v][k] <- ∞

Pour $k = 0$ à $n-2$:
    Pour tout sommet v :
        Pour tout arc (u, v) entrant dans v :
            Si d[u][k] + w(u, v) < d[v][k + 1] :
                d[v][k + 1] $\leftarrow$ d[u][k] + w(u, v)

Complexité :

Bellman-Ford

On va utiliser un tableau `d[v][k]` pour stocker $d_k(v)$.

Algorithme de Bellman-Ford

`d[r] <- 0`
Pour `v` $\neq$ `r` :
  Pour `k` $= 0$ à $n-2$:
    `d[v][k] <-` ∞

Pour $k = 0$ à $n-2$:
  Pour tout sommet `v` :
    Pour tout arc (`u`, `v`) entrant dans `v` :
      Si `d[u][k] + w(u, v) < d[v][k + 1]` :
        `d[v][k + 1]` $\leftarrow$ `d[u][k] + w(u, v)`

<u>Complexité</u> : O(np)

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

d[r] <- 0
Pour v $\neq$ r :
    Pour k $= 0$ à $n - 2$:
        d[v][k] <- ∞

Pour $k = 0$ à $n - 2$:
    Pour tout arc (u, v) :
        Si d[u][k] + w(u, v) < d[v][k + 1] :
            d[v][k + 1] $\leftarrow$ d[u][k] + w(u, v)

Bellman-Ford

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

Bellman-Ford

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

Algorithme de Bellman-Ford

```
d[r] <- 0
Pour v ≠ r :
    d[v] <- ∞

Pour k = 0 à n − 2 :
    d' ← copie de d
    Pour tout arc (u, v) :
        Si d[u] + w(u, v) < d[v] :
            d[v] ← d'[u] + w(u, v)
```

Bellman-Ford

```
let bellman g w r =
  let n = Array.length g in
  let d = Array.make n max_int in
  d.(r) <- 0;
  for k = 0 to n - 2 do
    for u = 0 to n - 1 do
      List.iter (fun v ->
        d.(v) <- min d.(v) (sum d.(u) (w u v))
      ) g.(u)
    done
  done;
  d
```

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).

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).
On va résoudre P(k) $=$ « trouver $d_k(u, v)$, pour tous sommets u, v ».

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).
On va résoudre P(k) $=$ « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

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). On va résoudre P(k) $=$ « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

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).
On va résoudre P(k) $=$ « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :
$$d_{k+1}(u, v) = d_k(u, v)$$
2. Si C utilise k comme sommet intermédiaire :

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).
On va résoudre P(k) = « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :
$$d_{k+1}(u, v) = d_k(u, v)$$
2. Si C utilise k comme sommet intermédiaire :
$$d_{k+1}(u, v) = d_k(u, k) + d_k(k, v)$$

Équation de récurrence :

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).
On va résoudre P(k) = « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

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

2. Si C utilise k comme sommet intermédiaire :

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

Équation de récurrence :

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

Floyd-Warshall

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

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

Floyd-Warshall

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

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

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

Floyd-Warshall

Initialiser $d_0(u, v) \leftarrow w(u, v)$ si $(u, v) \in \vec{E}$.
∞ sinon.

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

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

Floyd-Warshall

Initialiser $d_0(u,v) \leftarrow w(u,v)$ si $(u,v) \in \vec{E}$.
∞ sinon.

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

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

Floyd-Warshall

`d.(u).(v)` contient le dernier $d_k(u, v)$ calculé :

Initialiser `d.(u).(v)` $\leftarrow w(u, v)$ si $(u, v) \in \overrightarrow{E}$, ∞ sinon.

Pour $k = 0$ à $n - 1$:

  Pour tout sommet u :

    Pour tout sommet v :

      `d.(u).(v)` $\leftarrow$ `min d.(u).(v) (d.(u).(k) + d.(k).(v))`

Floyd-Warshall

`d.(u).(v)` contient le dernier $d_k(u, v)$ calculé :

Initialiser `d.(u).(v)` $\leftarrow w(u, v)$ si $(u, v) \in \overrightarrow{E}$, ∞ sinon.

Pour $k = 0$ à $n - 1$:
  Pour tout sommet u :
    Pour tout sommet v :
      `d.(u).(v)` $\leftarrow$ `min d.(u).(v) (d.(u).(k) + d.(k).(v))`

Complexité :

Floyd-Warshall

`d.(u).(v)` contient le dernier $d_k(u, v)$ calculé :

Initialiser `d.(u).(v)` $\leftarrow w(u, v)$ si $(u, v) \in \overrightarrow{E}$, ∞ sinon.

Pour $k = 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :
            `d.(u).(v)` $\leftarrow$ `min d.(u).(v) (d.(u).(k) + d.(k).(v))`

Complexité : $\boxed{\mathrm{O}(n^3)}$

Floyd-Warshall

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

Pour $k = 0$ à $n-1$:

  Pour tout sommet u :

    Pour tout sommet v :

      `d.(u).(v)` $\leftarrow$ `min d.(u).(v) (d.(u).(k) + d.(k).(v))`

Question

Comment détecter un cycle de poids négatif?

Floyd-Warshall

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

Pour $k = 0$ à $n - 1$:

  Pour tout sommet u :

    Pour tout sommet v :

      `d.(u).(v)` $\leftarrow$ `min d.(u).(v) (d.(u).(k) + d.(k).(v))`

Question

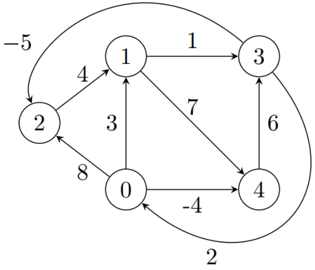
Comment détecter un cycle de poids négatif ?

Il y a un cycle de poids négatif $\iff \exists$ `u`, `d.(u).(u) < 0`.

Floyd-Warshall

On utilise souvent une matrice d'adjacence $A = (a_{i,j})$ modifée pour représenter un graphe $\vec{G} = (V, \vec{E})$ pondéré par w :

- $a_{i,j} = w(i,j)$, si $(i,j) \in \vec{E}$
- $a_{i,j} = 0$, si $i = j$
- $a_{i,j} = \infty$, si $(i,j) \notin \vec{E}$



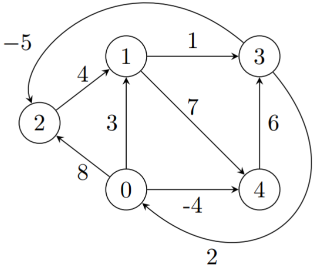
$$A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Matrice d'adjacence

Floyd-Warshall

On utilise souvent une matrice d'adjacence $A = (a_{i,j})$ modifée pour représenter un graphe $\vec{G} = (V, \vec{E})$ pondéré par w :

- $a_{i,j} = w(i,j)$, si $(i,j) \in \vec{E}$
- $a_{i,j} = 0$, si $i = j$
- $a_{i,j} = \infty$, si $(i,j) \notin \vec{E}$



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

Matrice des distances renvoyée par Floyd-Warshall

Floyd-Warshall

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

```
let floyd_warshall g =
    let n = Array.length g in
    let d = Array.map Array.copy g in (* copie de g *)
    for k = 0 to n - 1 do
      for i = 0 to n - 1 do
        for j = 0 to n - 1 do
          d.(i).(j) <- min d.(i).(j) (sum d.(i).(k) d.(k).(j))
        done
      done
    done;
    d
```

Floyd-Warshall

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

Pour $k = 0$ à $n - 1$:

    Pour tout sommet u :

        Pour tout sommet v :

            `d.(u).(v)` $\leftarrow$ `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?

Floyd-Warshall

Initialiser `d.(u).(v)` $\leftarrow w(u, v)$ si $(u, v) \in \overrightarrow{E}$, ∞ sinon.

Pour $k = 0$ à $n - 1$:

  Pour tout sommet u :

    Pour tout sommet v :

      `d.(u).(v)` $\leftarrow$ `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`.

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 $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

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 $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

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

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

2. Si C utilise k comme sommet intermédiaire :

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

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

Floyd-Warshall

Initialiser `d.(u).(v)` $\leftarrow w($`u, v`$)$ si $($`u, v`$) \in \vec{E}$, ∞ sinon.
Initialiser `pere.(u).(v)` $\leftarrow$ `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)` $\leftarrow$ `d.(u).(k) + d.(k).(v)`
                `pere.(u).(v)` $\leftarrow$ `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`.

Floyd-Warshall

```
let init_pere n =
  let pere = make_matrix n n (-1) in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      pere.(i).(j) <- i
    done
  done;
  pere;;
```

```
let floyd_warshall d =
  let n = vect_length d in
  let pere = init_pere n in
  for k = 0 to n - 1 do
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        if d.(i).(j) > sum d.(i).(k) d.(k).(j) then
          (d.(i).(j) <- sum d.(i).(k) d.(k).(j);
           pere.(i).(j) <- pere.(k).(j))
      done
    done
  done;
  pere;;
```

Floyd-Warshall

```
let floyd_warshall d =
  let n = vect_length d in
  let pere = init_pere n in
  for k = 0 to n - 1 do
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        if d.(i).(j) > sum d.(i).(k) d.(k).(j) then
          (d.(i).(j) <- sum d.(i).(k) d.(k).(j);
           pere.(i).(j) <- pere.(k).(j))
      done
    done
  done;
  pere;;
```

```
let rec chemin pere u v =
  if u = v then [u]
  else v::chemin pere u pere.(u).(v);;
```