Plus courts chemins dans les graphes pondérés

MP/MP* Option info

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 $w : E \longrightarrow \mathbb{R}$ (en `Caml`, `w : int -> int -> int` ou `w : int array array`).

Graphe pondéré

Un graphe **pondéré** est un graphe $G = (V, E)$ muni d'une fonction de poids $w : E \longrightarrow \mathbb{R}$ (en `Caml`, `w : int -> int -> int` ou `w : int array array`).

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 $w : E \longrightarrow \mathbb{R}$ (en `Caml`, `w : int -> int -> int` ou `w : int array array`).

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 $w : E \longrightarrow \mathbb{R}$ (en `Caml`, `w : int -> int -> int` ou `w : int array array`).

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, s'il en existe un.

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 $|V| - 1$.

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

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 $\overrightarrow{G}$ 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 $\overrightarrow{G}$.
2. G et $\overrightarrow{G}$ ont même matrice d'adjacence et même liste d'adjacence.

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.

Pourquoi cela ne marche t-il pas s'il y a des poids négatifs?

Algorithme de Dijkstra

<u>But</u>: calculer les distances d'un sommet r aux autres, où les poids sont ≥ 0.

Algorithme de Dijkstra

But: calculer les distances d'un sommet r aux autres, où les poids sont ≥ 0.

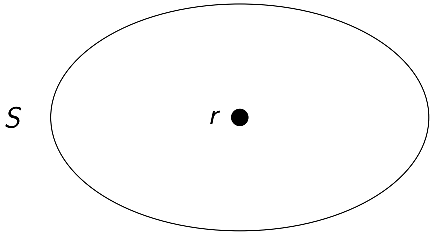
Idée: calculer les distances par ordre croissant, en partant de r.

Algorithme de Dijkstra

But: calculer les distances d'un sommet r aux autres, où les poids sont ≥ 0.

Idée: calculer les distances par ordre croissant, en partant de r.

Supposons connaître les distances de r à tous les sommets de S.

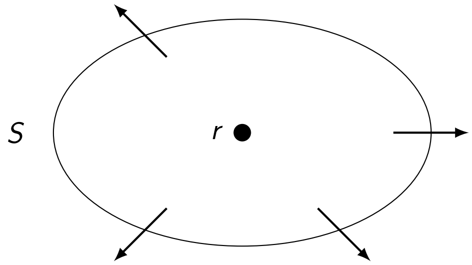


Algorithme de Dijkstra

But: calculer les distances d'un sommet r aux autres, où les poids sont ≥ 0.

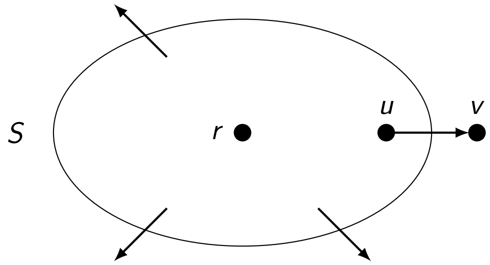
Idée: calculer les distances par ordre croissant, en partant de r.

Supposons connaître les distances de r à tous les sommets de S.



Algorithme de Dijkstra

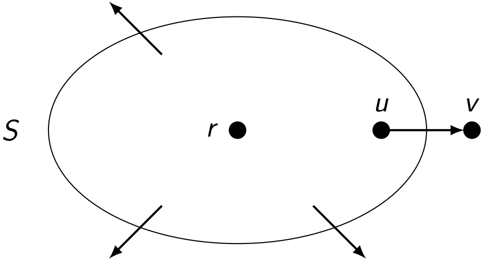
<u>But</u>: calculer les distances d'un sommet r aux autres, où les poids sont ≥ 0.
<u>Idée</u>: calculer les distances par ordre croissant, en partant de r.
Supposons connaître les distances de r à tous les sommets de S.



Soit (u, v) sortant de S tel que $\boldsymbol{d(r,u) + w(u,v)}$ **soit minimum**.

Algorithme de Dijkstra

<u>But</u>: calculer les distances d'un sommet r aux autres, 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.



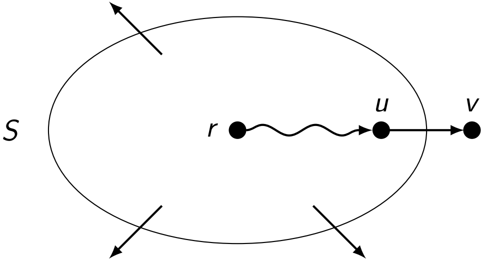
Soit (u, v) sortant de S tel que $\boldsymbol{d(r,u) + w(u,v)}$ **soit minimum**.
Alors, si tous les poids sont ≥ 0: $\boxed{d(r,v) = d(r,u) + w(u,v)}$.

Algorithme de Dijkstra

<u>But</u>: calculer les distances d'un sommet r aux autres, 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.



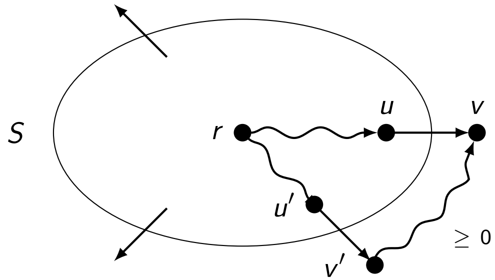
En effet: 1) Il existe un chemin de longueur $d(r, u) + w(u, v)$.

Algorithme de Dijkstra

<u>But</u>: calculer les distances d'un sommet r aux autres, 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` $(= \overline{S})$ 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)` $= \min\limits_{u \notin next} (d(r, u) + w(u, v))$.

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans `next` ($= \overline{S}$) 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)` $= \min\limits_{u \notin next} (d(r, u) + w(u, v))$.

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)`

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans `next` ($= \overline{S}$) 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)` $= \min\limits_{u \notin next} (d(r, u) + w(u, v))$.

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

```
Tant que next ≠ ∅:
    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.

Algorithme de 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)`

Complexité

Algorithme de 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)
```

Complexité si `next` est un **tableau de booléens**:

Algorithme de 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)`

Complexité si `next` est un **tableau de booléens**:
chaque recherche de minimum est en $O(|V|)$ donc la complexité totale est $\boxed{O(|V|^2)}$.

Implémentation de Dijkstra avec un tableau

```ocaml
let dijkstra g w r =
  let dist = Array.make g.n max_int in
  let vu = Array.make g.n false in
  dist.(r) <- 0;
  for k = 0 to g.n - 1 do
    let rec mini i =
      if i = g.n then max_int, i
      else if vu.(i) then mini (i+1)
      else min (dist.(i), i) (mini (i+1)) in
    let du, u = mini 0 in
    vu.(u) <- true;
    do_list (fun v -> dist.(v) <- min dist.(v) (du + w u v)) (g.voisins u);
  done; dist;;
```

`w : int -> int -> int` est la fonction de poids.

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**:

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. $|V|$ extractions du minimum: $O(|V|\log(|V|))$
2. au plus $|\vec{E}|$ mises à jour: $O(|\vec{E}|\log(|V|))$

Total: $O(|V|\log(|V|)) + O(|\vec{E}|\log(|V|)) = \boxed{O(|\vec{E}|\log(|V|))}$.

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. $|V|$ extractions du minimum: $O(|V| \log(|V|))$
2. au plus $|\vec{E}|$ mises à jour: $O(|\vec{E}| \log(|V|))$

Total: $O(|V| \log(|V|)) + O(|\vec{E}| \log(|V|)) = \boxed{O(|\vec{E}| \log(|V|))}$.

C'est mieux qu'avec un tableau si $|\vec{E}| = O\left(\frac{|V|^2}{\log(|V|)}\right)$.

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 (cf infollg)...

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

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

```
let dijkstra g w r =
  let dist = Array.make g.n max_int in
  let fp = fp_new () in
  fp.add (0, r);
  while not fp.is_empty () do
    let d, u = fp.take_min () in
    if dist.(u) = max_int then
      (dist.(u) <- d;
       do_list (fun v -> fp.add (d + w u v, v)) (g.voisins 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` $\neq \emptyset$:

```
    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`.

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`.

Le graphe des pères est un arbre (**un arbre des plus courts chemins**), non unique a priori.

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 pere = Array.make g.n (-1) in
  let fp = fp_new () in
  fp.add (0, r, r);
  while not fp.is_empty () do
    let d, u, p = fp.take_min () in
    if pere.(u) = -1 then
      (pere.(u) <- p;
      do_list (fun v -> fp.add (d + w u v, v, u)) (g.voisins u))
  done;
  pere;;
```

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

Plus courts chemins entre toutes paires de sommets

On veut connaître les plus courts chemins entre toutes les paires de sommets dans un graphe $G = (V, \overrightarrow{E})$.

Plus courts chemins entre toutes paires de sommets

On veut connaître les plus courts chemins entre toutes les paires de sommets dans un graphe $G = (V, \overrightarrow{E})$.

1. Si tous les poids sont ≥ 0: on peut utiliser Dijkstra depuis chaque sommet, en $O(|V| \times |\overrightarrow{E}| \log(|V|))$.

Plus courts chemins entre toutes paires de sommets

On veut connaître les plus courts chemins entre toutes les paires de sommets dans un graphe $G = (V, \vec{E})$.

1. Si tous les poids sont ≥ 0: on peut utiliser Dijkstra depuis chaque sommet, en $O(|V| \times |\vec{E}| \log(|V|))$.
2. Si les poids sont quelconques: on va voir l'algorithme de **Floyd-Warshall** par programmation dynamique en $O(|V|^3)$.

Programmation dynamique

Pour pouvoir résoudre un problème P par programmation dynamique:

1. S'intéresser plus généralement à P(k), où k est un paramètre et P(n) $=$ P.
2. Montrer comment résoudre le cas de base (souvent P(0)).
3. Montrer que la résolution de P(k) se ramène à P(k'), $k' < k$ (trouver une équation de récurrence).

Programmation dynamique

Pour pouvoir résoudre un problème P par programmation dynamique:

1. S'intéresser plus généralement à $\mathrm{P}(k)$, où k est un paramètre et $\mathrm{P}(n) = \mathrm{P}$.
2. Montrer comment résoudre le cas de base (souvent $\mathrm{P}(0)$).
3. Montrer que la résolution de $\mathrm{P}(k)$ se ramène à $\mathrm{P}(k')$, $k' < k$ (trouver une équation de récurrence).

Pour implémenter la méthode, stocker les solutions des $\mathrm{P}(k)$ dans un tableau et calculer les $\mathrm{P}(k)$ de proche en proche:

```
for k = ... to ... do p.(k) <- ...
```

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:

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

Mais ça ne permet pas de faire de la programmation dynamique: on ne se ramène pas à des sous-problèmes « plus petits »!

Programmation dynamique: 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 $< k$

Programmation dynamique: 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'y a pas de chemin).

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « trouver $d_k(u, v)$, pour tous sommets u, v ».

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « 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 $< k + 1$.

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « 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:

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « 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:

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « 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:

Programmation dynamique: 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'y a pas de chemin).
On va calculer $\mathrm{P}(k) =$ « 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:

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

Programmation dynamique: Floyd-Warshall

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

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

Programmation dynamique: Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
    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)`.

Programmation dynamique: Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
  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)` contienne le dernier $d_k(u,v)$ calculé

Programmation dynamique: Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
  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)` contienne le dernier $d_k(u,v)$ calculé (ça marche car $d_{k+1}(u,k) = d_k(u,k)$).

Programmation dynamique: Floyd-Warshall

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

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

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

Programmation dynamique: Floyd-Warshall

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

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

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

Complexité:

Programmation dynamique: Floyd-Warshall

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

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

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

<u>Complexité</u>: $\boxed{O(|V|^3)}$

Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
  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 $(u, v) \in \overrightarrow{E}$, ∞ sinon.

Pour $k = 0$ à $|V| - 1$:
    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`.

Infini en Caml

`max_int` est le plus grand entier représentable en Caml (dépend du processeur: 32 ou 64 bits).

Problèmes :

```
#max_int;;
- : int = 4611686018427387903
#pow 2 62 - 1;;
- : int = 4611686018427387903
#max_int + 1;;
- : int = -4611686018427387904
#max_int + max_int;;
- : int = -2
```

Infini en Caml

`max_int` est le plus grand entier représentable en Caml (dépend du processeur: 32 ou 64 bits).

Problèmes :

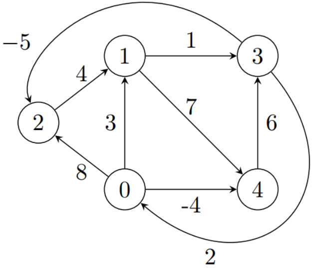
```
#max_int;;
- : int = 4611686018427387903
#pow 2 62 - 1;;
- : int = 4611686018427387903
#max_int + 1;;
- : int = -4611686018427387904
#max_int + max_int;;
- : int = -2
```

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

Matrice d'adjacence pondérée

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 $(i,j) \notin \vec{E}$



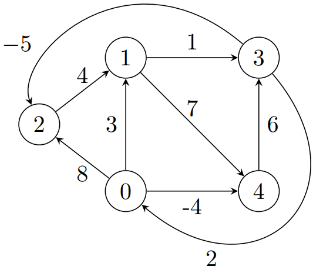
$$A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Matrice d'adjacence

Matrice d'adjacence pondérée

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 d =
  let n = Array.length d in
  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;;
```

Floyd-Warshall

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

```
let floyd_warshall d =
  let n = Array.length d in
  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;;
```

On pourrait copier l'entrée pour éviter de modifier la matrice d'adjacence à l'extérieur de la fonction...

Programmation dynamique: Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:

  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 autre v?

Programmation dynamique: Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
  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 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`.

Plus courts chemins avec Floyd-Warshall

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

Pour $k = 0$ à $|V| - 1$:
    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 = Array.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 = Array.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 = Array.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);;
```