Algorithmique et Complexité
Cours 2/7 : L'algorithme des plus courts chemins

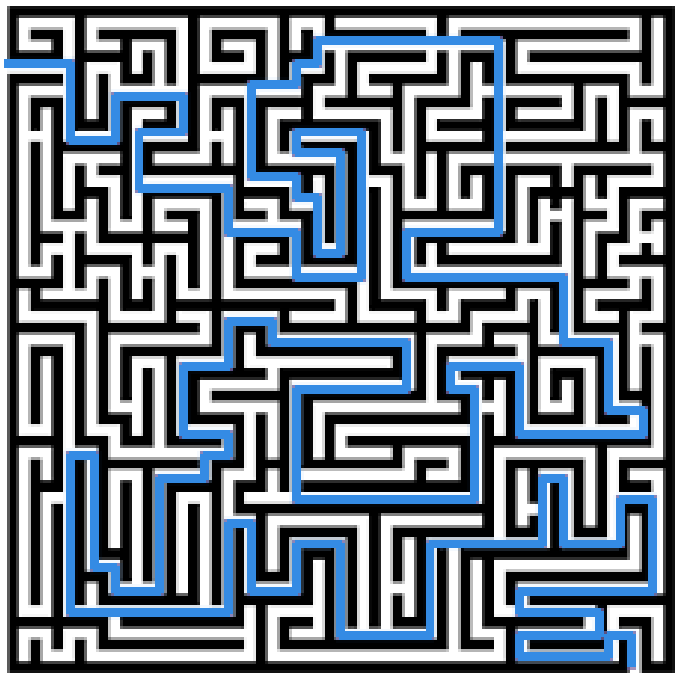
CentraleSupélec – Gif

ST2 – Gif

Plan

Plan

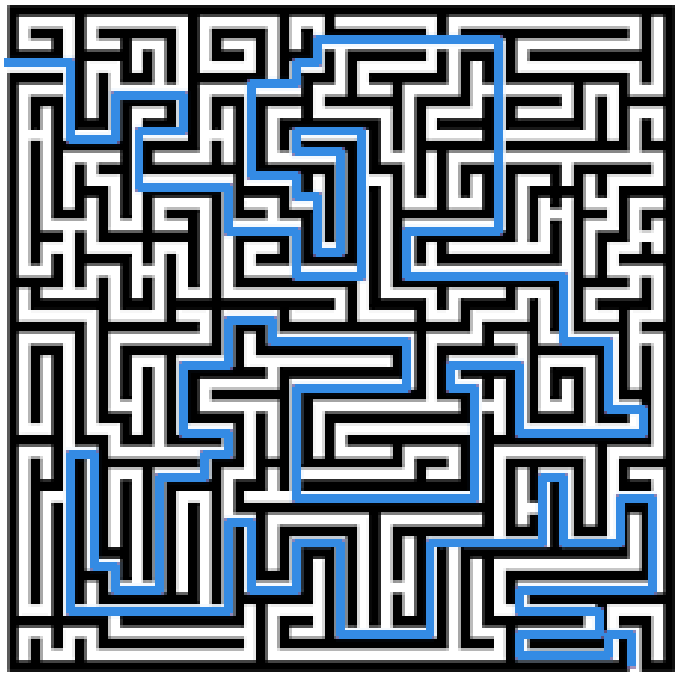
Rappel : problème du labyrinthe



Recherche de chemin dans un graphe

- Parcours en profondeur ou en largeur
- Renvoie un chemin possible

Rappel : problème du labyrinthe

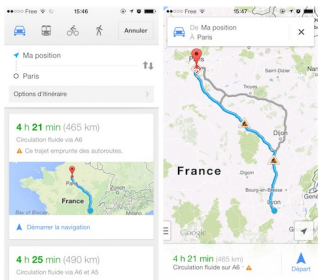


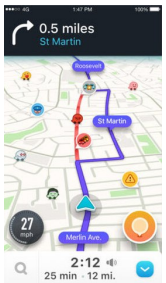
Recherche de chemin dans un graphe

- Parcours en profondeur ou en largeur
- Renvoie un chemin possible

➜ *Et s'il y en a plusieurs ?*

Application pratique : Waze ou Google Itinéraire

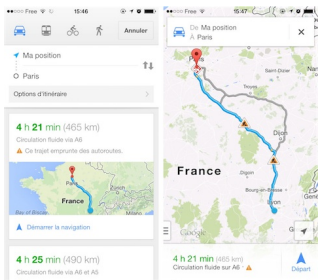


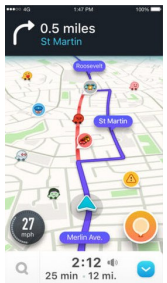


Trouver le meilleur chemin

➜ Problème d'optimisation

Application pratique : Waze ou Google Itinéraire

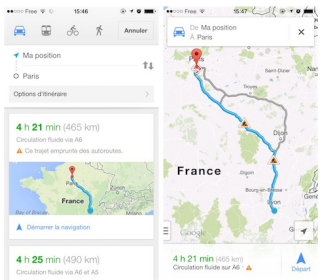


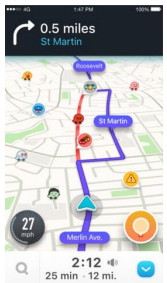


Trouver le meilleur chemin

- ➜ Problème d'optimisation
- ✓ Parcours en largeur sur un graphe non-orienté
  - ➜ *Donne effectivement le plus court chemin en nombre d'arêtes*

Application pratique : Waze ou Google Itinéraire





Trouver le meilleur chemin

- ➜ Problème d'optimisation
- ✓ Parcours en largeur sur un graphe non-orienté
  - ➜ *Donne effectivement le plus court chemin en nombre d'arêtes*
- ✗ Toutes les routes ne sont pas à double-sens
  - ➜ Graphes orientés
- ✗ Chaque portion de route prend un temps différent
  - ➜ Pondération des arêtes

Optimisation : approche naïve

Principe

Énumérer tous les chemins possibles pour trouver le plus court.

Optimisation : approche naïve

Principe

Énumérer tous les chemins possibles pour trouver le plus court.

Algorithme simplifié

```
def all_paths(G,s,t):
    C = all_paths_between_s_and_t_in_G(G,s,t)
    return min(C, key=length)
```

Optimisation : approche naïve

Principe

Énumérer tous les chemins possibles pour trouver le plus court.

Algorithme simplifié

```
def all_paths(G,s,t):
    C = all_paths_between_s_and_t_in_G(G,s,t)
    return min(C, key=length)
```

Difficulté

Il y a jusqu'à $\mathcal{O}(|E|!)$ éléments dans C !

➜ Peut-on trouver un algorithme plus efficace ?

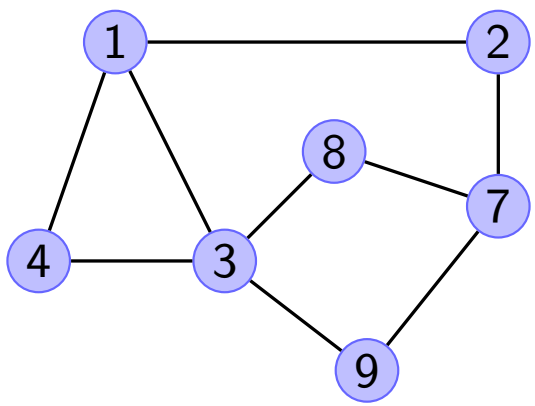
Plan

Graphes : définitions

Rappel : graphe non-orienté

On considère $G = (V, E)$ où :

- V un ensemble de sommets (ou nœuds) ;
- E un ensemble d'arêtes ;
- Une arête $e \in E$ est une paire d'éléments de V ;

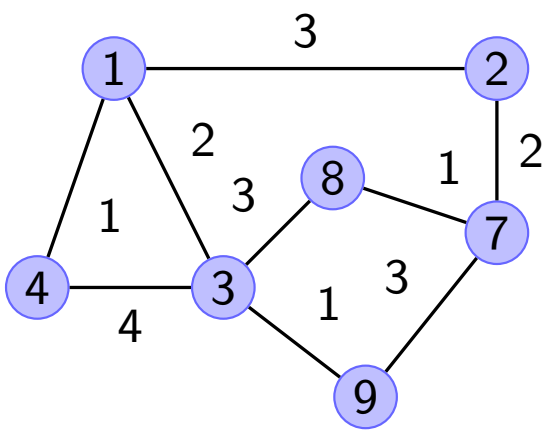


Graphes : définitions

Graphe non-orienté pondéré

On considère $G = (V, E)$ où :

- V un ensemble de sommets (ou nœuds) ;
- E un ensemble d'arêtes ;
- Une arête $e \in E$ est une paire d'éléments de V ;
- $\omega : E \longrightarrow \mathbb{R}$ est une fonction de pondération des arêtes ;

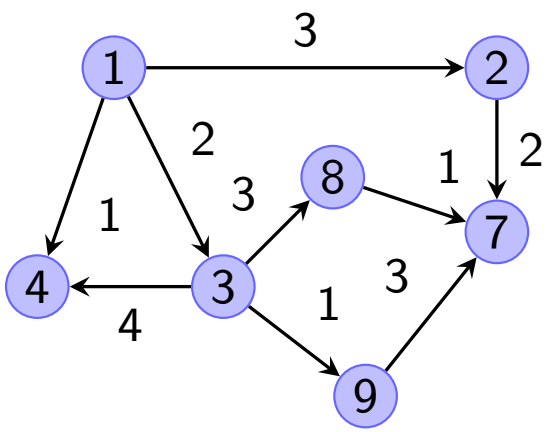


Graphes : définitions

Graphe ~~non~~-orienté pondéré

On considère $G = (V, E)$ où :

- V un ensemble de sommets (ou nœuds) ;
- E un ensemble d'arcs ;
- Un arc $e \in E$ est un couple d'éléments de V ;
- $\omega : E \longrightarrow \mathbb{R}$ est une fonction de pondération des arcs ;



Graphe : définitions (suite)

Chemin

Dans un graphe orienté :

- Un chemin de x à y est une suite finie d'arcs consécutifs reliant x à y.

Graphe : définitions (suite)

Chemin

Dans un graphe orienté :

- Un chemin de x à y est une suite finie d'arcs consécutifs reliant x à y.

Distance

Dans un graphe orienté pondéré :

- Le coût d'un chemin c est la somme des poids des arcs :

$$cout(c) = \sum_{e \in c} \omega(e)$$

Graphe : définitions (suite)

Chemin

Dans un graphe orienté :

- Un chemin de x à y est une suite finie d'arcs consécutifs reliant x à y.

Distance

Dans un graphe orienté pondéré :

- Le coût d'un chemin c est la somme des poids des arcs :

$$cout(c) = \sum_{e \in c} \omega(e)$$

- On parle aussi de distance de x à y (pour c reliant x à y).

Structure de données

Liste d'adjacence

- Encombrement mémoire en $\mathcal{O}(|E| + |V|)$
- Parcours des voisins d'un sommet u en $\mathcal{O}(deg(u))$
- Stockage des pondérations : {a:{b:2,c:3},...}
- Accès au poids d'un arc en $\mathcal{O}(1)$

comme ajout d'un arc, suppression d'un arc,...

Structure de données

Liste d'adjacence

- Encombrement mémoire en $\mathcal{O}(|E| + |V|)$
- Parcours des voisins d'un sommet u en $\mathcal{O}(deg(u))$
- Stockage des pondérations : `{a:{b:2,c:3},...}`
- Accès au poids d'un arc en $\mathcal{O}(1)$

  `comme ajout d'un arc, suppression d'un arc,...`

Matrice d'adjacence

- Encombrement mémoire en $\mathcal{O}(|V|^2)$
- Parcours des voisins d'un sommet u en $\mathcal{O}(|V|)$
- Stockage des pondérations : $tab[i,j] = \omega(i,j)$
- Accès au poids d'un arc en $\mathcal{O}(1)$

  `comme ajout d'un arc, suppression d'un arc,...`

Problème de plus court chemin

Problème d'optimisation

Données :

- Un graphe orienté $G = (V, E)$
- Une fonction de pondération $\omega : E \longrightarrow \mathbb{R}$
- Un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$

Problème de plus court chemin

Problème d'optimisation

Données :

- Un graphe orienté $G = (V, E)$
- Une fonction de pondération $\omega : E \longrightarrow \mathbb{R}$
- Un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$

Question :

➜ Quelle est le plus court chemin de s à t ?
Soit C l'ensemble des chemins solution (reliant s à t).
On cherche $c \in C$ tel que $\forall c' \in C, cout(c') \geq cout(c)$

Problème de plus court chemin

Problème d'optimisation

Données :

- Un graphe orienté $G = (V, E)$
- Une fonction de pondération $\omega : E \longrightarrow \mathbb{R}$
- Un sommet de départ $s \in V$ et un sommet d'arrivée $t \in V$

Question :

➜ Quelle est le plus court chemin de s à t ?
Soit C l'ensemble des chemins solution (reliant s à t).
On cherche $c \in C$ tel que $\forall c' \in C, cout(c') \geq cout(c)$

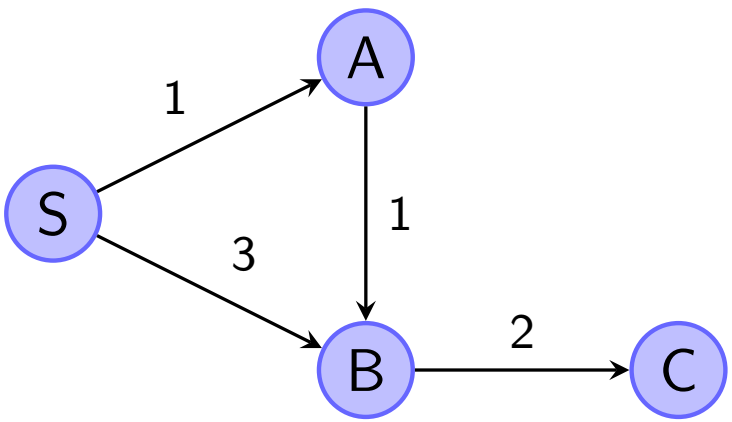
Remarque

Le même problème peut être défini sur les graphes non-orientés pondérés.

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

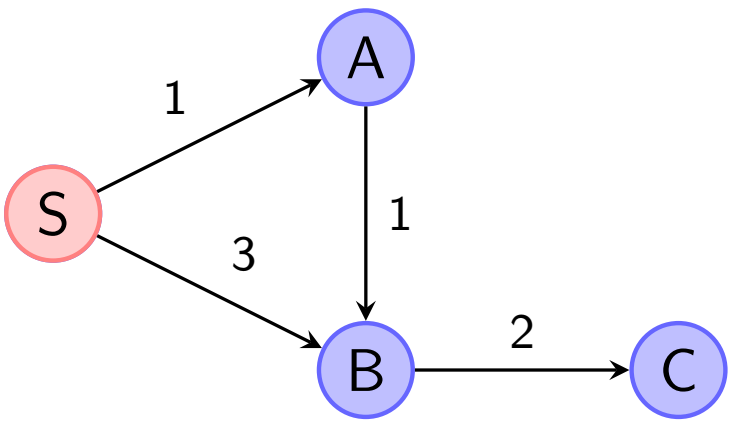


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

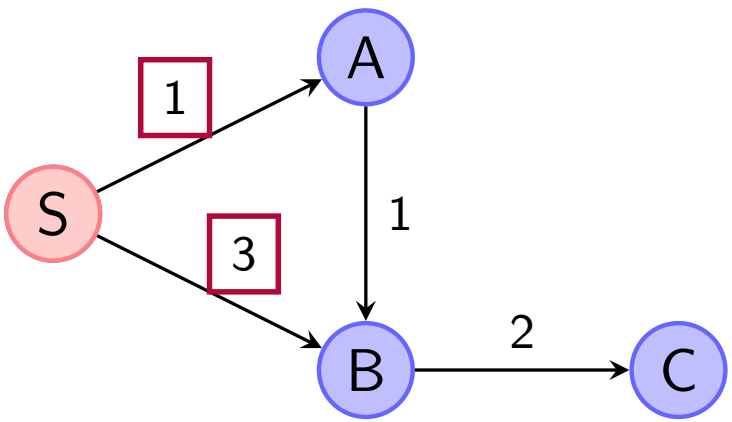


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

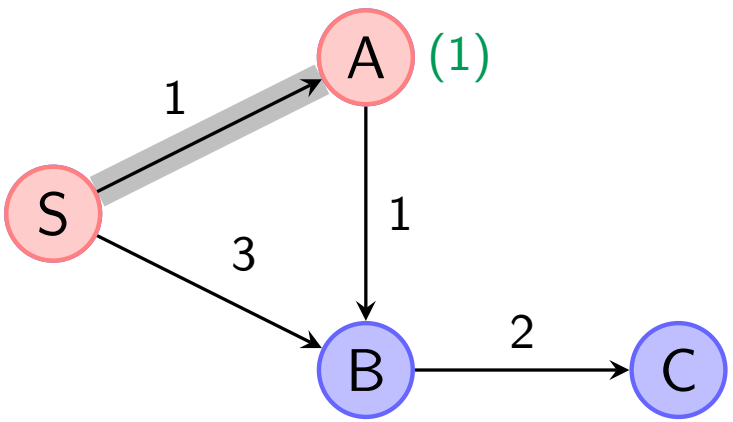


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

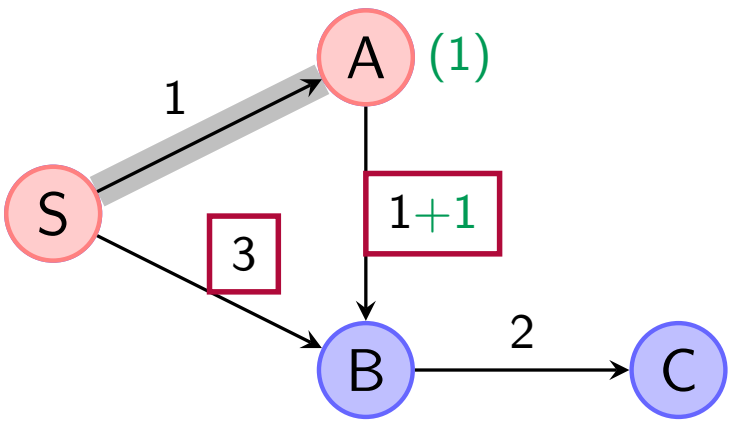


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

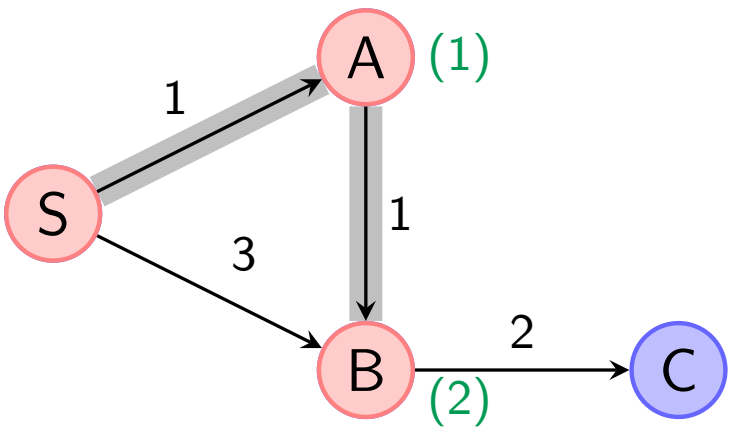


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.

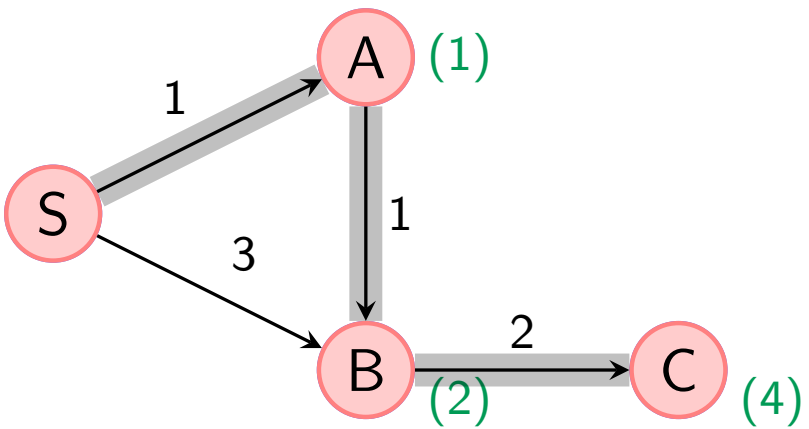


skip

Un plus court chemin

Idée de l'algorithme (inspiré par Dijkstra, 1959)

Parcours en largeur en tenant compte des poids.



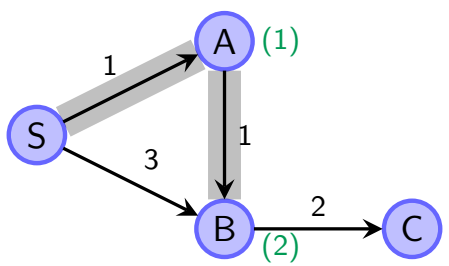
etc.

Algorithme des plus courts chemins

Structures de données

L'algorithme nécessite de manipuler :

- La liste des prochains sommets à visiter (comme dans BFS)
  - ➜ Dans l'algorithme des plus courts chemins, on parle de frontière.
- Le coût du meilleur chemin de s à chaque sommet visité
  - ➜ Mémoriser une distance pour chaque sommet.
- Les arcs sélectionnés
  - ➜ Mémoriser un sommet parent pour chaque sommet.



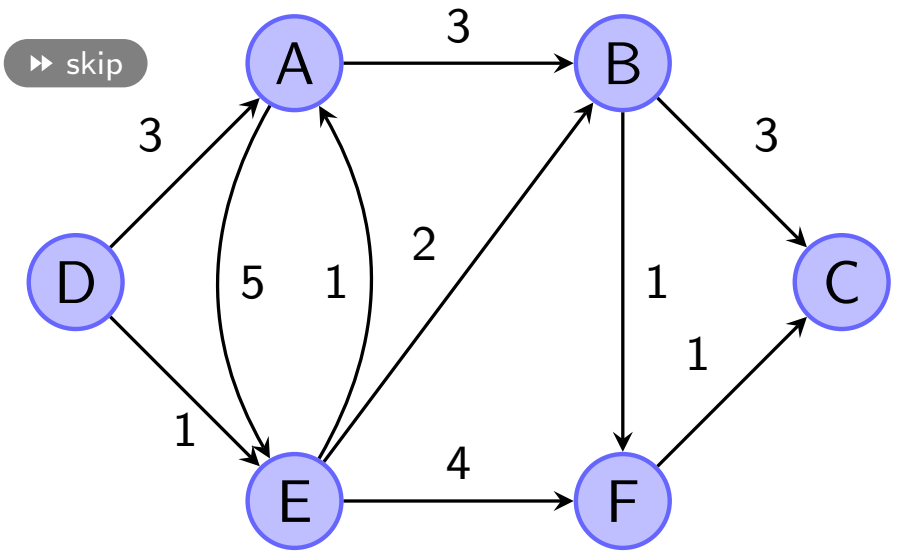
Algorithme des plus courts chemins en python

```
def shortest_path(graph,s):
    frontier = [s]
    parent = {s: None}
    dist = {s: 0}

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):
            if y not in parent:
                frontier.append(y)
            # update
            new_dist = dist[x] + distance(graph,x,y)
            if y not in dist or dist[y] > new_dist:
                dist[y] = new_dist
                parent[y] = x

    return parent
```

Exemple complet

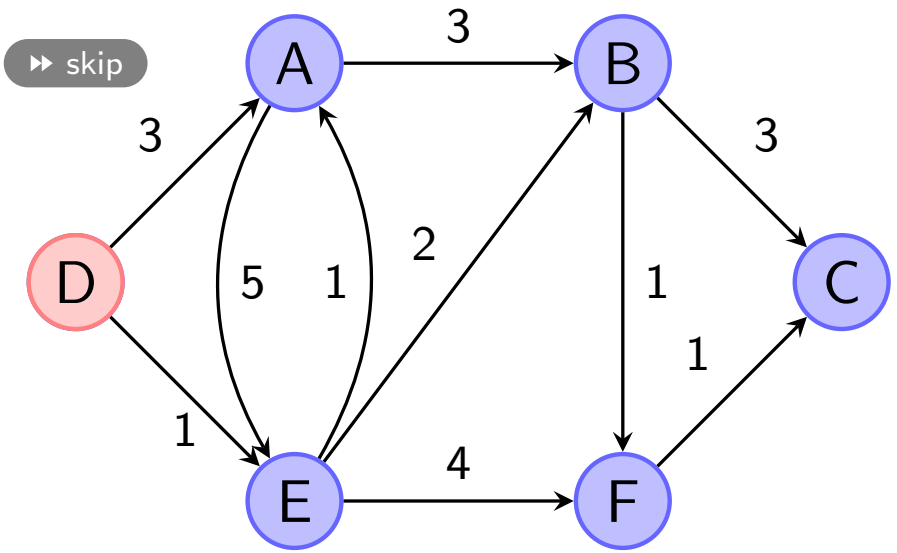


Nœud	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontière = $\{D\}$

x =

Exemple complet

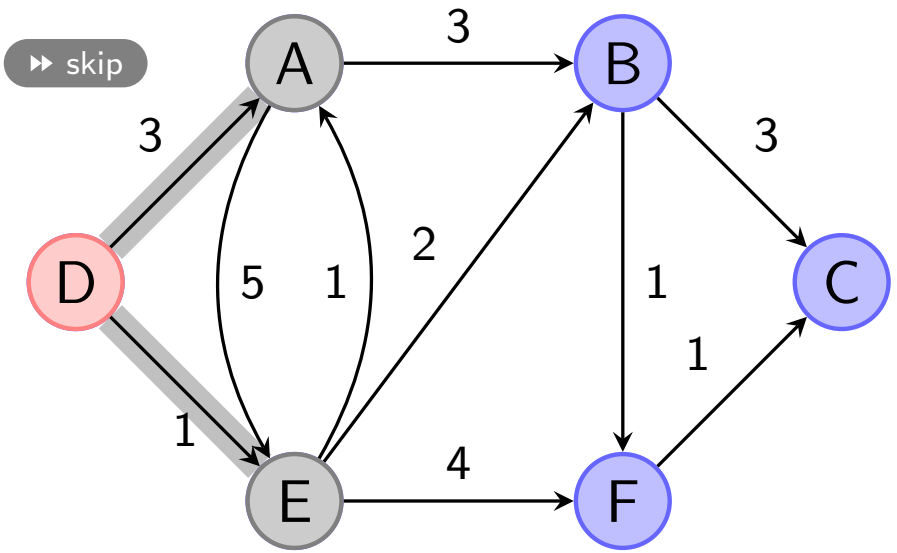


Nœud	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontière = $\{\}$

x = D

Exemple complet

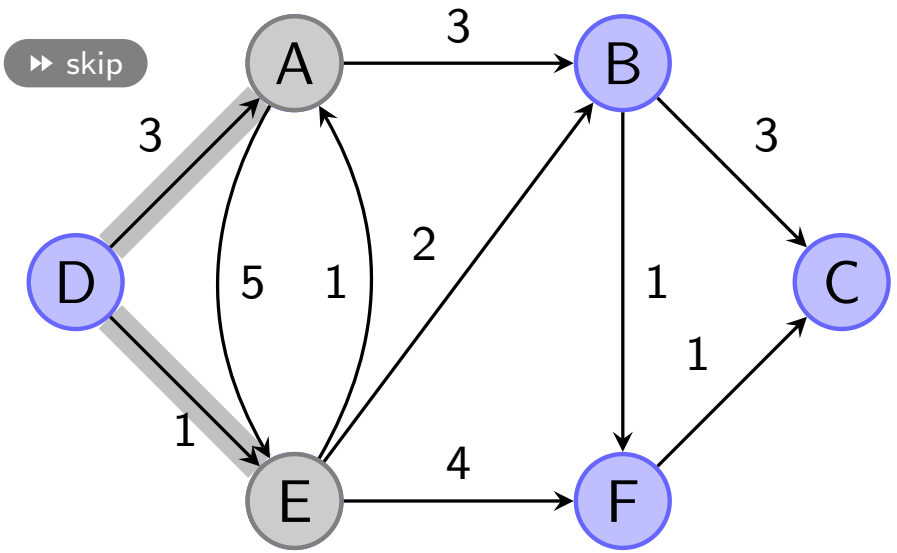


Nœud	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontière = $\{A, E\}$

x = D

Exemple complet

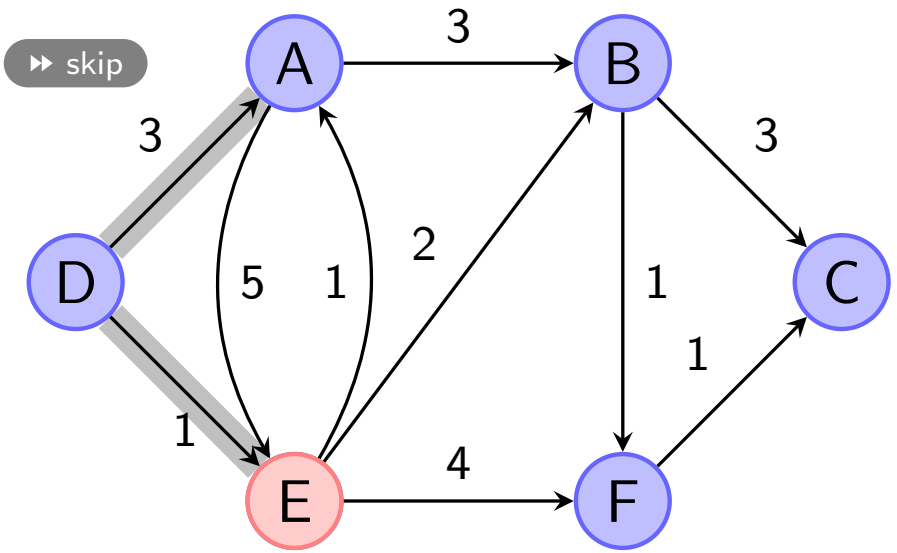


Nœud	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontière = $\{A, E\}$

x = D

Exemple complet

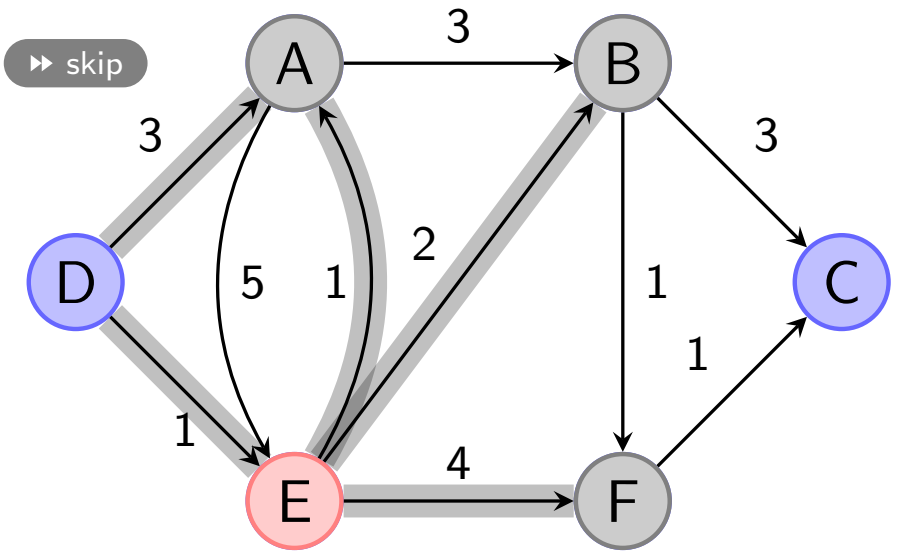


Nœud	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontière = $\{A\}$

x = E

Exemple complet

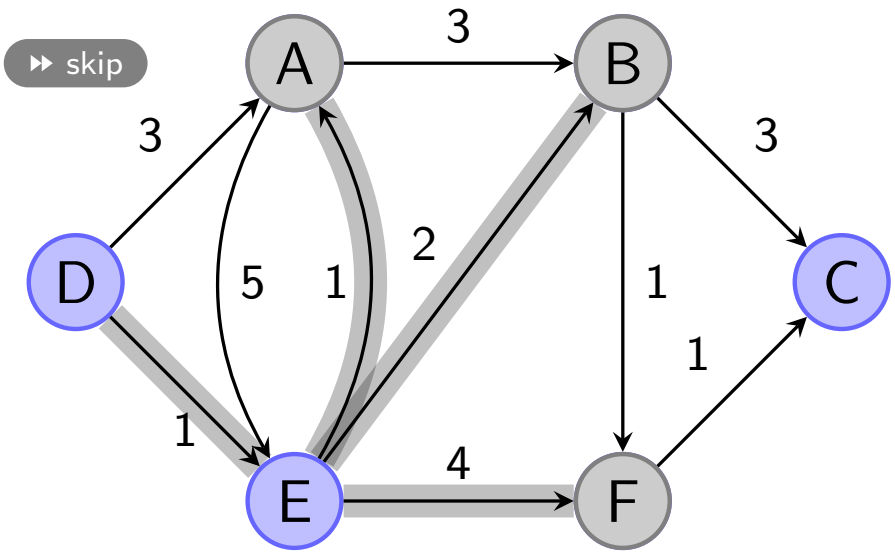


Nœud	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontière = $\{A, B, F\}$

x = E

Exemple complet

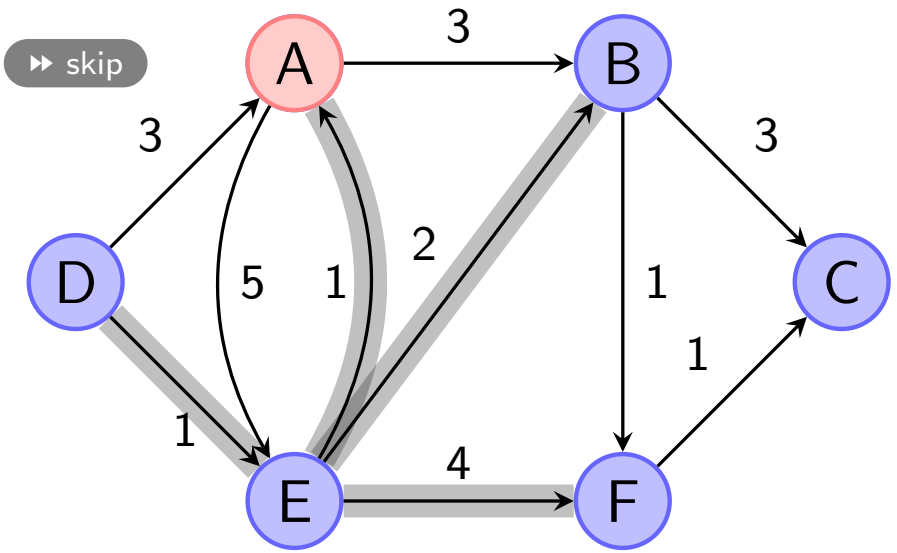


Nœud	Distance	Parent
A	1+1	E
B	1+2	E
C	∞	•
D	0	•
E	1	D
F	1+4	E

Frontière = $\{A, B, F\}$

x = E

Exemple complet

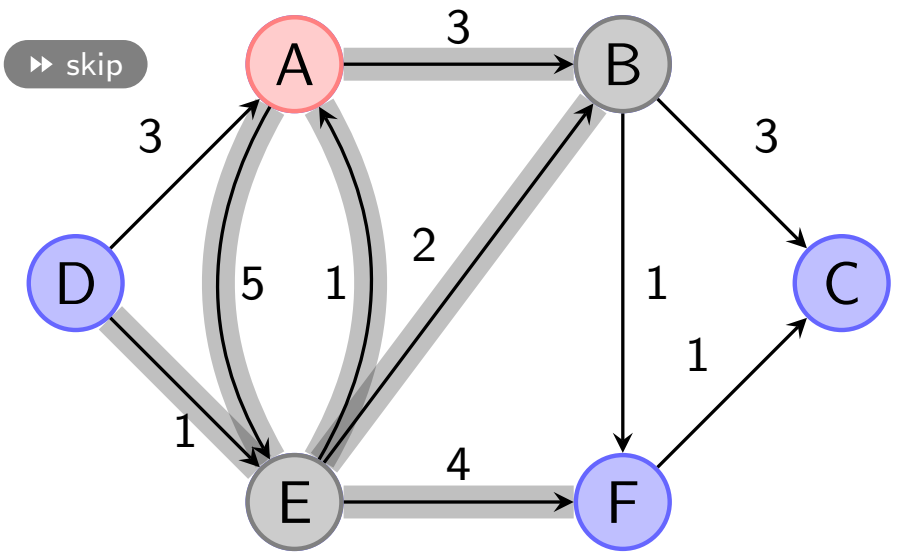


Nœud	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

x = A

Exemple complet

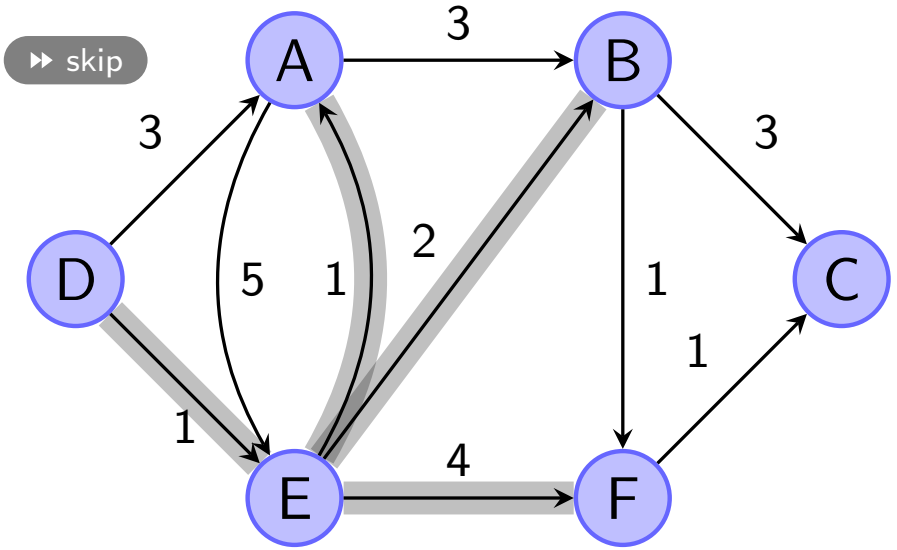


Nœud	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

x = A

Exemple complet

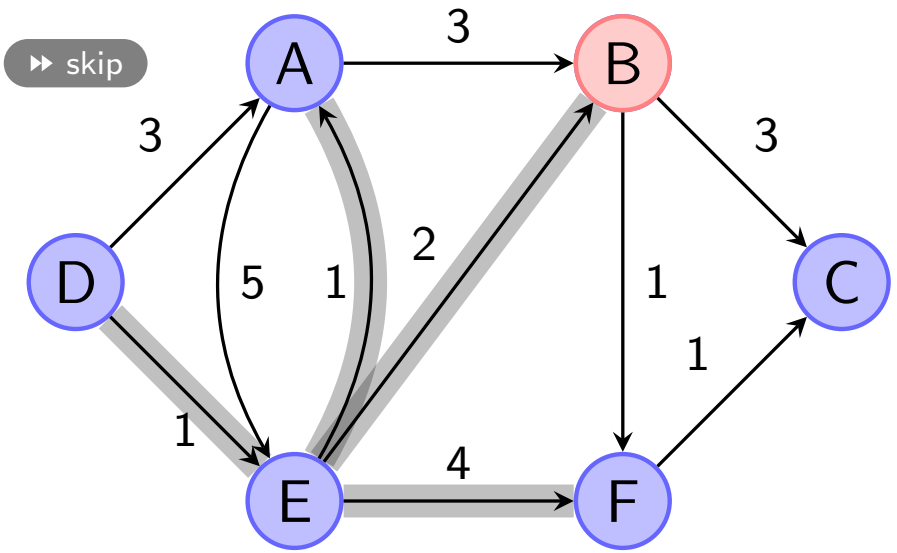


Nœud	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

x = A

Exemple complet

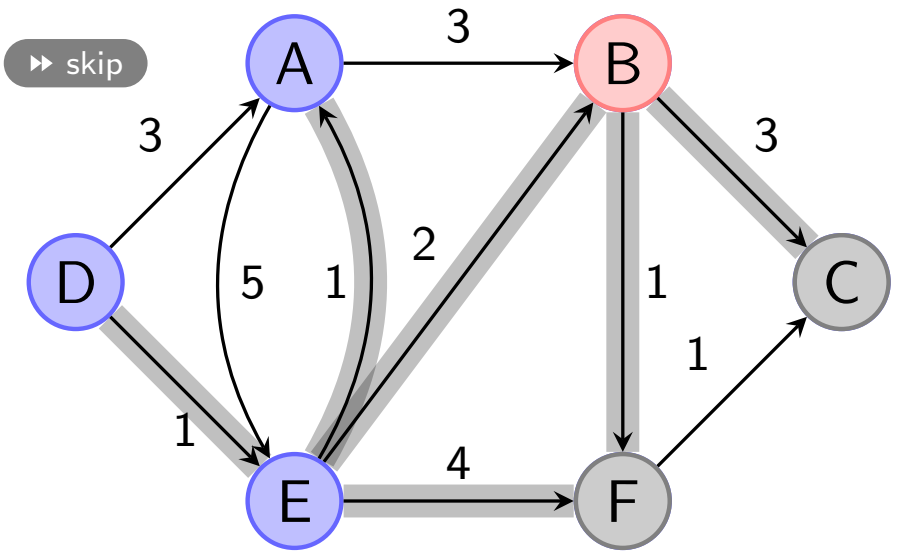


Nœud	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontière = $\{F\}$

x = B

Exemple complet

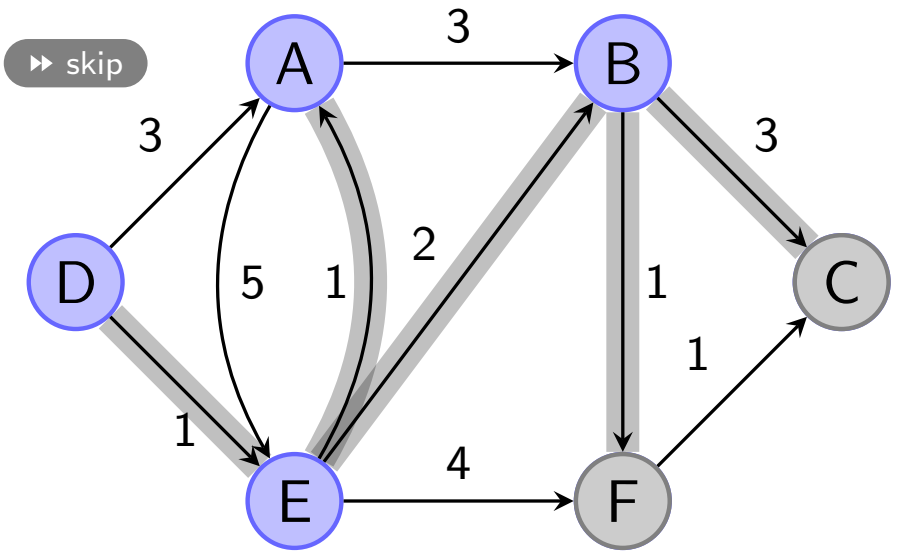


Nœud	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontière = $\{C, F\}$

x = B

Exemple complet

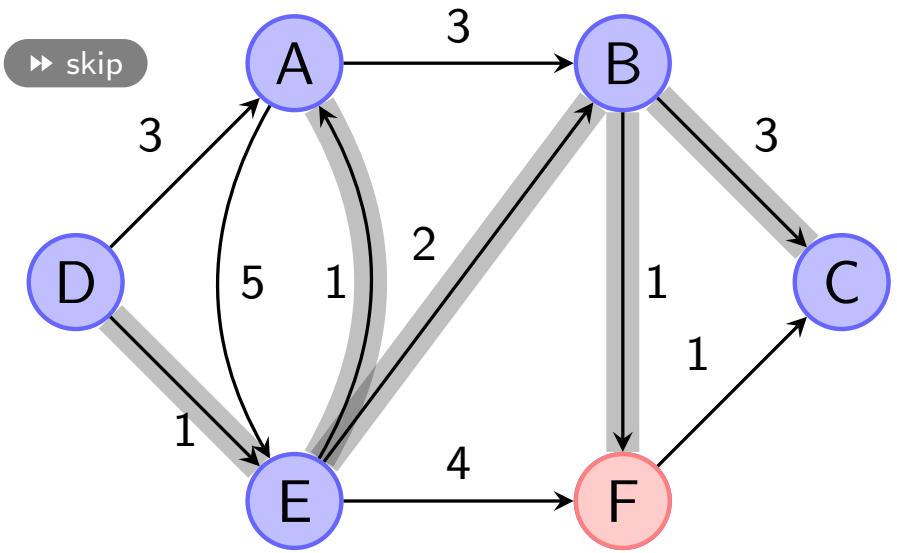


Nœud	Distance	Parent
A	2	E
B	3	E
C	3+3	B
D	0	•
E	1	D
F	3+1	B

Frontière = $\{C, F\}$

x = B

Exemple complet

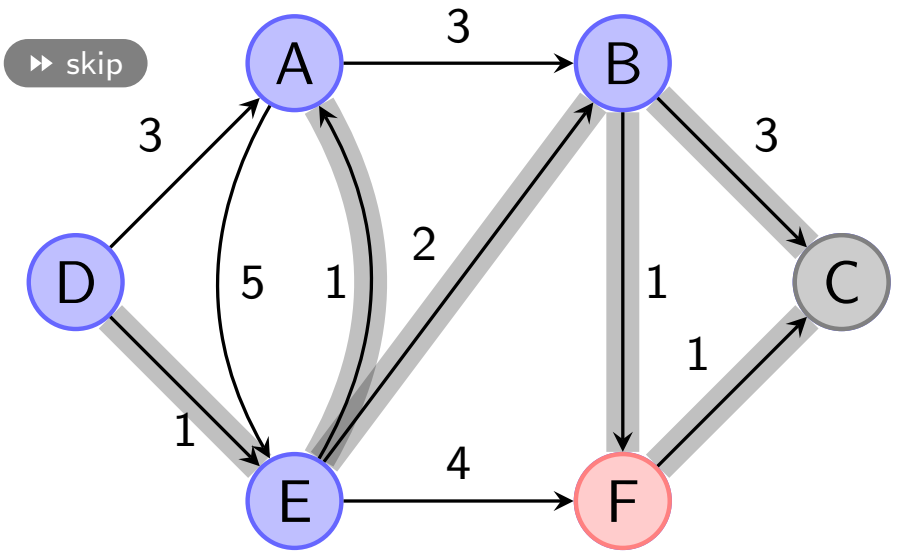


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

x = F

Exemple complet

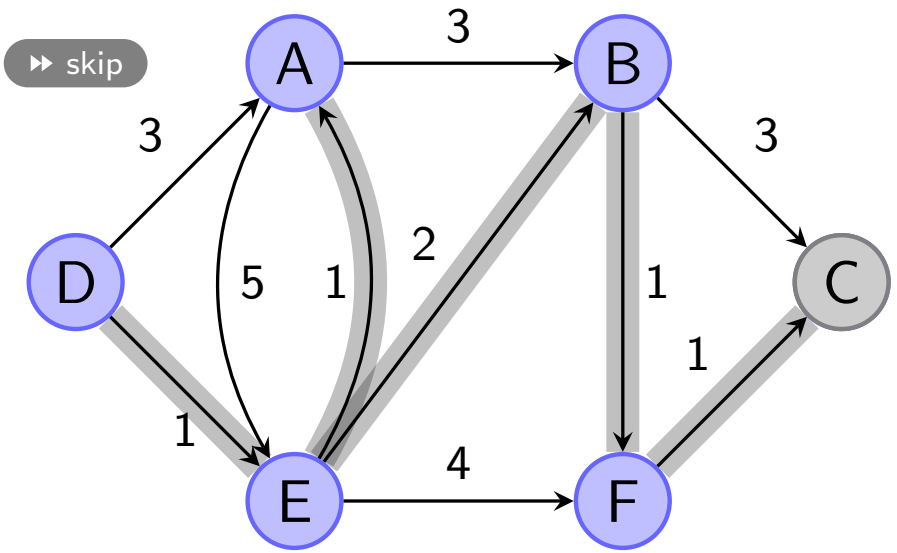


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

x = F

Exemple complet

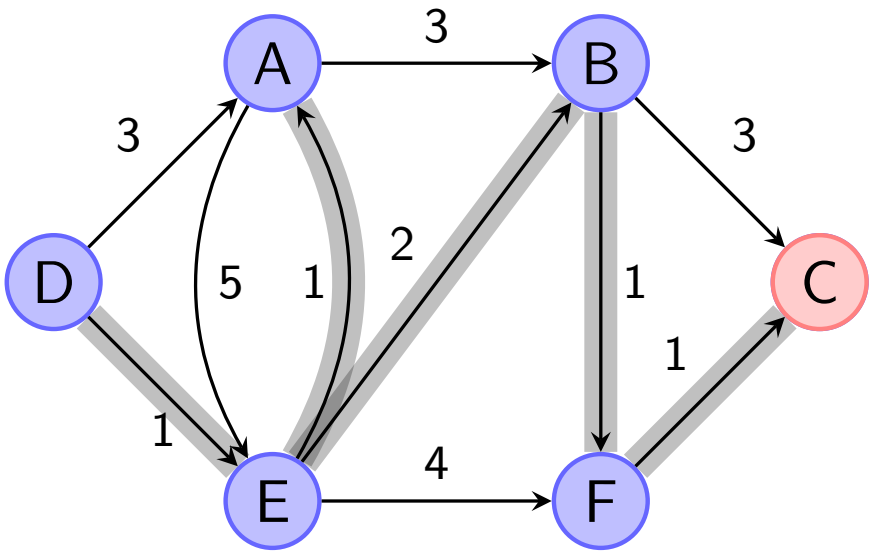


Nœud	Distance	Parent
A	2	E
B	3	E
C	4+1	F
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

x = F

Exemple complet



Nœud	Distance	Parent
A	2	E
B	3	E
C	5	F
D	0	•
E	1	D
F	4	B

Frontière = {}

x = C

Reconstruction du chemin

Quel est le chemin de s à t ?

- Le tableau "distance" donne le coût minimum de s à t ;
- Le tableau "parent" donne le sommet précédent de chaque noeud visité ;

➜ Comment reconstruire le chemin à partir de "parent" ?

Reconstruction du chemin

Quel est le chemin de s à t ?

- Le tableau "distance" donne le coût minimum de s à t ;
- Le tableau "parent" donne le sommet précédent de chaque noeud visité ;

➜ Comment reconstruire le chemin à partir de "parent" ?

Code

```
def construct_path(parent,t):
    path = [t]
    current = t
    while not parent[current] is None:
        current = parent[current]
        path.insert(0,current)
    return path
```

Plan

Un problème concret

Problèmes de graphe...

- Trouver le plus court chemin dans un graphe (extract_min_dist)
- Construire un arbre couvrant de coût minimum (Cours 3)

➜ Nécessite d'utiliser une file de priorité !

File de priorité

Stockage des données organisé suivant un ordre de priorité

Un problème concret

Problèmes de graphe...

- Trouver le plus court chemin dans un graphe (extract_min_dist)
- Construire un arbre couvrant de coût minimum (Cours 3)

➜ Nécessite d'utiliser une file de priorité !

File de priorité

Stockage des données organisé suivant un ordre de priorité

Définition

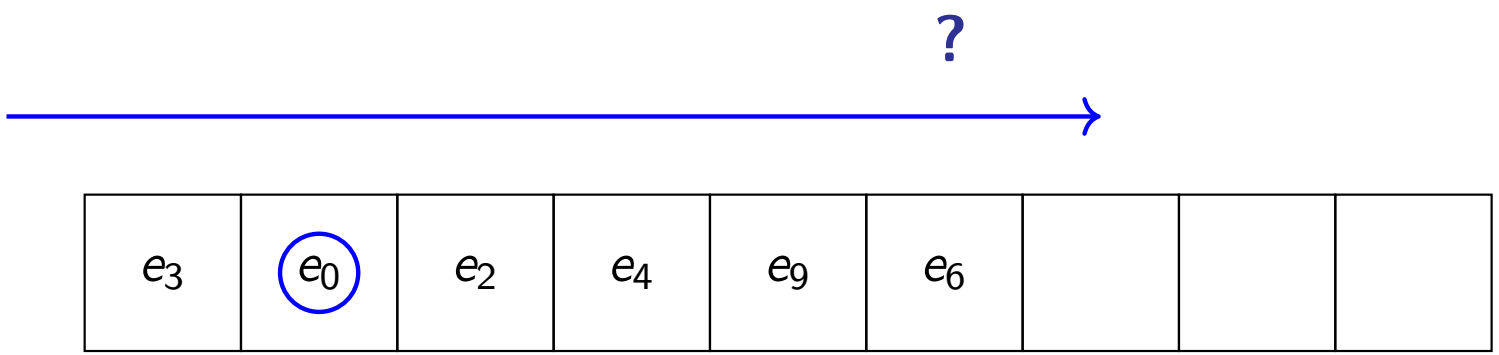
Structure de données abstraite opérant sur un ensemble ordonné et muni de fonctions efficaces pour :

1. Trouver l'élément minimum (ou maximum) ;
2. Insérer un élément dans la file ;
3. Retirer l'élément minimum (une fois trouvé).

Implémentation : tableau

Tableau d'éléments non-ordonnés

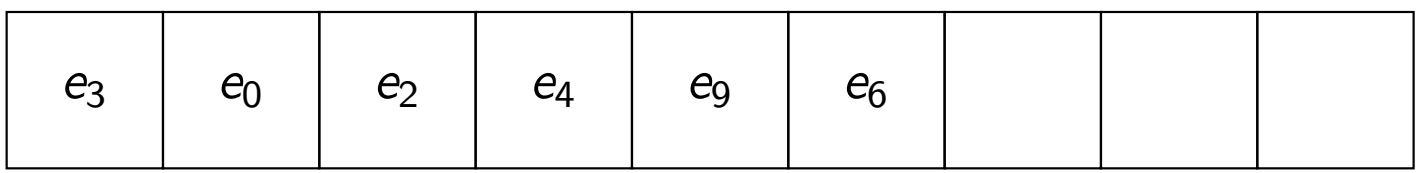
- Rechercher l'élément minimum ?



Implémentation : tableau

Tableau d'éléments non-ordonnés

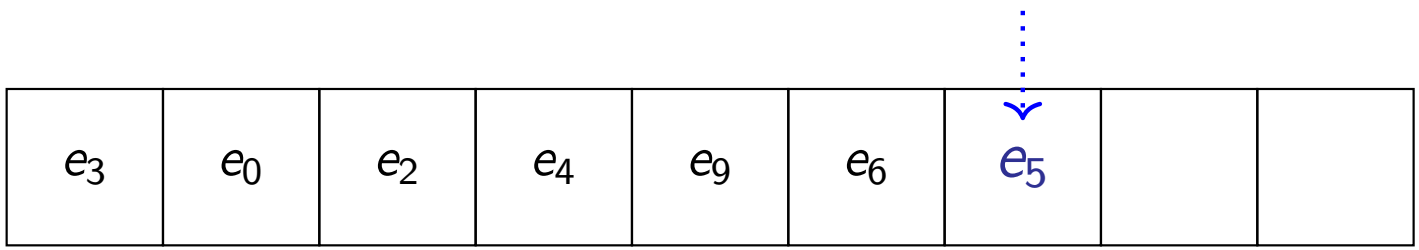
- ✗ Rechercher l'élément minimum : $\mathcal{O}(n)$
- Insérer un élément ?



Implémentation : tableau

Tableau d'éléments non-ordonnés

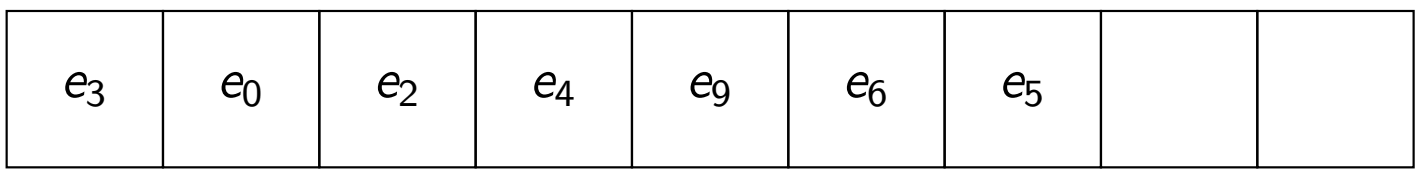
- ✗ Rechercher l'élément minimum : $\mathcal{O}(n)$
- ✓ Insérer un élément (à la fin s'il reste de la place) : $\mathcal{O}(1)$



Implémentation : tableau

Tableau d'éléments non-ordonnés

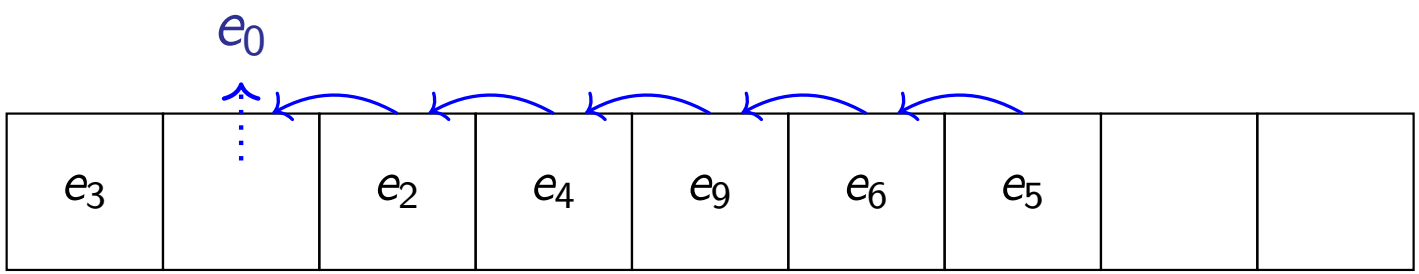
- ✗ Rechercher l'élément minimum : $\mathcal{O}(n)$
- ✓ Insérer un élément (à la fin s'il reste de la place) : $\mathcal{O}(1)$
- Retirer l'élément minimum ?



Implémentation : tableau

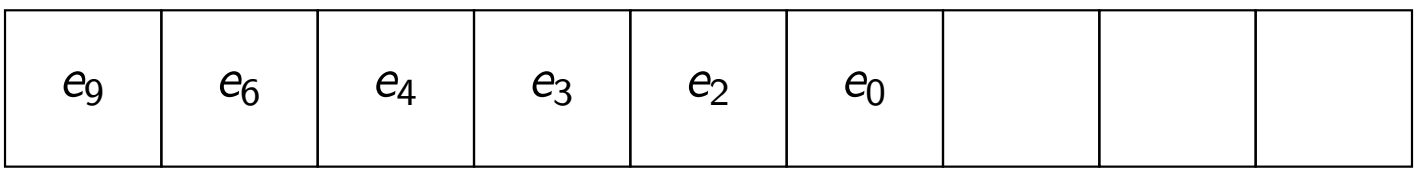
Tableau d'éléments non-ordonnés

- ✗ Rechercher l'élément minimum : $\mathcal{O}(n)$
- ✓ Insérer un élément (à la fin s'il reste de la place) : $\mathcal{O}(1)$
- ✗ Retirer l'élément minimum : $\mathcal{O}(n)$ (décaler tous les éléments)



Implémentation : tableau trié

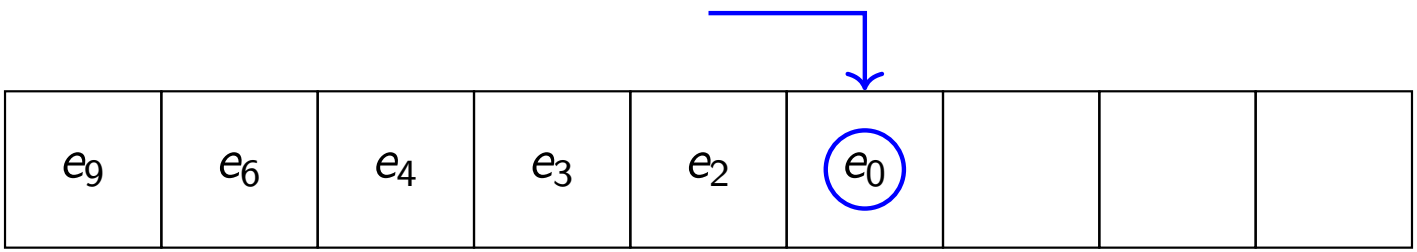
Tableau d'éléments ordonnés



Implémentation : tableau trié

Tableau d'éléments ordonnés

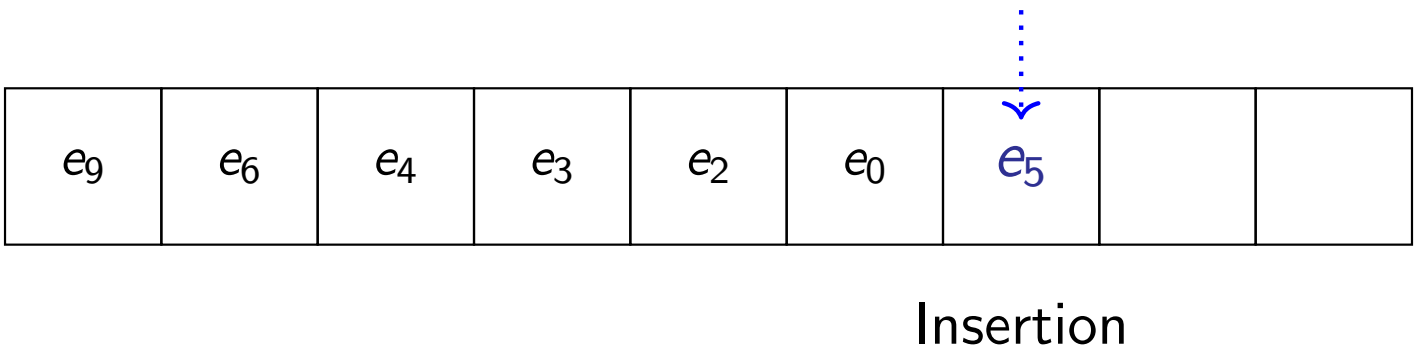
✓ Rechercher l'élément minimum : $\mathcal{O}(1)$



Implémentation : tableau trié

Tableau d'éléments ordonnés

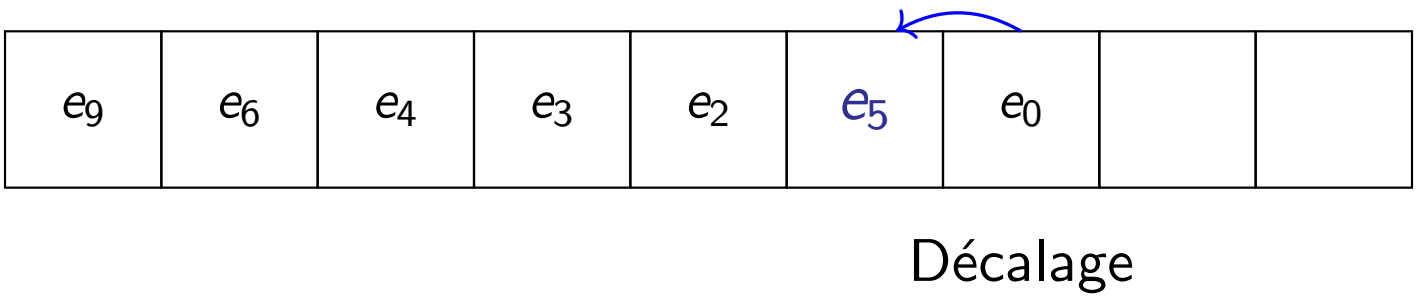
- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)



Implémentation : tableau trié

Tableau d'éléments ordonnés

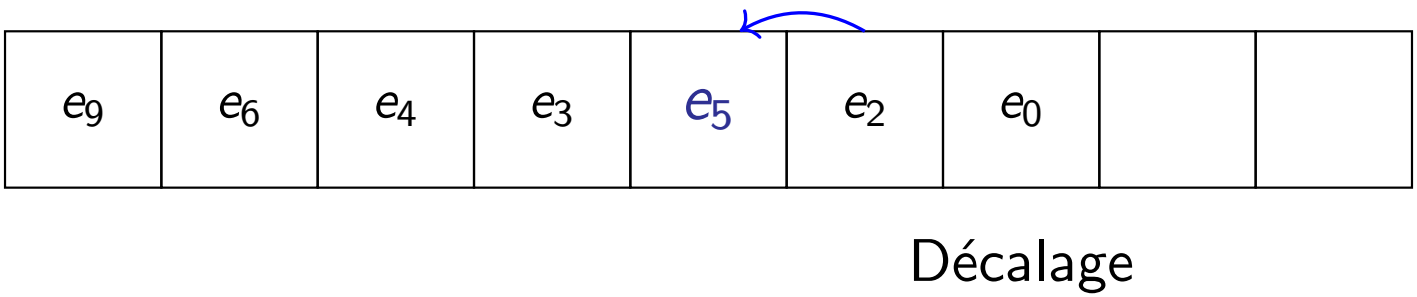
- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)



Implémentation : tableau trié

Tableau d'éléments ordonnés

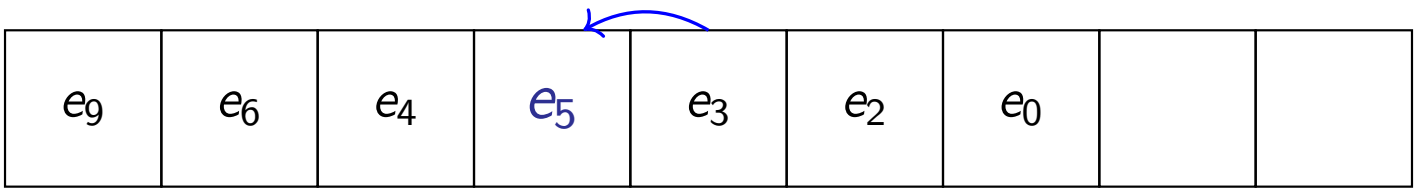
- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)



Implémentation : tableau trié

Tableau d'éléments ordonnés

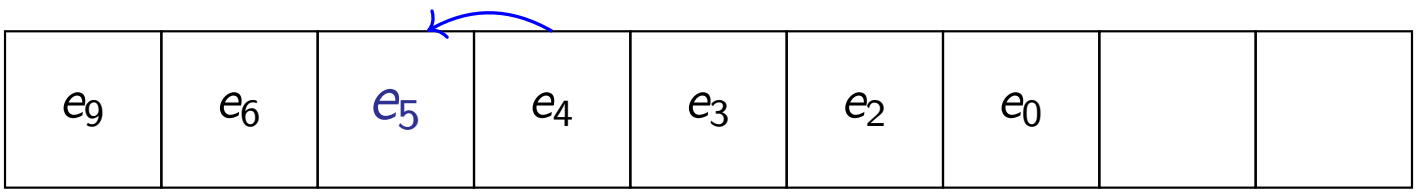
- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)



Implémentation : tableau trié

Tableau d'éléments ordonnés

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)



Implémentation : tableau trié

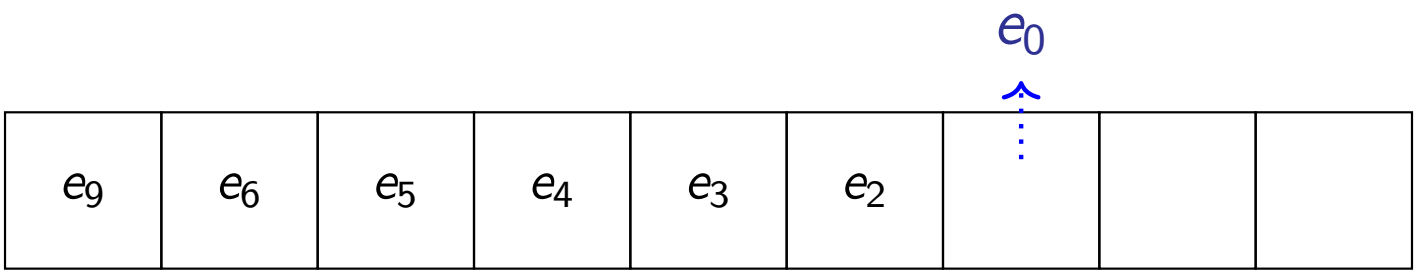
Tableau d'éléments ordonnés

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)

Implémentation : tableau trié

Tableau d'éléments ordonnés

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✗ Insérer un élément : $\mathcal{O}(n)$ (décaler les éléments)
- ✓ Retirer l'élément minimum : $\mathcal{O}(1)$ (si c'est trié dans le bon sens)



Implémentation : tas

Définition

Un tas est une structure de données abstraite conçue pour gérer efficacement une liste de priorités.

On veut pouvoir :

- Accéder le plus rapidement possible à l'élément de priorité maximum
- Trouver un compromis de performance (entre $\mathcal{O}(1)$ et $\mathcal{O}(n)$) pour l'insertion et l'extraction

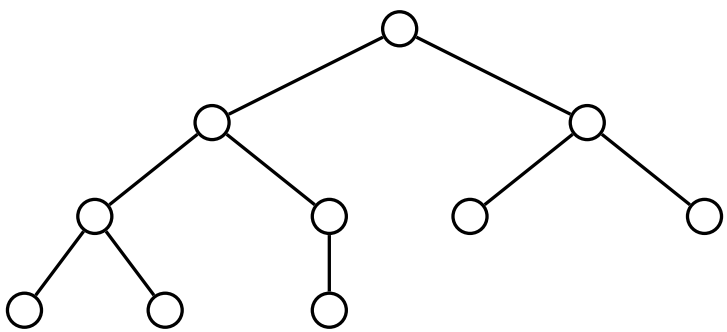
Implémentation : tas

Définition

Un tas est une structure de données abstraite conçue pour gérer efficacement une liste de priorités.

Arbre

Un graph non-orienté connexe et acyclique est un arbre.



Implémentation : tas

Définition

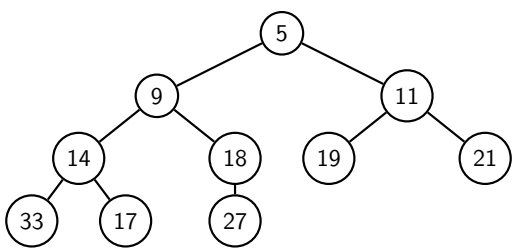
Un tas est une structure de données abstraite conçue pour gérer efficacement une liste de priorités.

Principe

Un arbre dont les sommets sont les valeurs de priorité, tel que :

- tas-min : la valeur de chaque sommet est inférieure à celles de ses fils
- tas-max : la valeur de chaque sommet est supérieure à celles de ses fils

➜ Dans un tas-min (resp. tas-max), la racine est alors l'élément minimal (resp. maximal)

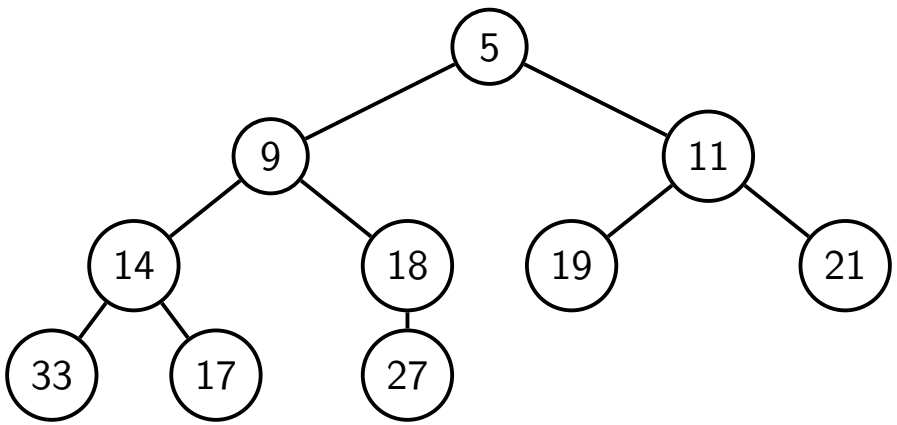


Tas binaire (Heap)

Tas binaire

Arbre binaire quasi-parfait :

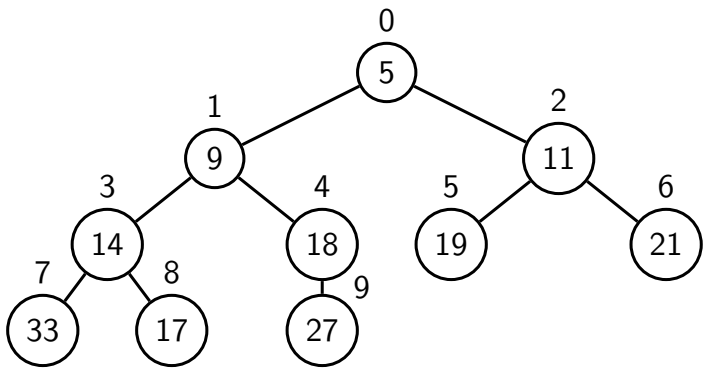
Toutes ses feuilles sont sur au plus deux niveaux, l'avant dernier niveau étant complet et les feuilles du dernier niveau sont regroupées le plus à gauche possible.



Implémentation d'un tas

En pratique

Un tas-min est un tableau vérifiant la propriété suivante :



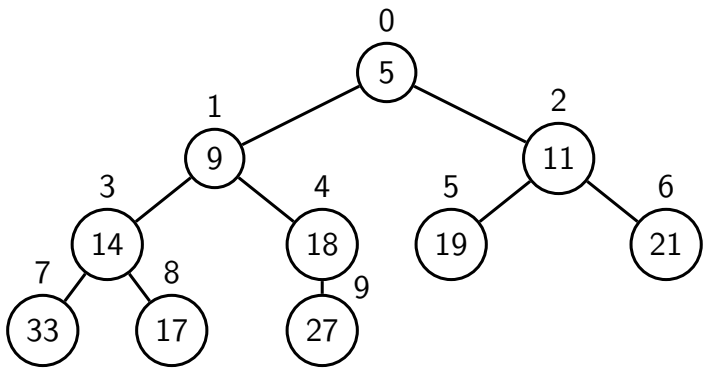
	0	1	2	3	4	5	6	7	8	9
tab :	5	9	11	14	18	19	21	33	17	27

racine	:	sommet 0
fils gauche du sommet i	:	sommet $2i+1$
fils droit du sommet i	:	sommet $2i+2$
parent du sommet i	:	sommet ?
sommet i est une feuille	:	sommet ?

Implémentation d'un tas

En pratique

Un tas-min est un tableau vérifiant la propriété suivante :



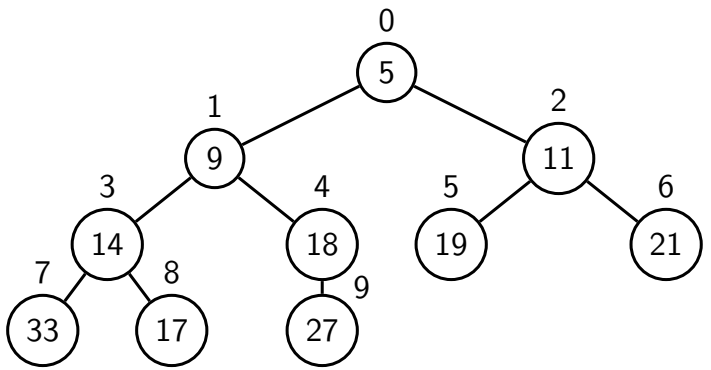
	0	1	2	3	4	5	6	7	8	9
tab :	5	9	11	14	18	19	21	33	17	27

racine	:	sommet 0
fils gauche du sommet i	:	sommet $2i+1$
fils droit du sommet i	:	sommet $2i+2$
parent du sommet i	:	sommet $\lfloor (i-1)/2 \rfloor$
sommet i est une feuille	:	sommet ?

Implémentation d'un tas

En pratique

Un tas-min est un tableau vérifiant la propriété suivante :



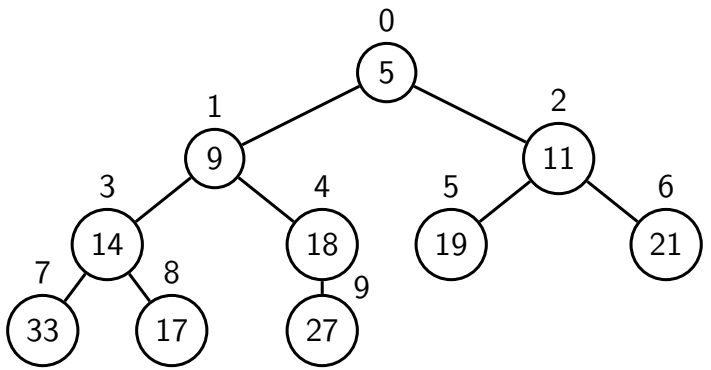
	0	1	2	3	4	5	6	7	8	9
tab :	5	9	11	14	18	19	21	33	17	27

racine	:	sommet 0
fils gauche du sommet i	:	sommet $2i+1$
fils droit du sommet i	:	sommet $2i+2$
parent du sommet i	:	sommet $\lfloor (i-1)/2 \rfloor$
sommet i est une feuille	:	sommet $2i+1 \geq n$

Implémentation d'un tas

En pratique

Un tas-min est un tableau vérifiant la propriété suivante :



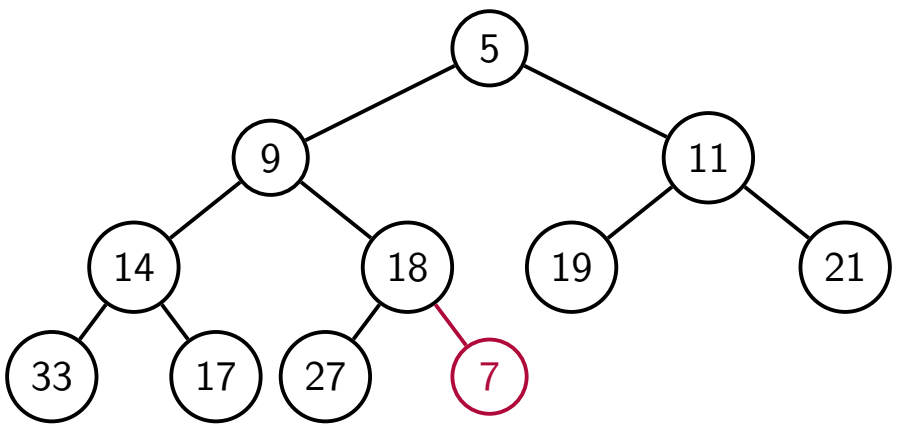
	0	1	2	3	4	5	6	7	8	9
tab :	5	9	11	14	18	19	21	33	17	27

$$\texttt{tab}[\lfloor (i-1)/2 \rfloor] < \texttt{tab}[\texttt{i}] \text{ pour tout } i \geq 1$$

Insertion dans un Tas-min

1. On insère la clef v à la fin du dernier niveau de l'arbre
   - à la fin du tableau

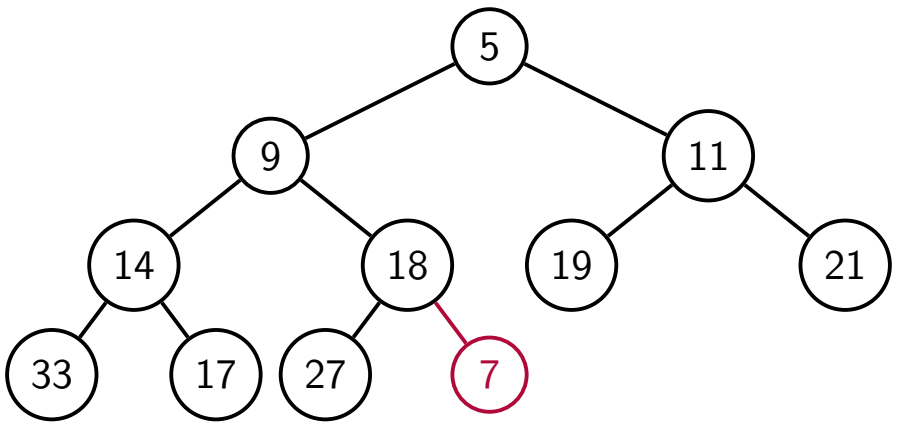
Exemple



Insertion dans un Tas-min

1. On insère la clef v à la fin du dernier niveau de l'arbre
   - à la fin du tableau
2. Tant que la clef de v est plus petite que la clef du père de v :
   - On échange le père et v

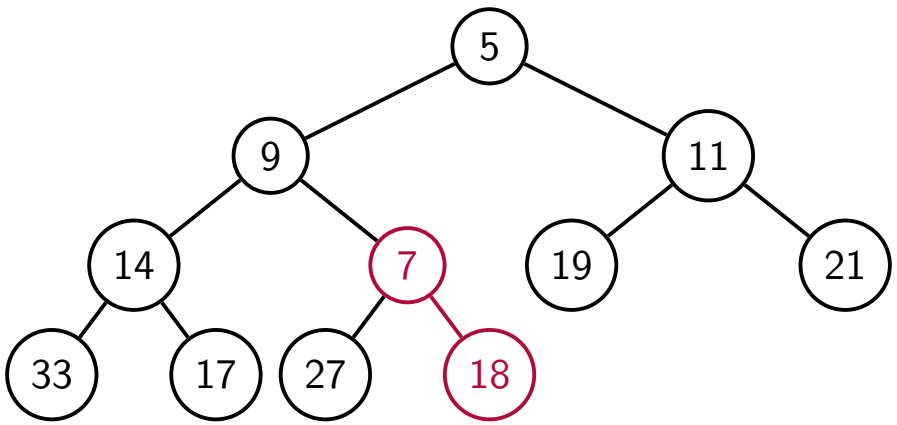
Exemple



Insertion dans un Tas-min

1. On insère la clef v à la fin du dernier niveau de l'arbre
   - à la fin du tableau
2. Tant que la clef de v est plus petite que la clef du père de v :
   - On échange le père et v

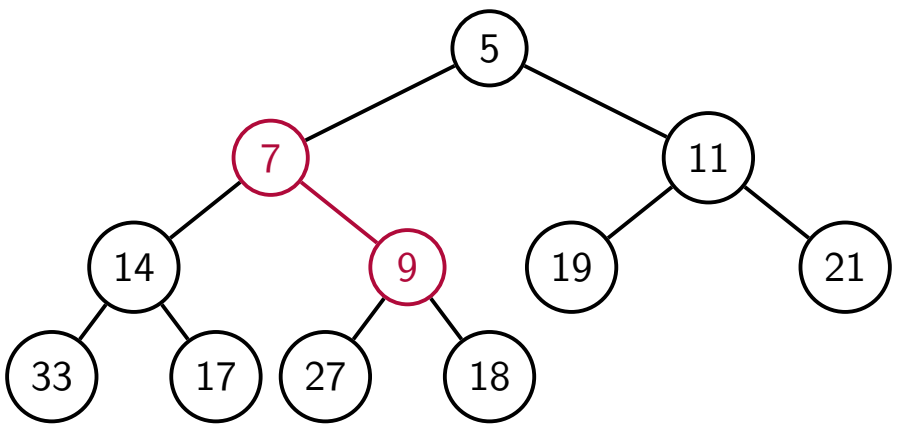
Exemple



Insertion dans un Tas-min

1. On insère la clef v à la fin du dernier niveau de l'arbre
   - à la fin du tableau
2. Tant que la clef de v est plus petite que la clef du père de v :
   - On échange le père et v

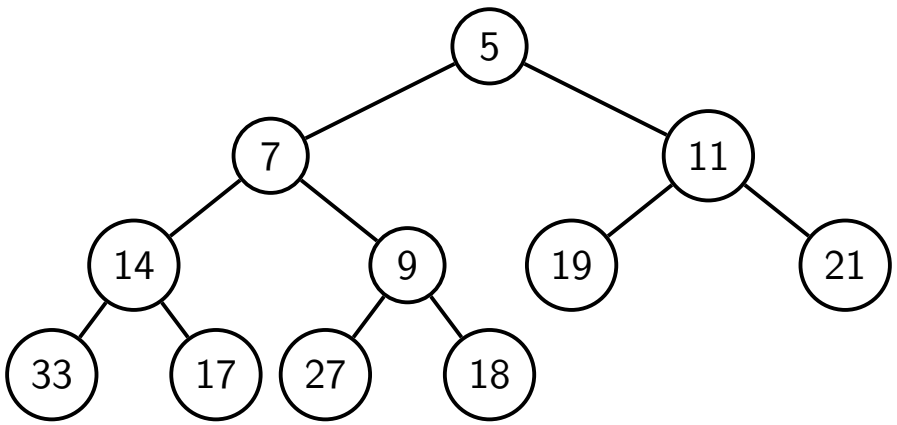
Exemple



Insertion dans un Tas-min

1. On insère la clef v à la fin du dernier niveau de l'arbre
   - à la fin du tableau
2. Tant que la clef de v est plus petite que la clef du père de v :
   - On échange le père et v

Exemple

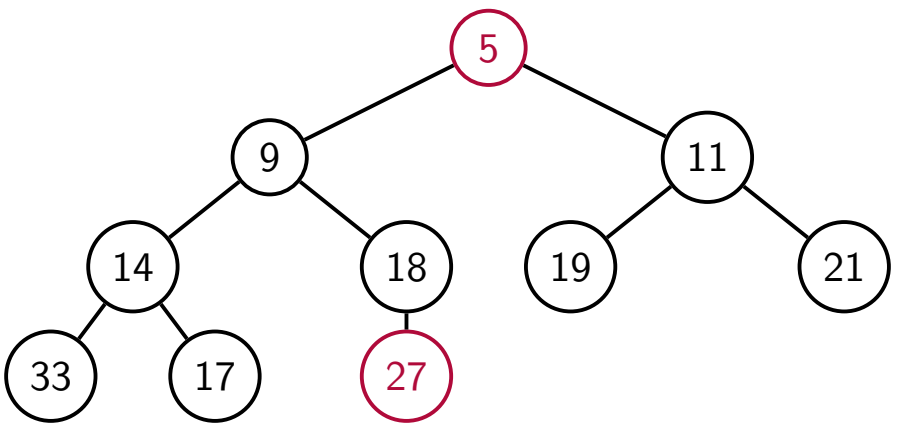


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau

Exemple

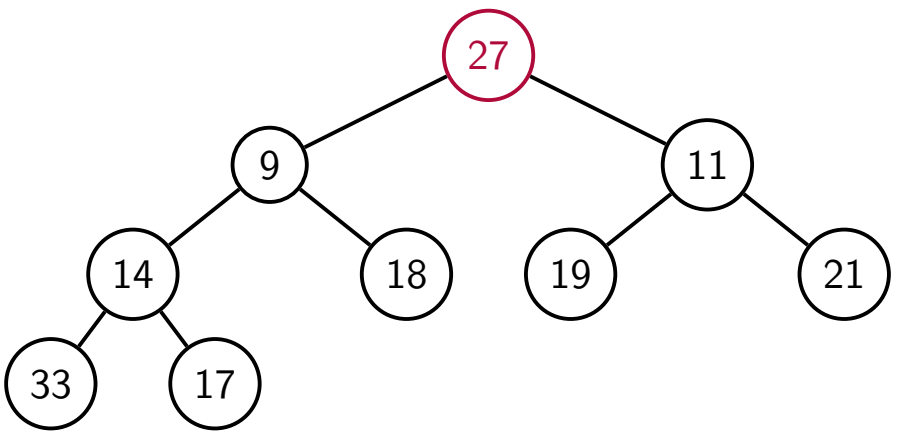


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau

Exemple

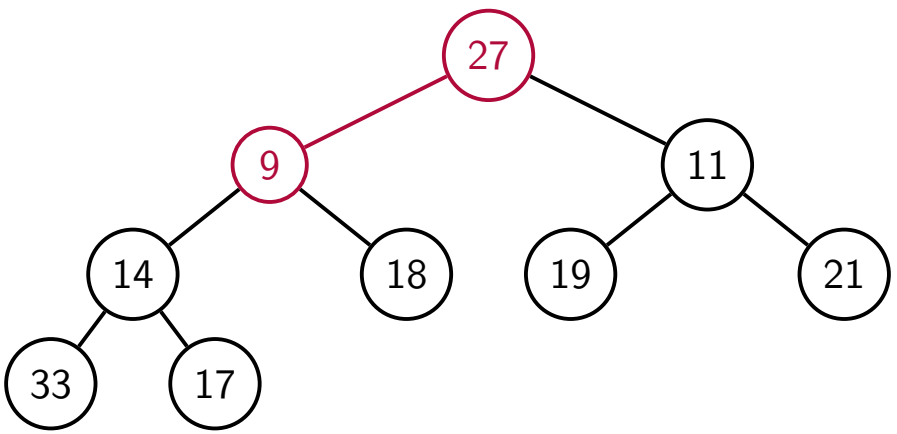


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau
2. Tant que la clef de v est supérieure à celle de l'un de ses fils :
   - On échange v avec le plus petit de ses fils

Exemple

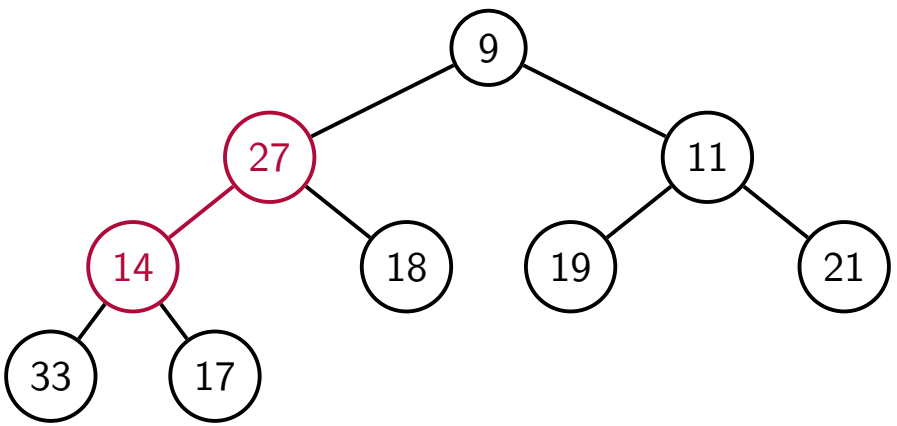


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau
2. Tant que la clef de v est supérieure à celle de l'un de ses fils :
   - On échange v avec le plus petit de ses fils

Exemple

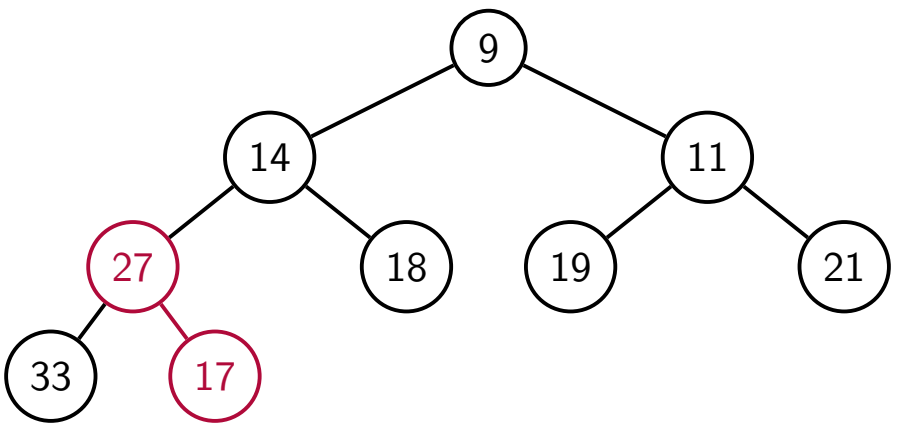


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau
2. Tant que la clef de v est supérieure à celle de l'un de ses fils :
   - On échange v avec le plus petit de ses fils

Exemple

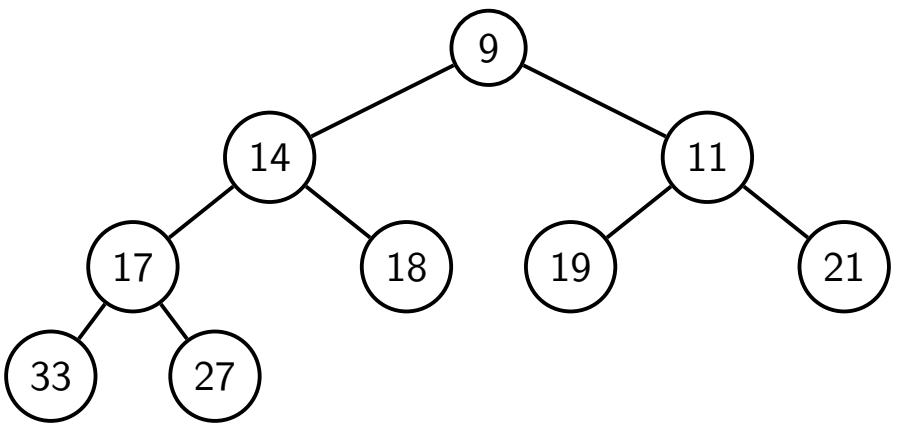


Extraire le min d'un Tas-min

Pour enlever la racine :

1. On remplace la racine par le dernier élément v de l'arbre
   - le dernier élément du tableau
2. Tant que la clef de v est supérieure à celle de l'un de ses fils :
   - On échange v avec le plus petit de ses fils

Exemple



Tas : complexité des opérations

La hauteur d'un tas binaire de n éléments est $\log_2 n$

Complexité

Tas : complexité des opérations

La hauteur d'un tas binaire de n éléments est $\log_2 n$

Complexité

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$

Tas : complexité des opérations

La hauteur d'un tas binaire de n éléments est $\log_2 n$

Complexité

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✓ Insérer un élément : $\mathcal{O}(\log n)$

Tas : complexité des opérations

La hauteur d'un tas binaire de n éléments est $\log_2 n$

Complexité

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✓ Insérer un élément : $\mathcal{O}(\log n)$
- ✓ Retirer l'élément minimum : $\mathcal{O}(\log n)$

Tas : complexité des opérations

La hauteur d'un tas binaire de n éléments est $\log_2 n$

Complexité

- ✓ Rechercher l'élément minimum : $\mathcal{O}(1)$
- ✓ Insérer un élément : $\mathcal{O}(\log n)$
- ✓ Retirer l'élément minimum : $\mathcal{O}(\log n)$
- ✓ Diminuer la clef d'un élément : $\mathcal{O}(\log n)$
  - on le remonte dans l'arbre (similaire à l'insertion)
  - correspond au `update` dans l'algorithme de Dijkstra

File de priorité : synthèse des complexités

Operation	*Array*	*Sorted Array*	*Binary Heap*
get min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert (*update*)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
extract min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

En passant de n à $log(n)$, on gagne beaucoup...

Pour $n = 1000$, $log_2(n) \simeq 10$; pour $n = 10^9$, $log_2(n) \simeq 30$!

Construction d'un Tas

Question

Comment construire un tas à partir d'une liste ?

Construction d'un Tas

Question

Comment construire un tas à partir d'une liste ?

Attention !

Pour construire un tas-min à partir d'un tableau non-trié, appliquer l'insertion n fois n'est pas optimal.
On peut faire mieux que $\mathcal{O}(n.log(n))$!

Construction d'un Tas

Question

Comment construire un tas à partir d'une liste ?

Attention !

Pour construire un tas-min à partir d'un tableau non-trié, appliquer l'insertion n fois n'est pas optimal.
On peut faire mieux que $\mathcal{O}(n.log(n))$!

Exercice

- **Indice** : on commence par la fin du tableau non trié et on remonte...

➜ Au final la complexité est en $\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \mathcal{O}(h) = \mathcal{O}(n)$!
(plus rapide que le tri)

Plan

Rappel de l'algorithme des plus courts chemins

```
def shortest_path(graph,s):
    frontier = [s]
    parent = {}
    parent[s] = None
    dist = {}
    dist[s] = 0

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):
            if y not in parent:
                frontier.append(y)
            new_dist = dist[x] + distance(graph,x,y)
            if y not in dist or dist[y] > new_dist:
                dist[y] = new_dist
                parent[y] = x

    return parent
```

Prenons un peu de recul...

```
def shortest_path(graph,s):
    # Initialisation

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):

            # mise a jour frontier, parent et dist

    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):
    # Initialisation                                    ← O(1)

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):

            # mise a jour frontier, parent et dist

    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):

    # Initialisation                                    ← O(1)


    while len(frontier)>0:
    on va le faire |V| fois (un sommet retiré à chaque tour)
        x = extract_min_dist(frontier,dist)

        for y in neighbors(graph, x):


            # mise a jour frontier, parent et dist


    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):

    # Initialisation                                    ← O(1)


    while len(frontier)>0:
    on va le faire |V| fois (un sommet retiré à chaque tour)
        x = extract_min_dist(frontier,dist)
                                                  complexité C_extract_min
        for y in neighbors(graph, x):


            # mise a jour frontier, parent et dist


    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):

    # Initialisation                                    ← O(1)


    while len(frontier)>0:
    on va le faire |V| fois (un sommet retiré à chaque tour)
        x = extract_min_dist(frontier,dist)
                                                  complexité C_extract_min
        for y in neighbors(graph, x):
          Là c'est plus subtil...on passe ici seulement |E| fois
          (on ne met à jour que lorsqu'un nouvel arc est visité)
          # mise a jour frontier, parent et dist


    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):

    # Initialisation                                    ← O(1)


    while len(frontier)>0:
    on va le faire |V| fois (un sommet retiré à chaque tour)
        x = extract_min_dist(frontier,dist)
                                                  complexité C_extract_min
        for y in neighbors(graph, x):
          Là c'est plus subtil...on passe ici seulement |E| fois
          (on ne met à jour que lorsqu'un nouvel arc est visité)
          # mise a jour frontier, parent et dist
                                                  complexité C_update

    # Calcul du chemin resultat
```

Prenons un peu de recul...

```
def shortest_path(graph,s):

    # Initialisation                                    ← O(1)


    while len(frontier)>0:
    on va le faire |V| fois (un sommet retiré à chaque tour)
        x = extract_min_dist(frontier,dist)
                                                        complexité C_extract_min
        for y in neighbors(graph, x):
          Là c'est plus subtil...on passe ici seulement |E| fois
          (on ne met à jour que lorsqu'un nouvel arc est visité)
          # mise a jour frontier, parent et dist
                                                        complexité C_update

    # Calcul du chemin resultat                         ← O(|V|)
```

En résumé...

Complexité de l'algorithme des plus courts chemins

$$\mathcal{O}(1 + |V| \times C_{extract\_min} + |E| \times C_{update} + |V|)$$

En résumé...

Complexité de l'algorithme des plus courts chemins

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

En résumé...

Complexité de l'algorithme des plus courts chemins

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

Implémentation

Les valeurs de $C_{extract\_min}$ et C_{update} dépendent de l'implémentation de la frontière.

En résumé...

Complexité de l'algorithme des plus courts chemins

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

Implémentation

Les valeurs de $C_{extract\_min}$ et C_{update} dépendent de l'implémentation de la frontière.

Implémentation de la frontière : file prioritaire

- implémentation naïve avec une simple liste
- implémentation avec tas binaire

Implémentation naïve

Complexité : $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

avec :

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

La frontière contient au pire $|V|$ éléments.

Implémentation naïve

Complexité : $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

avec :

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

La frontière contient au pire $|V|$ éléments.

Complexité : $\mathcal{O}(|V| \times |V| + |E|)$

avec $|E|$ compris entre 0 et $|V|^2$ (graphe dense)

Implémentation naïve

Complexité : $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

avec :

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

La frontière contient au pire $|V|$ éléments.

Complexité : $\mathcal{O}(|V| \times |V| + |E|)$

avec $|E|$ compris entre 0 et $|V|^2$ (graphe dense)

En pratique...

➜ La complexité de l'algorithme des plus courts chemins est en $\mathcal{O}(|V|^2)$

Implémentation naïve

Complexité : $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

avec :

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

La frontière contient au pire $|V|$ éléments.

Complexité : $\mathcal{O}(|V| \times |V| + |E|)$

avec $|E|$ compris entre 0 et $|V|^2$ (graphe dense)

En pratique...

➜ La complexité de l'algorithme des plus courts chemins est en $\mathcal{O}(|V|^2)$

Mais s'il y a peu d'arêtes, on peut peut-être faire mieux !

Implémentation avec tas binaire

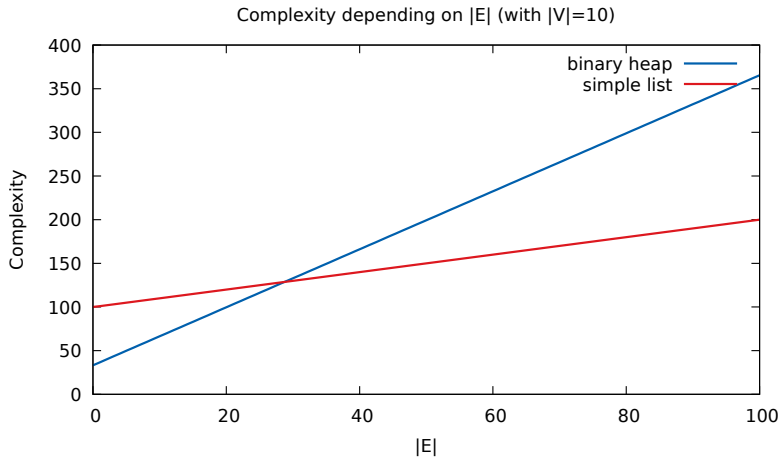
En utilisant une structure de données arborescente (tas binaire) on obtient :

- $C_{extract\_min} = \mathcal{O}(log(|V|))$
- $C_{update} = \mathcal{O}(log(|V|))$

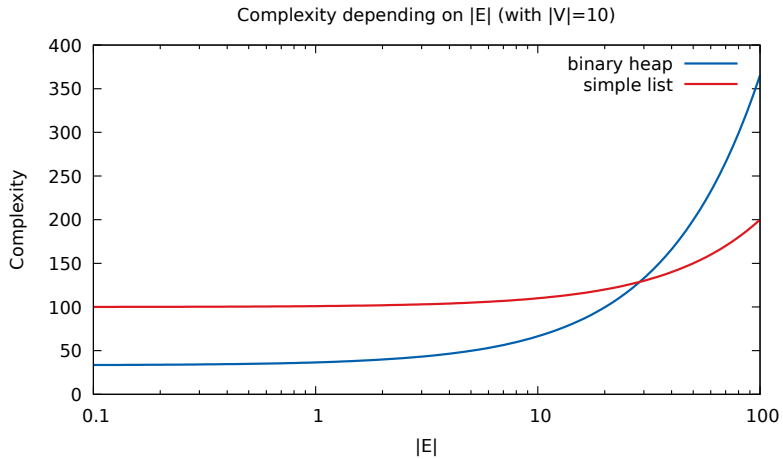
La complexité de l'algorithme des plus courts chemins est alors :

$$\mathcal{O}((|V| + |E|) \times log(|V|))$$

Liste simple vs Tas binaire



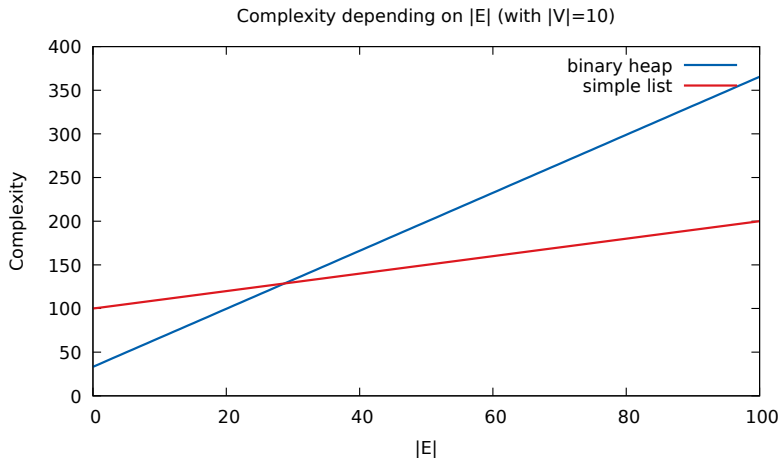
(a) Échelle linéaire



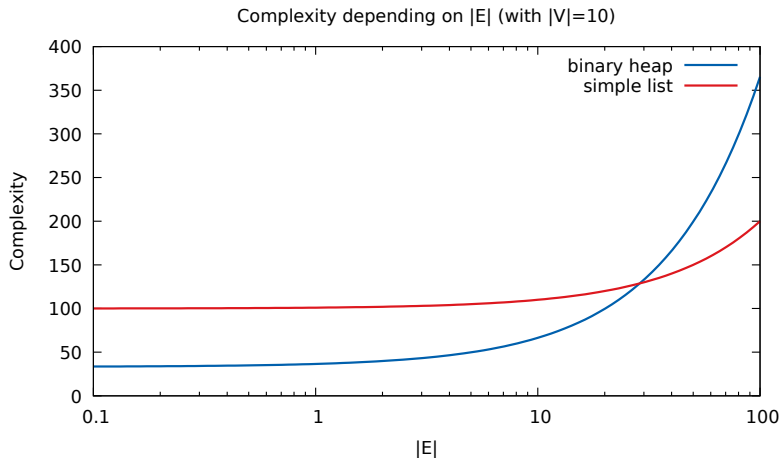
(b) Échelle logarithmique

Figure – Comparaison des complexités selon la densité du graphe (donnée par $|E|$) pour $|V| = 10$ fixé.

Liste simple vs Tas binaire



(a) Échelle linéaire



(b) Échelle logarithmique

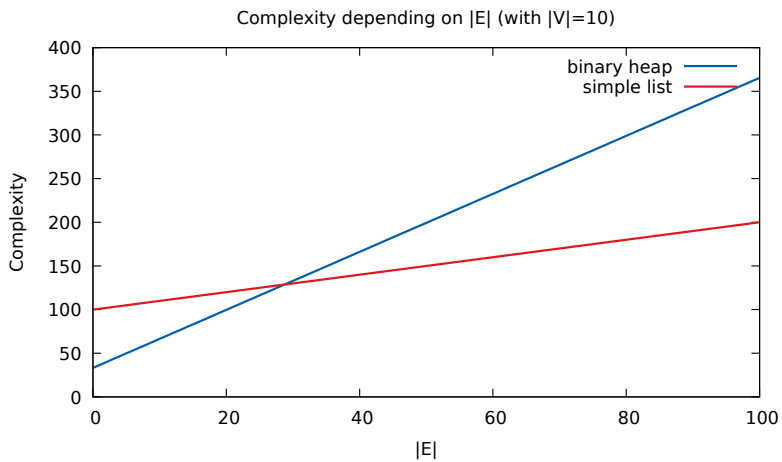
Figure – Comparaison des complexités selon la densité du graphe (donnée par $|E|$) pour $|V| = 10$ fixé.

En théorie

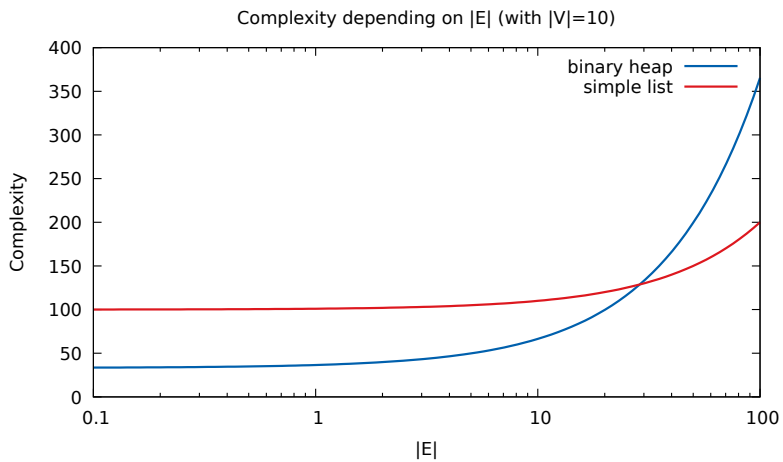
➜ Quand $|V|^2 + |E|$ est il moins intéressant que $(|V| + |E|) \times log(|V|)$?

Cela dépend de la densité du graphe : ($|E|$ par rapport à $|V|$)

Liste simple vs Tas binaire



(a) Échelle linéaire



(b) Échelle logarithmique

Figure – Comparaison des complexités selon la densité du graphe (donnée par $|E|$) pour $|V| = 10$ fixé.

En pratique

Nous verrons en TP que $\mathcal{O}(|E|)$ pour la mise à jour est très surestimée. . .et que le tas binaire s'en tire mieux que prévu !

Plan

Ce qu'il faut retenir

- Graphe orienté : $G = (V, E)$ et fonction de pondération $\omega : E \to \mathbb{R}$
- Algorithme des plus courts chemins
  - BFS légèrement modifié ;
  - Plus court chemin entre s et tous les autres sommets ;
- Complexité dépend des structures de données choisies pour l'implémentation
  - Complexité "naïve" (avec une simple liste) en $\mathcal{O}(|V|^2 + |E|)$
  - Complexité avec tas binaire en $\mathcal{O}((|V| + |E|) \times log(|V|))$

  ➜ Gain variable selon les instances !
- On peut prouver que cet algorithme est correct (la solution trouvée est optimale)

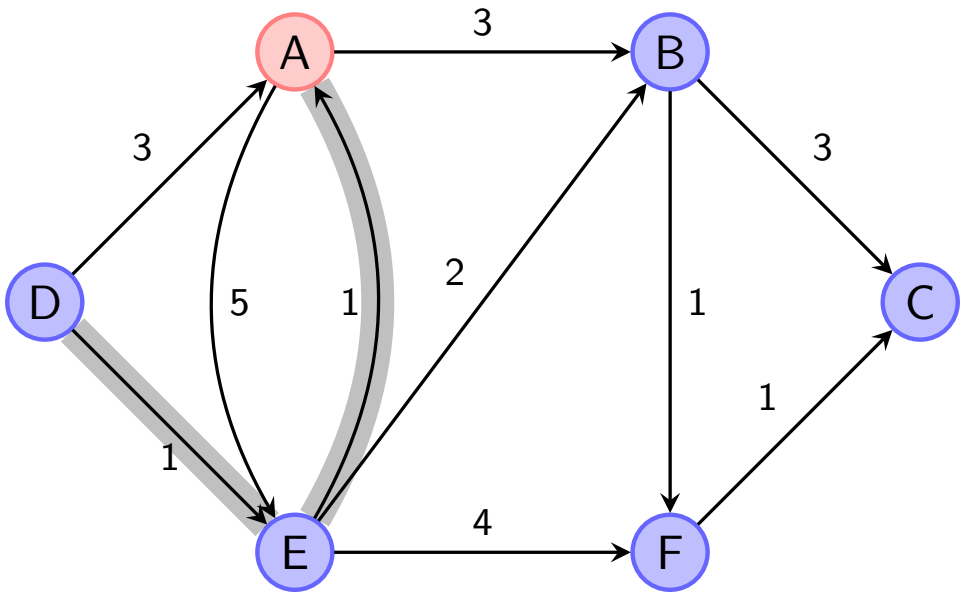
Plan

Optimalité

Propriété

Dès qu'un nœud sort de la frontière, il a son plus court chemin

Ex : la distance de E dans notre cas n'est pas révisée lorsqu'on visite A



Optimalité

Propriété

Dès qu'un nœud sort de la frontière, il a son plus court chemin

Ex : la distance de E dans notre cas n'est pas révisée lorsqu'on visite A

Plus formellement...

On définit un invariant de boucle :

À chaque tour n, soit S_n l'ensemble des nœuds déjà visités

(par construction, $|S_n| = n$)

1. $\forall x \in S_n$, $distance(x)$ est la longueur du chemin le plus court dans G
2. $\forall x \notin S_n$, $distance(x)$ est la longueur du chemin le plus court dans le sous-graphe $S_n \cup \{x\}$

La propriété est un corollaire de cet invariant

Idée de la preuve

Preuve par récurrence

On montre l'invariant de boucle par récurrence sur n

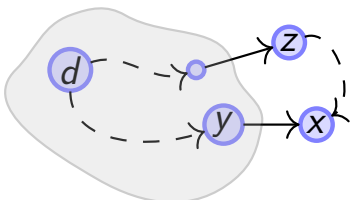
Idée de la preuve

Preuve par récurrence

On montre l'invariant de boucle par récurrence sur n

Que dit l'invariant ?

- Tous les nœuds dans S_n ont la valeur définitive de $distance(x)$
- Tous les voisins x (*i.e.* la frontière) ont leur distance minimale dans $S_n \cup \{x\}$



$\rightarrow$ Le chemin par z est plus long (cette partie de la preuve, plus difficile, repose sur le fait que les poids sont positifs)

- Les autres sont à $+\infty$

Idée de la preuve

Preuve par récurrence

On montre l'invariant de boucle par récurrence sur n

Conclusion de l'invariant

Les nœuds de S_n (sortis de la frontière) ont leur plus court chemin.

⏩ skip details

Preuve d'optimalité I

Preuve par récurrence

On montre l'invariant de boucle par récurrence sur n

$n = 1$

$S_1 = \{d\}$ et ses voisins ont pour $distance(x)$ le poids de l'arc.

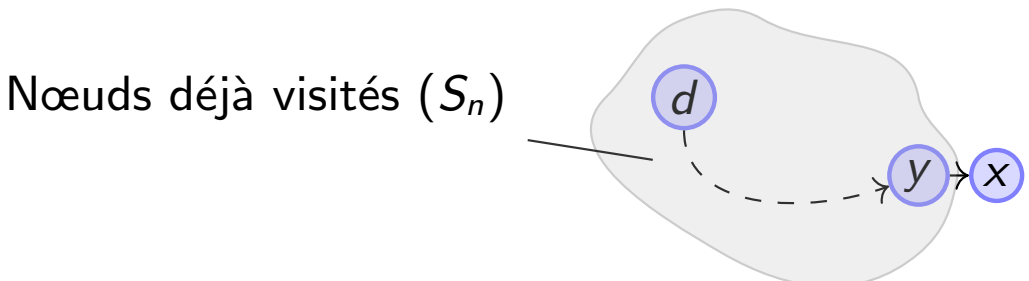
1. $distance(d) = 0$ est minimale (distances positives)
2. $\forall v$ voisin de d, $distance(v) = \omega((d, v))$ est minimale dans le sous-graphe $\{d, v\}$

Preuve d'optimalité II

Cas général

On suppose l'hypothèse vraie au tour n.
Soit x le nœud choisi par l'algorithme au tour $n+1$:

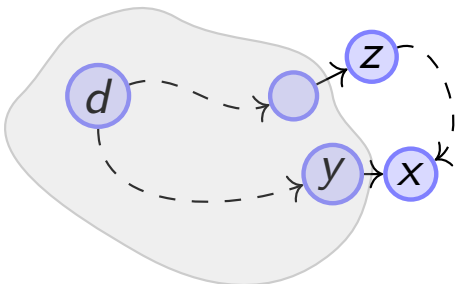
- Il est successeur de S_n (sinon $distance(x) = \infty$: il n'aurait pas été choisi)
- Il a la plus petite $distance(.)$
- On note y son prédécesseur dans S_n



Preuve d'optimalité II

Preuve de l'invariant par l'absurde

Considérons maintenant un autre chemin vers x et notons z le premier nœud non-visité sur ce chemin :



Nous allons montrer que ce chemin est au moins aussi long.

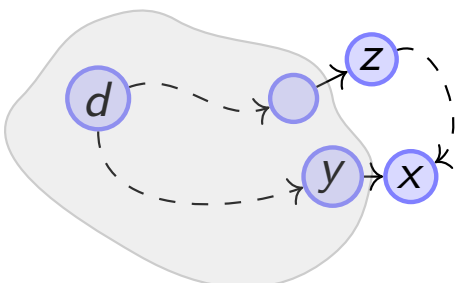
Preuve d'optimalité II

$distance(z)$ est minimal

$cout([d, \ldots, z]) \geq distance(z)$ puisque d'après l'hypothèse de récurrence 2, $distance(z)$ est minimal dans $S_n \cup \{z\}$

$distance(x)$ est minimal

Par définition, $distance(z) \geq distance(x)$ puisque x a été choisi et pas z



$$cout([d, \ldots, z]) \geq distance(x)$$

Preuve d'optimalité III

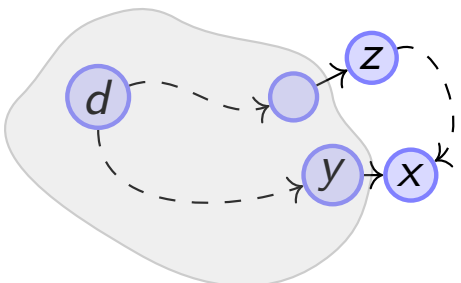
Coût complet

Or $cout([d, \ldots, z, \ldots, x]) = cout([d, \ldots, z]) + cout([z, \ldots, x])$ par définition

Donc $cout([d, \ldots, z, \ldots, x]) \geq cout([d, \ldots, z]) \geq distance(x)$

dans le cas où $cout([z, \ldots, x]) \geq 0$

(cette preuve d'optimalité repose sur le fait que le coût est positif !).



Tout chemin sortant de S_n est au moins aussi long.

Preuve d'optimalité IV

Preuve par récurrence

Si l'hypothèse est vraie au tour n, alors le nouveau nœud x vérifie aussi la propriété $distance(x) = dist(d, x)$

Invariant :

- ✓ $\forall x \in S_{n+1}$, $distance(x)$ est minimale dans G
- ➜ $\forall y \notin S_{n+1}$, $distance(y)$ est minimale dans $S_{n+1} \cup \{y\}$

Deuxième partie

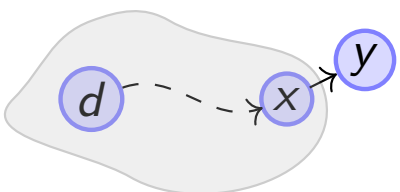
Il faut considérer tous les successeurs y de x qui ne sont pas dans S_n

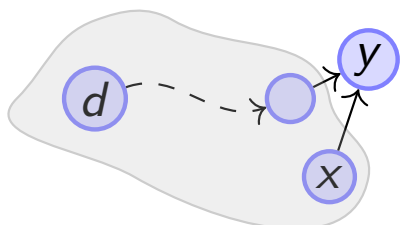
Preuve d'optimalité V

Deuxième partie (suite)

Considérons le plus court chemin pour y dans S_{n+1} :

- S'il passe par x, celui-ci est à la fin (car x a son plus court chemin dans S_{n+1}).
  Alors $distance(y) = distance(x) + \omega((x, y))$ est minimale
- S'il ne passe pas par x, $distance(y)$ était correct au tour précédent.
  Alors $distance(y) \leq distance(x) + \omega((x, y))$ (sinon c'est que le plus court chemin passe par x) donc $distance(y)$ n'a pas été modifiée et la propriété reste vraie dans S_{n+1}





Preuve d'optimalité VI

Conclusion

Si l'hypothèse vraie au tour n, alors elle l'est au rang $n+1$
Invariant :

- ✓ $\forall x \in S_{n+1}$, $distance(x)$ est minimale dans G
- ✓ $\forall y \notin S_{n+1}$, $distance(y)$ est minimale dans $S_{n+1} \cup \{y\}$