# Algorithmique des graphes

## Cours 7 – Calcul de distances

František Kardoš

frantisek.kardos@u-bordeaux.fr

# DS

jeudi le 10 novembre à 11h

groupes Info A2, A3, A4 : A29/Amphis B et D
groupes Info A1, A5, MathInfo, CMI : A9/Amphi 2

# Calcul de distances

Entrée : Un graphe (orienté ou pas) $G$, avec une pondération des arêtes

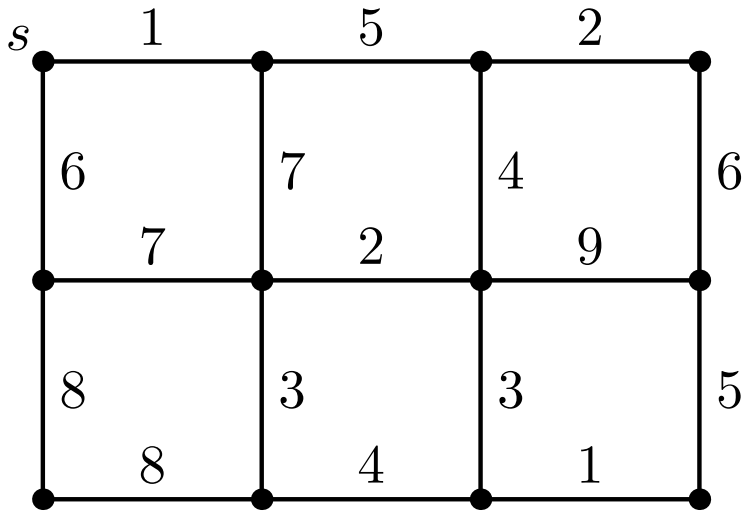
$$w : E(G) \to \mathbb{R}^+;$$

un sommet de départ $s$
Sortie : La distance $dist(s, v)$ pour tout sommet $v$

Rappel : la distance entre deux sommets est le poids minimum d'une chaîne (d'un chemin) les reliant.

# Calcul de distances

Entrée : Graphe $G$ avec les arêtes pondérées ; sommet de départ $s$



# Quelques observations

Pour accéder à un sommet $u$ depuis $s$, plusieurs chaînes/chemins optimaux peuvent exister.

# Quelques observations

Pour accéder à un sommet $u$ depuis $s$, plusieurs chaînes/chemins optimaux peuvent exister.

La chaîne optimale (le chemin optimal) n'est pas forcément celle (celui) d'un plus petit nombre d'arêtes (arcs).

# Quelques observations

Pour accéder à un sommet $u$ depuis $s$, plusieurs chaînes/chemins optimaux peuvent exister.

La chaîne optimale (le chemin optimal) n'est pas forcément celle (celui) d'un plus petit nombre d'arêtes (arcs).

Si le prédécesseur de $u$ dans une chaîne optimale (un chemin optimal) est un voisin (entrant) $v$ de $u$, alors la sous-chaîne (le sous-chemin) de $s$ à $v$ est optimal.e aussi.

# Quelques observations

Pour accéder à un sommet $u$ depuis $s$, plusieurs chaînes/chemins optimaux peuvent exister.

La chaîne optimale (le chemin optimal) n'est pas forcément celle (celui) d'un plus petit nombre d'arêtes (arcs).

Si le prédécesseur de $u$ dans une chaîne optimale (un chemin optimal) est un voisin (entrant) $v$ de $u$, alors la sous-chaîne (le sous-chemin) de $s$ à $v$ est optimal.e aussi.

On a donc

$$dist(s,u) = \min_{vu \in E(G)} \{dist(s,v) + w(vu)\}.$$

# Algorithme de Dijkstra

Les sommets sont visités dans l'ordre croissant de leur distance à partir du sommet de départ $s$.
A chaque étape, la distance d'un sommet est marquée définitive (le sommet est colorié noir) ;
Les distances d'autres sommets sont éventuellement mises à jour.

Si pour un sommet $u$ la valeur de $d[u]$ est différente de $\infty$, alors il existe au moins une chaîne de $s$ à $u$ ; parmi les chaînes reliant $s$ et $u$, dans celle de longueur (poids total) $d[u]$ le prédécesseur de $u$ est mémorisé comme $pere[u]$.

# Algorithme de Dijkstra

Initialisation :

1: **for all** $v$ de $V(G)$ **do**
2: $\quad d(v) \leftarrow \infty$
3: $\quad pere(v) \leftarrow NIL$
4: $\quad couleur(v) \leftarrow BLANC$
5: **end for**
6: $d(s) \leftarrow 0$
7: $F \leftarrow FILE\_PRIORITE(\{s\}, d)$

# Algorithme de Dijkstra

Propagation :

```
 8: while F ≠ ∅ do
 9:     pivot ← EXTRAIRE_MIN(F)
10:     for all e = (pivot, v) arc sortant de pivot do
11:         if couleur(v) = BLANC then
12:             if d(v) = ∞ then
13:                 INSERER(F, v)
14:             end if
15:             if d[v] > d[pivot] + w(e) then
16:                 d[v] ← d[pivot] + w(e)
17:                 pere[v] ← pivot
18:             end if
19:         end if
20:     end for
21:     couleur[pivot] ← NOIR
22: end while
```

# Limitations de l'algorithme de Dijkstra

L'algorithme de Dijkstra marche bien si les poids sont positifs.

Dans un graphe orienté contenant des arcs de poids négatif l'algorithme peut échouer : Il déclare la distance vers un sommet comme définitive trop tôt.

# Algorithme de Bellman : Graphes orientés sans circuit

Principe : Considérer les sommets dans un tel ordre que s'il y a un arc $u \to v$, alors le sommet $v$ est considéré plus tard que le sommet $u$.

L'arc $uv$ est découvert lors de la visite du sommet $u$. Si cet arc permet d'accéder à $v$ par un chemin plus court que ce qu'on connaît pour l'instant, l'information sur la distance et le prédécesseur de sommet $v$ est mise à jour.

La distance d'un sommet est définitive si on a déjà considéré tous ses voisins entrants.

# Algorithme de Bellman : Graphes orientés sans circuit

1: **for all** $v$ de $V(G)$ **do**
2: $\quad d[v] \leftarrow \infty$
3: $\quad pere[v] \leftarrow NIL$
4: $\quad npred[v] \leftarrow deg^{-}[v]$
5: **end for**
6: $d[s] \leftarrow 0$
7: INSERER_FILE(F, s)

# Algorithme de Bellman : Graphes orientés sans circuit

```
 8: while F non vide do
 9:     u ← TETE_FILE(F)
10:     DEFILER(F)
11:     for v ∈ Adj(u) do
12:         if d[v] > d[u] + w(u, v) then
13:             d[v] ← d[u] + w[u, v]
14:             pere[v] ← u
15:         end if
16:         npred(v) ← nped[v] − 1
17:         if npred(v) = 0 then
18:             INSERER_FILE(F, v)
19:         end if
20:     end for
21: end while
```

# Algorithme de Ford : Graphes orientés avec circuits

L'algorithme calcule en $n - 1$ étapes les distances à partir d'un sommet de départ $s$.

Après l'étape numéro $i$, pour tout sommet $v$ la longueur du plus court chemin de $s$ à $v$ composé d'au plus $i$ arcs est calculée.

Comment faire ?

# Algorithme de Ford : Graphes orientés avec circuits

L'algorithme calcule en $n - 1$ étapes les distances à partir d'un sommet de départ $s$.

Après l'étape numéro $i$, pour tout sommet $v$ la longueur du plus court chemin de $s$ à $v$ composé d'au plus $i$ arcs est calculée.

Comment faire ? À chaque étape, tout arc $uv$ du graphe est inspecté, s'il ne peut pas être le dernier arc d'un chemin de $s$ à $v$.

# Algorithme de Ford : Graphes orientés avec circuits

1: **for all** $v$ de $V(G)$ **do**
2: $\quad d[v] \leftarrow \infty$
3: $\quad pere[v] \leftarrow NIL$
4: **end for**
5: $d[s] \leftarrow 0$

# Algorithme de Ford : Graphes orientés avec circuits

```
5:  for i de 1 à n − 1 do
6:      for tout arc e = (u, v) ∈ E(G) do
7:          if d[v] > d[u] + w(u, v) then
8:              d[v] ← d[u] + w[u, v]
9:              pere[v] ← u
10:         end if
11:     end for
12: end for
```

# Algorithme de Ford : Détection de circuit absorbant

13: **for** tout arc $e = (u, v) \in E(G)$ **do**
14: **if** $d[v] > d[u] + w(u, v)$ **then**
15: **return** FAUX
16: **end if**
17: **end for**
18: **return** VRAI

# Algorithme de Floyd

permet de calculer les distances pour toutes paires de sommets en même temps.
Après l'exécution de l'algorithme,

- la matrice $W_{i,j}$ contiendra la plus courte distance entre le sommet $i$ et le sommet $j$ ;
- la matrice $Pred_{i,j}$ le sommet prédécesseur de $j$ sur un plus court chemin de $i$ à $j$.

# Algorithme de Floyd

```
1:  for i de 1 à n do
2:      for j de 1 à n do
3:          if i = j then
4:              W(i, j) ← 0
5:              Pred(i, j) ← i
6:          else if ij ∈ E(G) then
7:              W(i, j) ← w(i, j)
8:              Pred(i, j) ← i
9:          else
10:             W(i, j) ← ∞
11:             Pred(i, j) ← NIL
12:         end if
13:     end for
14: end for
```

# Algorithme de Floyd

15: **for** $k$ de 1 à $n$ **do**
16: **for** $i$ de 1 à $n$ **do**
17: **for** $j$ de 1 à $n$ **do**
18: **if** $W(i,k) + W(k,j) < W(i,j)$ **then**
19: $W(i,j) \leftarrow W(i,k) + W(k,j)$
20: $Pred(i,j) \leftarrow Pred(k,j)$
21: **end if**
22: **end for**
23: **end for**
24: **end for**

# Algorithme de Floyd : Détection de circuit absorbant

Comment détecter la présence d'un circuit absorbant à la fin de l'exécution de l'algorithme de Floyd ?