MTH6412B

Implémentation d'algorithmes de recherche opérationnelle

Algorithmes de plus court chemin

Dominique Orban
dominique.orban@polymtl.ca

Mathématiques et Génie Industriel
École Polytechnique de Montréal

Menu du jour

Vue d'ensemble

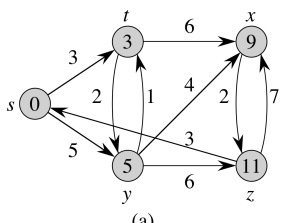
Différentes structures et différents problèmes :

- approche générale ;
- cas particuliers
  - graphe acyclique
  - arcs / arêtes de poids 1 ;
  - arcs / arêtes de poids ≥ 0 ;
  - du sommet source s à un sommet t spécifique ;
  - du sommet source s à tous les sommets $v \in S$;
  - entre toutes les paires $(s,t) \in S \times S$;
- Approche d'analyse : raffinements successifs à partir d'un algorithme générique jusqu'à des algorithmes spécialisés.

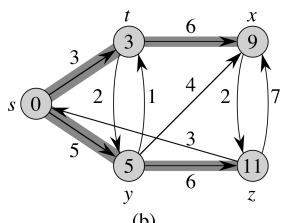
Définitions

- Graphe pondéré: $G = (S,A)$ avec fonction de coût $c : A \to \mathbb{R}$;
- on note $n := |S|$ et $m := |A|$;
- on note $s \rightsquigarrow v$ un chemin de $s \in S$ à $v \in S$;
- le coût d'un chemin est défini comme la somme, sur la séquence d'arcs formant le chemin, des coûts des arcs individuels ;
- problème générique : trouver, pour un graphe pondéré $G = (S,A)$, le coût $\delta(s,v)$ d'un chemin de coût minimum reliant $s \in S$ à $v \in S$ ainsi que la *séquence* d'arcs constituant ce chemin (il peut y avoir plusieurs tels chemins) ;
- $\delta(s,v) := +\infty$ si aucun chemin ne relie s à v ;
- $\delta(s,v) := -\infty$ si un chemin relie s à v mais qu'aucun chemin n'est de coût minimum (comment est-ce possible ?)
- un *plus court chemin* de s à v est un chemin de coût $\delta(s,v)$.

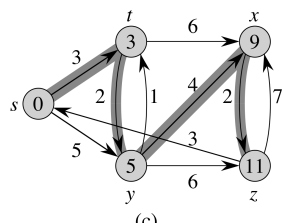
Illustration



(a)



(b)



(c)

Sous-structure optimale

Les plus courts chemins possèdent la propriété qu'ils sont composés de plus courts chemins.

Sous-structure optimale

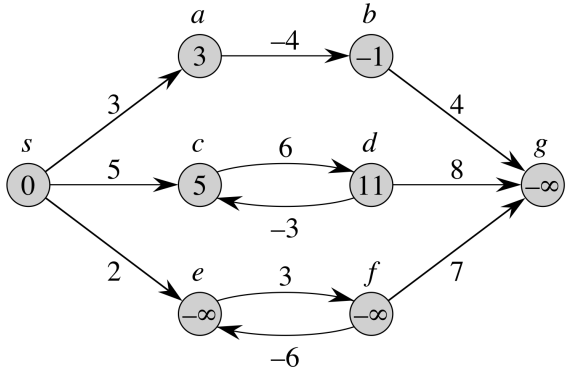
Les plus courts chemins possèdent la propriété qu'ils sont composés de plus courts chemins.

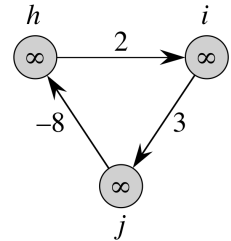
Lemme (Sous-structure optimale)

Soit $G = (S,A)$ un graphe orienté pondéré de fonction coût $c : A \to \mathbb{R}$. Soit $p = (v_0,v_1)(v_1,v_2)\cdots(v_{k-1},v_k)$ un plus court chemin $v_0 \rightsquigarrow v_k$. Pour tous $0 \leq i \leq j \leq k$, le sous-chemin $p_{ij} = (v_i,v_{i+1})\cdots(v_{j-1},v_j)$ est un plus court chemin $v_i \rightsquigarrow v_j$.

Cycles

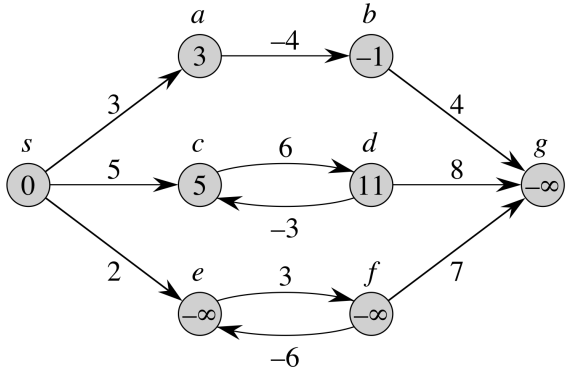
- Cycles de coût négatif :

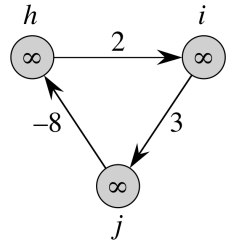




Cycles

- Cycles de coût négatif :

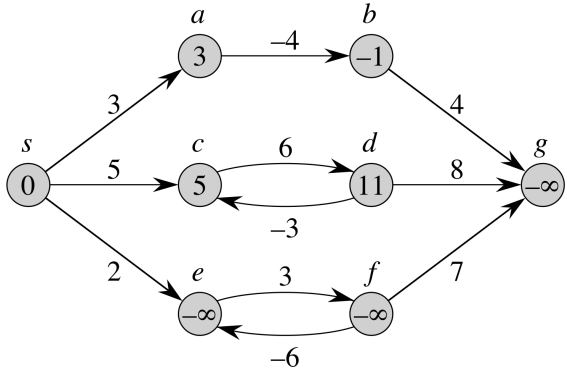


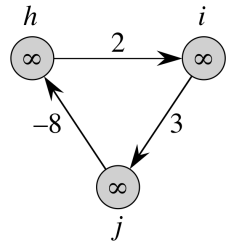


- Cycles de coût positif : inutiles ;

Cycles

- Cycles de coût négatif :

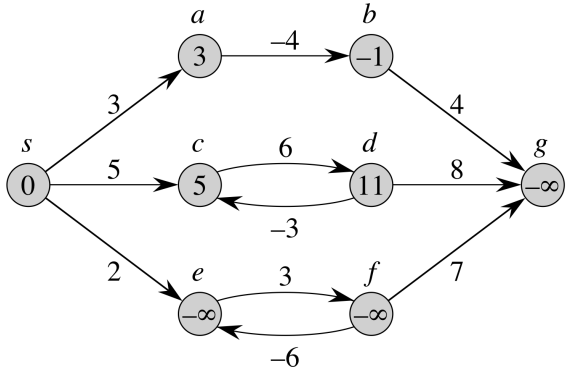


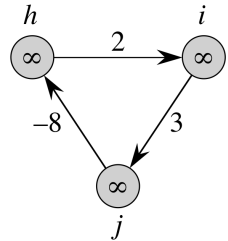


- Cycles de coût positif : inutiles ;
- Cycles de coût nul : peuvent être retirés du chemin.

Cycles

- Cycles de coût négatif :





- Cycles de coût positif : inutiles ;
- Cycles de coût nul : peuvent être retirés du chemin.

Sans perte de généralité, un plus court chemin ne contient pas de cycle. Il contient donc au plus $n - 1$ arcs.

Observations préliminaires

Lemme (Inégalité du triangle)

$\delta(s,v) \leq \delta(s,u) + c(u,v)$ pour tout arc $(u,v) \in A$.

Menu du jour

Algorithme de Dijkstra

Dijkstra : /'dɛɪkstra/

Algorithme de Dijkstra

Dijkstra : /'dɛɪkstra/

On suppose dans cette section que $c(a) \geq 0$ pour tout arc $a \in A$.

Algorithme de Dijkstra

Dijkstra : /'dɛikstra/

On suppose dans cette section que $c(a) \geq 0$ pour tout arc $a \in A$.

L'algorithme initialise des nœuds de type **`MarkedNode`**, possédant un attribut **`distance`** (initialisé à $+\infty$) et un attribut **`parent`** (initialisé à **`nothing`**). L'attribut **`distance`** de la source est initialisé à zéro.

- Un ensemble de sommets P contient à chaque itération les sommets dont le coût d'un plus court chemin depuis s a été déterminé ;
- on sélectionne le sommet de $S \setminus P$ ayant la plus petite estimation de la distance ;
- on ajoute ce sommet à P ;
- on met à jour les estimations de distance des sommets restants.

Relaxation

À chaque itération, l'attribut **`distance`** d'un nœud v donne une estimation du coût d'un plus court chemin $s \rightsquigarrow v$.

Relaxation

À chaque itération, l'attribut **`distance`** d'un nœud v donne une estimation du coût d'un plus court chemin $s \rightsquigarrow v$.

La relaxation d'un arc (u,v) consiste à vérifier si on peut améliorer cette estimation en passant par u. Si oui, elle consiste aussi à mettre à jour les attributs **`distance`** et **`parent`** de v.

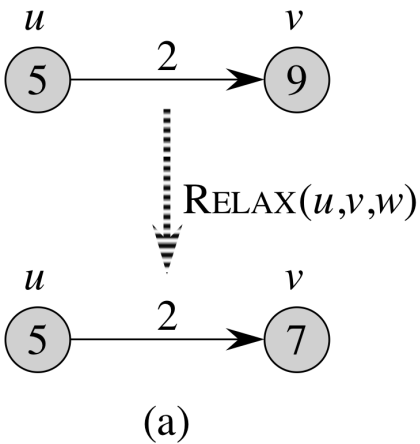
Relaxation

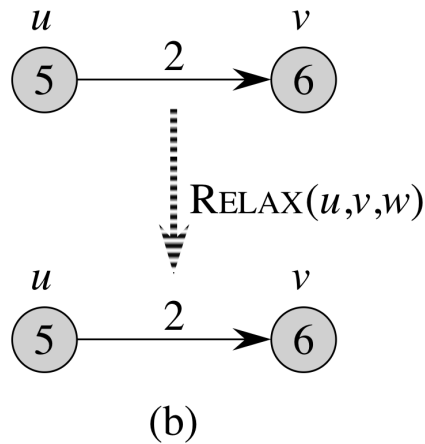
À chaque itération, l'attribut **`distance`** d'un nœud v donne une estimation du coût d'un plus court chemin $s \rightsquigarrow v$.

La relaxation d'un arc (u,v) consiste à vérifier si on peut améliorer cette estimation en passant par u. Si oui, elle consiste aussi à mettre à jour les attributs **`distance`** et **`parent`** de v.

```
"Relaxation de l'arc/arete e."
function relax(e)
    u = origin(e)
    v = destination(e)
    c = weight(e)
    ud = distance(u)

    if ud + c < distance(v)
        set_distance!(v, ud + c)
        set_parent!(v, u)
    end
end
```

Relaxation : illustration

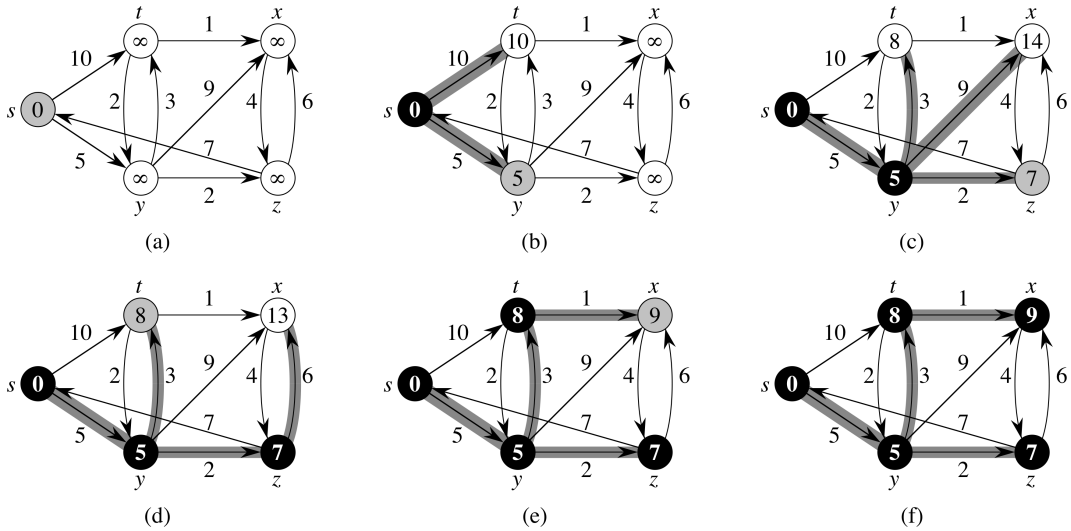




Pseudo-code

1. Choisir un sommet source s
2. initialiser **`s.distance = 0`**, **`u.distance =`** $+\infty$ pour **`u`** $\neq$ **`s`**
3. initialiser **`u.parent = nothing`** pour tout **`u`**
4. initialiser $P = \varnothing$
5. initialiser une file de priorité min qui contient S
6. tant que la file n'est pas vide
   - 6.1 extraire un élément u de priorité min de la file
   - 6.2 ajouter u à P
   - 6.3 pour chaque voisin v de u, relaxer (u,v)

Dijkstra : illustration



Convergence

On voit que l'algorithme de Dijkstra se termine en n itérations.

Convergence

On voit que l'algorithme de Dijkstra se termine en n itérations.

Théorème (Dijkstra)

L'algorithme de Dijkstra sur un graphe $G = (S,A)$ pondéré de fonction coût $c \geq 0$ et de source s se termine et l'attribut **`distance`** de chaque nœud v est égal à $\delta(s,v)$.

Convergence

On voit que l'algorithme de Dijkstra se termine en n itérations.

Théorème (Dijkstra)

L'algorithme de Dijkstra sur un graphe $G = (S,A)$ pondéré de fonction coût $c \geq 0$ et de source s se termine et l'attribut **`distance`** de chaque nœud v est égal à $\delta(s,v)$.

Exercice utile à la preuve du théorème de Dijkstra :

Lemme (Borne supérieure)

À chaque itération, l'attribut **`distance`** de tout nœud v est supérieur ou égal à $\delta(s,v)$. Une fois que la valeur $\delta(s,v)$ est atteinte, l'attribut **`distance`** ne change plus.

Complexité : analyse agrégée

Comme dans les explorations en largeur et en profondeur d'arbord, on voit facilement que chaque arc du graphe est examiné une et une seule fois en parcourant les listes d'adjacence.

Complexité : analyse agrégée

Comme dans les explorations en largeur et en profondeur d'arbord, on voit facilement que chaque arc du graphe est examiné une et une seule fois en parcourant les listes d'adjacence.

Le coût total de parcours des listes d'adjacence est donc en $\Theta(|A|)$.

Complexité : analyse agrégée

Comme dans les explorations en largeur et en profondeur d'arbord, on voit facilement que chaque arc du graphe est examiné une et une seule fois en parcourant les listes d'adjacence.

Le coût total de parcours des listes d'adjacence est donc en $\Theta(|A|)$.

Le coût des opérations de file dépend de l'implémentation de la file de priorité.

Complexité : analyse agrégée

Comme dans les explorations en largeur et en profondeur d'arbord, on voit facilement que chaque arc du graphe est examiné une et une seule fois en parcourant les listes d'adjacence.

Le coût total de parcours des listes d'adjacence est donc en $\Theta(|A|)$.

Le coût des opérations de file dépend de l'implémentation de la file de priorité.

En exercice, réfléchir à son implémentation de la file de priorité et la complexité de ses opérations **`push!()`** et **`popfirst!()`**.

Exercices

1. L'algorithme de Dijkstra est très similaire à celui de Prim. Que devez-vous changer dans votre implémentation de l'algorithme de Prim ?
2. quelle étape de la preuve du théorème de Dijkstra est erronée lorsqu'il existe des poids négatifs ?
3. supposons que dans le graphe orienté $G = (S,A)$ et pondéré, seuls les arcs sortant du nœud source s peuvent avoir un coût négatif. L'algorithme de Dijkstra produit-il le résutat correct dans ce cas ? Expliquer.
4. Quel est le lien entre l'algorithme de Dijkstra et l'algorithme d'exploration en largeur d'abord ?

Menu du jour

Algorithme de Bellman-Ford

Lorsqu'il existe des arcs de poids négatif, l'algorithme de Bellman-Ford

- signale si il existe un cycle de coût négatif (dans ce cas, les attributs **`distance`** ne sont pas nécessairement égaux aux $\delta(s,v)$) ;
- donne tous les poids des plus courts chemins $\delta(s,v)$ pour une source s et tous les nœuds v s'il n'y a pas de cycle de coût négatif.

Algorithme de Bellman-Ford

Lorsqu'il existe des arcs de poids négatif, l'algorithme de Bellman-Ford

- signale si il existe un cycle de coût négatif (dans ce cas, les attributs **`distance`** ne sont pas nécessairement égaux aux $\delta(s,v)$) ;
- donne tous les poids des plus courts chemins $\delta(s,v)$ pour une source s et tous les nœuds v s'il n'y a pas de cycle de coût négatif.

Idée : relaxer *tous* les arcs

Algorithme de Bellman-Ford

Lorsqu'il existe des arcs de poids négatif, l'algorithme de Bellman-Ford

- signale si il existe un cycle de coût négatif (dans ce cas, les attributs **`distance`** ne sont pas nécessairement égaux aux $\delta(s,v)$) ;
- donne tous les poids des plus courts chemins $\delta(s,v)$ pour une source s et tous les nœuds v s'il n'y a pas de cycle de coût négatif.

Idée : relaxer *tous* les arcs $(n-1)$ fois.

Algorithme de Bellman-Ford

Lorsqu'il existe des arcs de poids négatif, l'algorithme de Bellman-Ford

- signale si il existe un cycle de coût négatif (dans ce cas, les attributs **`distance`** ne sont pas nécessairement égaux aux $\delta(s,v)$) ;
- donne tous les poids des plus courts chemins $\delta(s,v)$ pour une source s et tous les nœuds v s'il n'y a pas de cycle de coût négatif.

Idée : relaxer *tous* les arcs $(n-1)$ fois.

Le concept est que ce nombre de passages est suffisant pour déterminer les $\delta(s,v)$ ou l'existence d'un cycle de poids négatif.

Implémentation

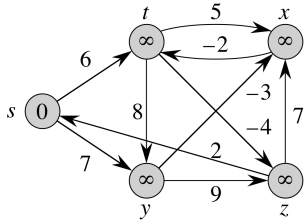
```
include("relax.jl")

function bellman_ford(G, s)
  set_distance!(s, 0.0)

  for i = 1 : nb_nodes(G) - 1
    for edge ∈ edges(G)  # toutes les arêtes sans duplication
      relax(edge)
    end
  end

  for edge ∈ edges(G)  # vérification d'existence d'un cycle de coût < 0
    if distance(to(edge)) > distance(from(edge)) + weight(edge)
      @warn "Bellman-Ford: negative cost cycle detected"
      return false
    end
  end
  return true
end
```

Implémentation

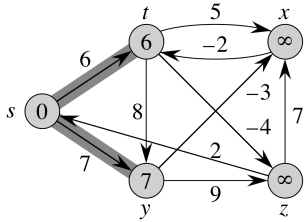
```
include("relax.jl")

function bellman_ford(G, s)
  set_distance!(s, 0.0)

  for i = 1 : nb_nodes(G) - 1
    for edge ∈ edges(G)  # toutes les arêtes sans duplication
      relax(edge)
    end
  end

  for edge ∈ edges(G)  # vérification d'existence d'un cycle de coût < 0
    if distance(to(edge)) > distance(from(edge)) + weight(edge)
      @warn "Bellman-Ford: negative cost cycle detected"
      return false
    end
  end
  return true
end
```

Complexité : $\Theta(|S| \cdot |A|) + O(|A|) = \Theta(|S| \cdot |A|)$.

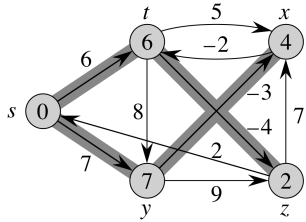
Illustration



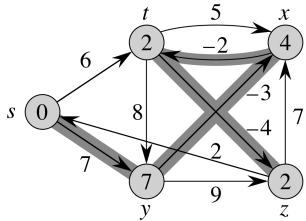
(a)



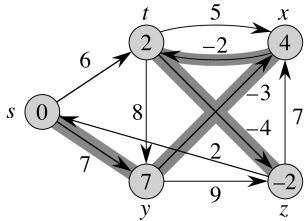
(b)



(c)



(d)



(e)

Convergence

Soit $G = (S,A)$ un graphe orienté pondéré de nœud source s et de fonction coût $c : A \to \mathbb{R}$.

Lemme

Si G ne contient aucun cycle de poids négatif joignable depuis s, après $|S| - 1$ itérations, l'attribut **`distance`** de chaque nœud v a la valeur $\delta(s,v)$.

Convergence

Soit $G = (S,A)$ un graphe orienté pondéré de nœud source s et de fonction coût $c : A \to \mathbb{R}$.

Lemme

Si G ne contient aucun cycle de poids négatif joignable depuis s, après $|S| - 1$ itérations, l'attribut **`distance`** de chaque nœud v a la valeur $\delta(s,v)$.

Théorème (Bellman-Ford)

Si G ne contient aucun cycle de poids négatif joignable depuis s, l'algorithme renvoie la valeur **`true`** et l'attribut **`distance`** de chaque nœud v a la valeur $\delta(s,v)$. S'il existe un cycle de poids négatif joignable depuis s, l'algorithme renvoie la valeur **`false`**.

Exercices

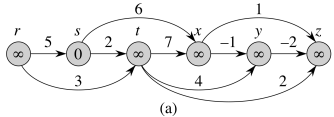
1. Si G ne contient aucun cycle de poids négatif joignable depuis s, montrer qu'après exécution de l'algorithme de Bellman-Ford, il existe un chemin $s \rightsquigarrow v$ si et seulement si l'attribut **`distance`** de v est fini ;

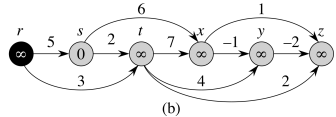
Cas particulier : graphes orientés acycliques

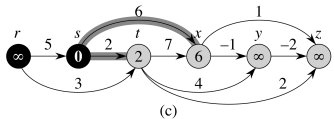
Ici il ne peut certainement exister aucun cycle de poids négatif (mais certains poids peuvent être négatifs). L'idée de l'algorithme est :

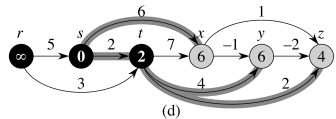
1. Effectuer un tri topologique sur le graphe ;
2. initialiser l'attribut **`distance`** de la source à zéro ;
3. parcourir les sommets dans l'ordre du tri topologique et relaxer chaque arc sortant.

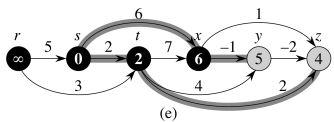
Illustration

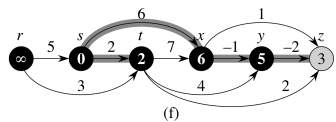


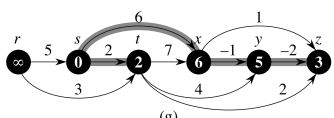












Menu du jour

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois.

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois. Coût :

$$O(|S| \cdot (|S| \cdot \text{coût(file)} + |A|)) \approx O(|S|^3 + |S| \cdot |A|)$$

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois. Coût :

$$O(|S| \cdot (|S| \cdot \text{coût(file)} + |A|)) \approx O(|S|^3 + |S| \cdot |A|)$$

- appliquer l'algorithme de Bellman-Ford $|S|$ fois.

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois. Coût :

$$O(|S| \cdot (|S| \cdot \text{coût(file)} + |A|)) \approx O(|S|^3 + |S| \cdot |A|)$$

- appliquer l'algorithme de Bellman-Ford $|S|$ fois. Coût :

$$O(|S|^2 \cdot |A|).$$

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois. Coût :

$$O(|S| \cdot (|S| \cdot \text{coût(file)} + |A|)) \approx O(|S|^3 + |S| \cdot |A|)$$

- appliquer l'algorithme de Bellman-Ford $|S|$ fois. Coût :

$$O(|S|^2 \cdot |A|).$$

Pour un graphe dense, ces complexités sont respectivement $O(|S|^3)$ et $O(|S|^4)$.
Peut-on faire mieux ?

Première idée

- Si $c(u,v) \geq 0$ pour tous u, $v \in S$, appliquer l'algorithme de Dijkstra $|S|$ fois. Coût :

$$O(|S| \cdot (|S| \cdot \text{coût(file)} + |A|)) \approx O(|S|^3 + |S| \cdot |A|)$$

- appliquer l'algorithme de Bellman-Ford $|S|$ fois. Coût :

$$O(|S|^2 \cdot |A|).$$

Pour un graphe dense, ces complexités sont respectivement $O(|S|^3)$ et $O(|S|^4)$.
Peut-on faire mieux ?

L'algorithme de Floyd-Warshall travaille avec des matrices d'adjacence et a une complexité $\Theta(|S|^3)$. Nous n'en parlons pas car il n'utilise aucune structure de données particulières et nous concentrons sur une variante qui utilise les listes d'adjacence.

Algorithme de Johnson

Idée :

1. Si $c \geq 0$, l'algorithme de Johnson applique l'algorithme de Dijkstra depuis chaque sommet ;

Algorithme de Johnson

Idée :

1. Si $c \geq 0$, l'algorithme de Johnson applique l'algorithme de Dijkstra depuis chaque sommet ;
2. sinon, on modifie les coûts des arcs de sorte que les nouveaux coûts $\hat{c}(u,v)$ vérifient :

Algorithme de Johnson

Idée :

1. Si $c \geq 0$, l'algorithme de Johnson applique l'algorithme de Dijkstra depuis chaque sommet ;
2. sinon, on modifie les coûts des arcs de sorte que les nouveaux coûts $\hat{c}(u,v)$ vérifient :
   2.1 p est un chemin de moindre coût de u à v pour les coûts c si et seulement si p est un chemin de moindre coût de u à v pour les coûts $\hat{c}$;

Algorithme de Johnson

Idée :

1. Si $c \geq 0$, l'algorithme de Johnson applique l'algorithme de Dijkstra depuis chaque sommet ;
2. sinon, on modifie les coûts des arcs de sorte que les nouveaux coûts $\hat{c}(u,v)$ vérifient :
   - 2.1 p est un chemin de moindre coût de u à v pour les coûts c si et seulement si p est un chemin de moindre coût de u à v pour les coûts $\hat{c}$;
   - 2.2 $\hat{c} \geq 0$.

Changement de fonction coût

Soit $G = (S,A)$ notre graphe orienté et $c : A \to \mathbb{R}$ notre fonction coût. Considérons une fonction quelconque $h : S \to \mathbb{R}$. Pour tous sommets u, $v \in S$, on définit

$$\hat{c}(u,v) := c(u,v) + h(u) - h(v).$$

Changement de fonction coût

Soit $G = (S,A)$ notre graphe orienté et $c : A \to \mathbb{R}$ notre fonction coût. Considérons une fonction quelconque $h : S \to \mathbb{R}$. Pour tous sommets u, $v \in S$, on définit

$$\hat{c}(u,v) := c(u,v) + h(u) - h(v).$$

Lemme

Soit $p : u \rightsquigarrow v$. Alors $\hat{c}(p) = c(p) + h(u) - h(v)$ et

1. p est un chemin de moindre coût pour la fonction de coût c si et seulement si p est un chemin de moindre coût pour la fonction $\hat{c}$, i.e.,

$$c(p) = \delta(u,v) \quad \iff \quad \hat{c}(p) = \hat{\delta}(u,v)$$

2. G contient un cycle de coût négatif pour c si et seulement si G contient un cycle de coût négatif pour $\hat{c}$.

Comment assurer que $\hat{c} \geq 0$?

Étant donné $G = (S,A)$, on construit un nouveau graphe $G' = (S',A')$ en introduisant un nouveau sommet : $S' := S \cup \{s\}$ où $s \notin S$.

Comment assurer que $\hat{c} \geq 0$?

Étant donné $G = (S,A)$, on construit un nouveau graphe $G' = (S',A')$ en introduisant un nouveau sommet : $S' := S \cup \{s\}$ où $s \notin S$.

On définit les arcs

$$A' := A \cup \{(s,v) \mid v \in S\}\,.$$

Comment assurer que $\hat{c} \geq 0$?

Étant donné $G = (S,A)$, on construit un nouveau graphe $G' = (S',A')$ en introduisant un nouveau sommet : $S' := S \cup \{s\}$ où $s \notin S$.

On définit les arcs

$$A' := A \cup \{(s,v) \mid v \in S\}\,.$$

Remarque : $\Gamma^-(s) = \emptyset$.

Comment assurer que $\hat{c} \geq 0$?

Étant donné $G = (S,A)$, on construit un nouveau graphe $G' = (S',A')$ en introduisant un nouveau sommet : $S' := S \cup \{s\}$ où $s \notin S$.

On définit les arcs

$$A' := A \cup \{(s,v) \mid v \in S\}\,.$$

Remarque : $\Gamma^-(s) = \emptyset$.

On définit $c(s,v) := 0$ pour tout $v \in S$.

Comment assurer que $\hat{c} \geq 0$?

Étant donné $G = (S,A)$, on construit un nouveau graphe $G' = (S',A')$ en introduisant un nouveau sommet : $S' := S \cup \{s\}$ où $s \notin S$.

On définit les arcs

$$A' := A \cup \{(s,v) \mid v \in S\}.$$

Remarque : $\Gamma^-(s) = \emptyset$.

On définit $c(s,v) := 0$ pour tout $v \in S$.

Supposons que G ne contient aucun cycle de coût négatif. On pose

$$h(v) := \delta(s,v) \quad v \in S'.$$

Algorithme de Johnson

```
function johnson(G::AbstractGraph{T}) where T
    s = MarkedNode(data(nodes(G)[1]), name="fictif", distance=0.0)
    add_node!(G, s)
    for node ∈ nodes(G)
        add_edge!(G, Edge(s, node))  # weight = 0 par défaut
    end

    D = Dict{Node{T}, Dict{Node{T}, Float64}}()
    bellman_ford(G, s) || (return D)

    h = Dict{Node{T}, Float64}()  # définition de la nouvelle fonction de coût
    for node ∈ nodes(G)
        node === s && continue
        h[node] = distance(node)
        set_distance!(node, Inf)  # réinitialisation
    end

    for edge ∈ edges(G)
        set_weight!(edge, weight(edge) + h[from(edge)] - h[to(edge)])
    end
```

Algorithme de Johnson

```
    for node ∈ nodes(G)
        node === s && continue
        dijkstra(G, node)  # calcul des PCCs à partir de node
        D[node] = Dict{Node{T}, Float64}()
        for v ∈ nodes(G)
            v === s && continue
            if distance(v) < Inf
                D[node][v] = distance(v) + h[v] - h[node]
                set_distance!(v, Inf)  # réinitialisation
            end
        end
    end
    return D
end
```

Exercices

1. Sur base de **`dijkstra()`**, modifier **`johnson()`** pour donner les chemins de moindre coût et non pas seulement leurs coûts ;
2. quel est le lien entre c et $\hat{c}$ lorsque $c \geq 0$?
3. quelle est la complexité de la méthode de Johnson ?
4. modifier la classe **`Graph`** pour qu'on puisse retirer un nœud ainsi que toutes les arêtes / tous les arcs qui lui sont adjacents.