Apect algorithmiques des graphes

A.Belcaid

ENSA-Safi

April 25, 2022

Ce chapitre présente un ensemble de **problèmes classiques** dans la théorie de graphes.

Ces problèmes couvrent:

- Algorithme de **plus court chemin** entre deux noeuds.

Introduction

Ce chapitre présente un ensemble de **problèmes classiques** dans la théorie de graphes.

Ces problèmes couvrent:

- Algorithme de **plus court chemin** entre deux noeuds.
- Algorithme d'**arbre minimal couvrant**.

Ce chapitre présente un ensemble de **problèmes classiques** dans la théorie de graphes.

Ces problèmes couvrent:

- Algorithme de **plus court chemin** entre deux noeuds.
- Algorithme d'**arbre minimal couvrant**.
- Problème de **flot maximal**.

Plus cours chemin

- Une grande classes de problèmes en **RO!** (**RO!**) peuvent être modélises en utilisant des **graphes ponderes**. Les problèmes de cheminement dans les graphes, en particulier la recherche du **plus court chemin**, compte parmi les plus anciens.

Plus cours chemin

- Une grande classes de problèmes en **RO!** peuvent être modélises en utilisant des **graphes ponderes**. Les problèmes de cheminement dans les graphes, en particulier la recherche du **plus court chemin**, compte parmi les plus anciens.

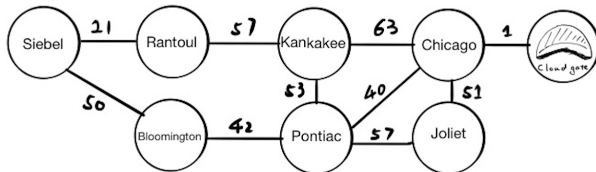


Figure: Exemple de graphe pondère, on veut chercher le chemin le plus court entre **Seibel** et **Cloud gate**

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

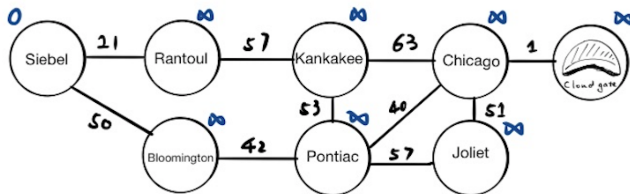


Figure: Configuration initiale de l'algorithme de Dijkstra

Algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:

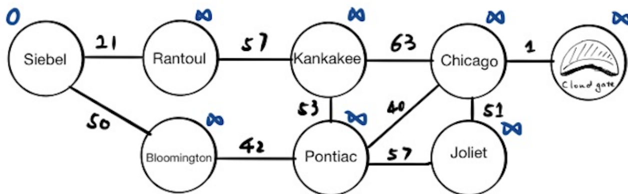


Figure: Configuration initiale de l'algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:
  1. Un noeud **source** qui est le noeud initial.

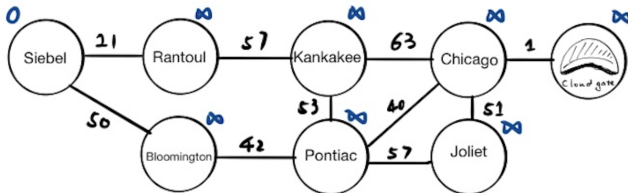


Figure: Configuration initiale de l'algorithme de Dijkstra

Algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:
  1. Un noeud **source** qui est le noeud initial.
  2. Un noeud **target** qui le noeud de notre destinations.

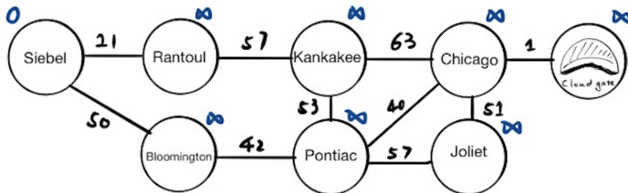


Figure: Configuration initiale de l'algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:
  1. Un noeud **source** qui est le noeud initial.
  2. Un noeud **target** qui le noeud de notre destinations.
  3. Un **ensemble**(set) qui retient l'ensemble des noeuds **visites**.

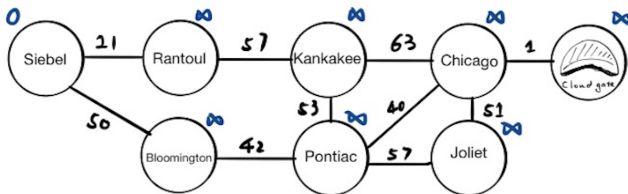


Figure: Configuration initiale de l'algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:
  1. Un noeud **source** qui est le noeud initial.
  2. Un noeud **target** qui le noeud de notre destinations.
  3. Un **ensemble**(set) qui retient l'ensemble des noeuds **visites**.
  4. Un tableau (ou **hashmap**) qui représente la distance optimale de la source.

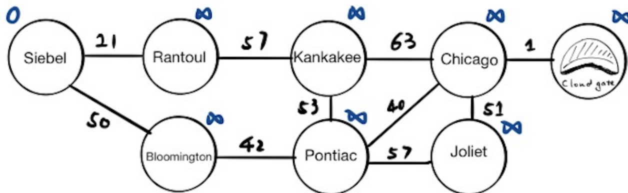


Figure: Configuration initiale de l'algorithme de Dijkstra

- L'algorithme de **Dijkstra** est un algorithme **glouton** qui calcule le plus court chemin dans un graphe pondère qui ne contient pas des **arrêtes de poids négatif**.

- Cet algorithme nécessite:
  1. Un noeud **source** qui est le noeud initial.
  2. Un noeud **target** qui le noeud de notre destinations.
  3. Un **ensemble**(set) qui retient l'ensemble des noeuds **visites**.
  4. Un tableau (ou **hashmap**) qui représente la distance optimale de la source.
  5. Une **File de priorite** qui nous donne le noeud le plus proche(glouton).

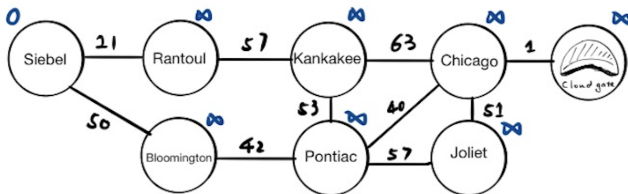
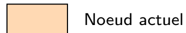


Figure: Configuration initiale de l'algorithme de Dijkstra

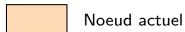
Principe Glouton de Dijkstra

- Chaque noeud doit être **visite** une seule fois.



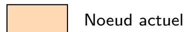
Principe Glouton de Dijkstra

- Chaque noeud doit être **visite** une seule fois.
- Pour chaque noeud visite, on **ameliore** la distance optimale de ces **noeuds incidents**.



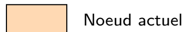
Principe Glouton de Dijkstra

- Chaque noeud doit être **visite** une seule fois.
- Pour chaque noeud visite, on **ameliore** la distance optimale de ces **noeuds incidents**.
- On tire toujours le noeud le **plus proche** dans la **file d'attente**.



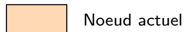
Principe Glouton de Dijkstra

- Chaque noeud doit être **visite** une seule fois.
- Pour chaque noeud visite, on **ameliore** la distance optimale de ces **noeuds incidents**.
- On tire toujours le noeud le **plus proche** dans la **file d'attente**.
- Pour illustrer ces mécanismes, on va adopter la légende suivante:

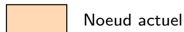


Principe Glouton de Dijkstra

- Chaque noeud doit être **visite** une seule fois.
- Pour chaque noeud visite, on **ameliore** la distance optimale de ces **noeuds incidents**.
- On tire toujours le noeud le **plus proche** dans la **file d'attente**.
- Pour illustrer ces mécanismes, on va adopter la légende suivante:



- Chaque noeud doit être **visite** une seule fois.
- Pour chaque noeud visite, on **ameliore** la distance optimale de ces **noeuds incidents**.
- On tire toujours le noeud le **plus proche** dans la **file d'attente**.
- Pour illustrer ces mécanismes, on va adopter la légende suivante:

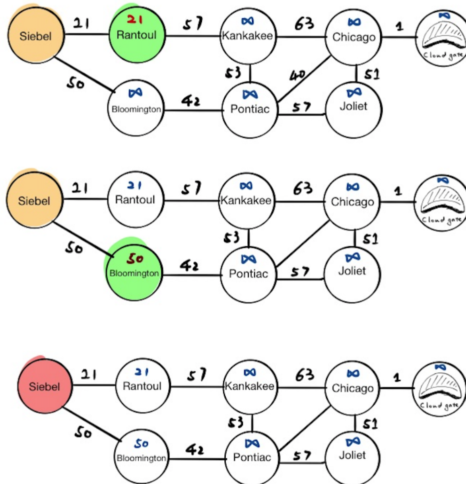


Notre file contient:

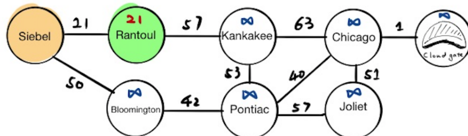
File d'attente

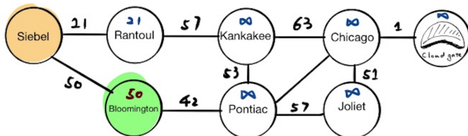
[[("Seibel", 0)]]

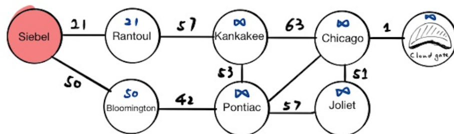
- Le premier noeud est **Seibel**, on modifie la distance de leurs voisins



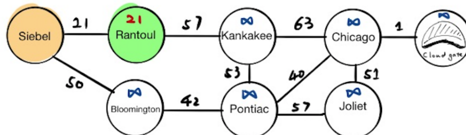
- Le premier noeud est **Seibel**, on modifie la distance de leurs voisins
  - Rantool

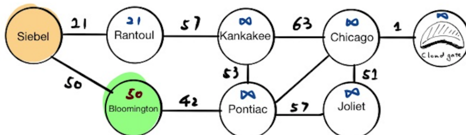


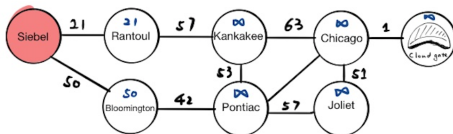




- Le premier noeud est **Seibel**, on modifie la distance de leurs voisins
  - Rantool
  - Bloomington

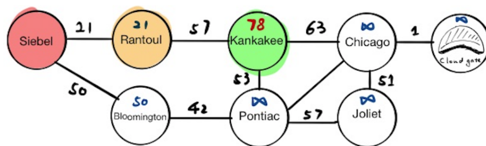






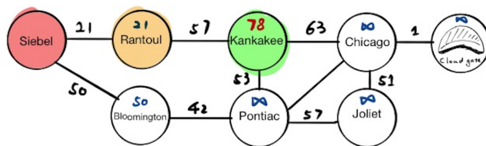
File d'attente

[("Rantoul", 21), ("Bloomington", 50)]



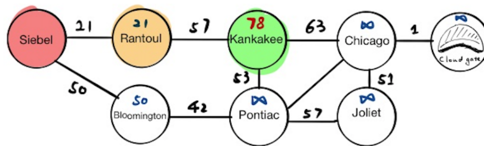
File d'attente

[("Rantoul", 21), ("Bloomington", 50)]

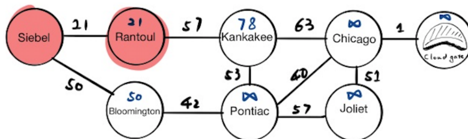


File d'attente

[("Rantoul", 21), ("Bloomington", 50)]

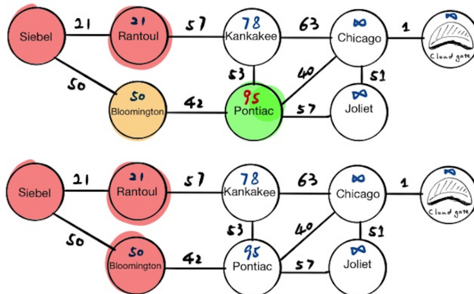


- On la marque comme **visite**:



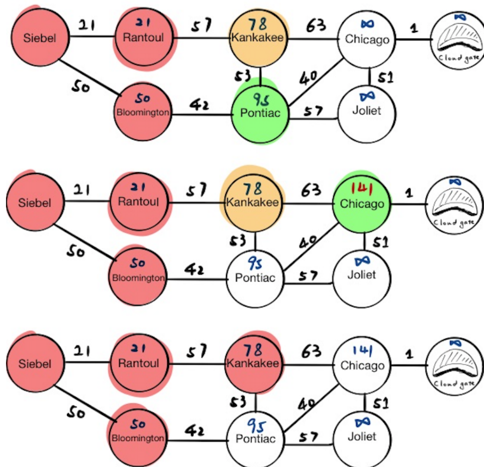
File d'attente

[("Kankakee", 78), ("Pontiac", 90)]



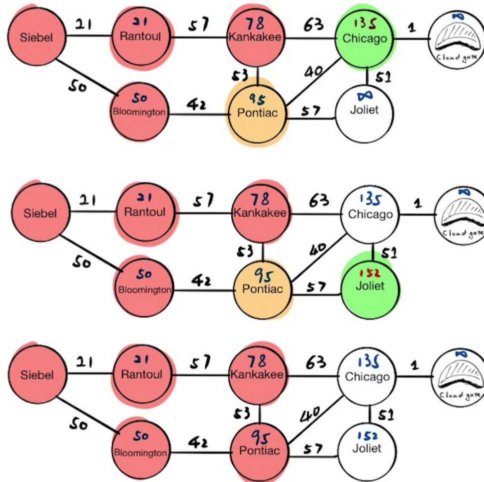
File d'attente

[("Pontiac", 90), ("Chicago", 141)]



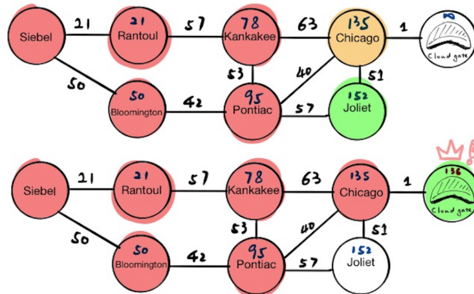
File d’attente

[(“Chicago”, 135), (“Joliet”, 152)]



File d'attente

[("Cloud Gate", 136), ("Joliet", 152)]



Algorithm Dijkstra(graph, source, destination)

```
Initialiser les distances D.
Initialiser les parents P.
Initialiser la file d'attente Q.
Initialiser les noeud visites E.
while On as pas atteint la destination do
    Obtenir le noeud le plus proche
    for noeud incident et non visite do
        if Amelioer sa distance then
            P[noeud] = current;
        end if
    end for
    ajouter au noeud visites.
end while
```

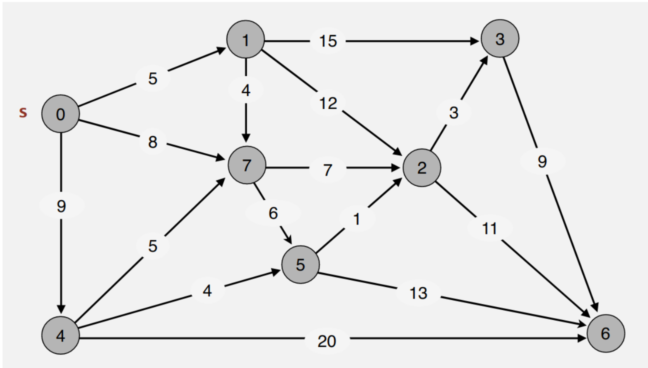


Figure: Appliquer l'algorithme de Dijkstra sur ce graphe

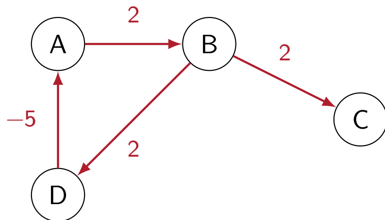


Figure: Illustration problème de cycle négatif

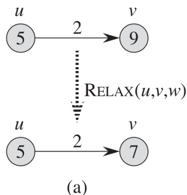
Relaxation

- L'algorithme de **Djikstra** et aussi les autres algorithmes de graphe pour le calcul de **court chemin** se base sur l'idée:

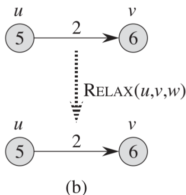
Relaxation

1: Relax(u, v, w)
2: **if** v.d $>$ u.d $+$ w(u,v) **then**
3: v.d $=$ u.d $+$ w(u,v)
4: v.π $=$ u
5: **end if**

- Ligne **3**: change la distance optimale.
- Ligne **4**: change le parent du noeud u.



(a)



(b)

Initialisation

- L'idee de relaxation necessite un:
  - Tableau des distances qu'on note d. Il represente la distance **optimale** au noeud s.
  - Tableau des **parents** qu'on note π.

Initialisation

1: Initiate-Single-Source(G, s)
2: **for** chaque noeud $v \in G.V$ **do**
3: $\quad v.d = \infty$
4: $\quad V.\pi =$ NIL
5: **end for**
6: $s.d = 0$

Bellman-Ford Algorithme

- L'Algorithme de **Bellman Ford** resout le probleme de plus court chemin dans le cas **general**.[1].
- Il renoive un **boolean** indiquant si on peut avoir un cycle de longeur negatif.
- L'algorithme **relaxe** les arretes du graphes pour tous les noeuds.

BellMan-Ford

```
1:  Bellman-Ford(G, w, s)

2:  Initiate-Single-Source(G, s)
3:  for i = 1 to |G.V| − 1 do
4:      for chaque arrete (u, v) ∈ G.E do
5:          Relax(u, v, w).
6:      end for
7:  end for

8:  for chaque arrete (u, v) ∈ G.E do
9:      if v.d > u.d + w(u, v) then
10:         return False
11:     end if
12: end for
```

[1]avec des poids negatifs

Exemple

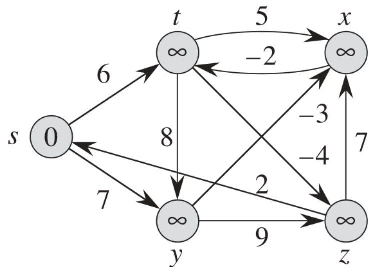


Figure: Illustration algorithme Bellman-Ford

Exemple

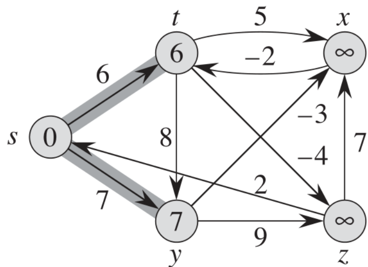


Figure: Illustration algorithme Bellman-Ford

Exemple

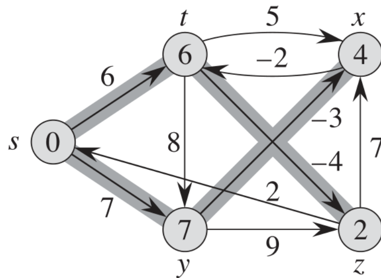


Figure: Illustration algorithme Bellman-Ford

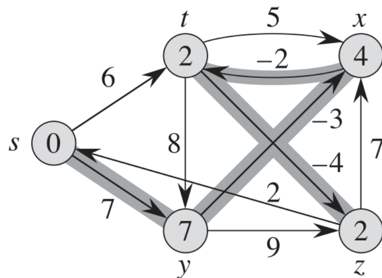


Figure: Illustration algorithme Bellman-Ford

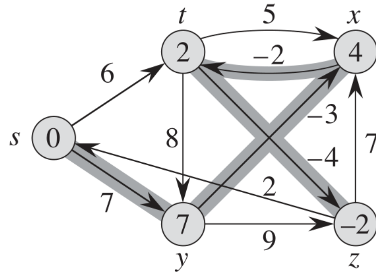


Figure: Illustration algorithme Bellman-Ford