Graphes II

CHAPITRE 9(WEISS)

NOTES DE COURS, É. BAUDRY

NOTES DE COURS, S. HAMEL

Graphes

- Recherche de chemins
- Algorithme de Dijkstra
- Permet de trouver un chemin optimal
  - La distance la plus courte reliant une paire de sommets dans un graphe connexe (fortement connexe)
- Permet de calculer un arbre de Chemins à partir d'un sommet de départ vers tous les autres sommets
  - Assume que les distances (poids des arrêtes) sont positives

Algorithme de Dijkstra

- Calcul incrémental des distances minimales vers chacun des sommets
- À la fin du calcul, chaque élément du tableau distances[i] contient la distance minimale entre le sommet de départ s et un sommet de destination i

Algorithme de Dijkstra

- Les distances sont initialisées à +∞ (ligne 2), à l'exception du sommet de départ à 0 (ligne 5). À la fin du calcul, chaque case du tableau parents[i] contient un pointeur (indice) vers le sommet parent dans l'arbre de chemins les plus courts

```
1. DIJKSTRA(G = (V,E); s ∈ V)
2.   pour tout v ∈ V
3.       distances[v] ← +∞
4.       parents[v] ← indéfini
5.   distances[s] ← 0
6.   Q ← créer FilePrioritaire(V)
7.   tant que ¬Q.vide()
8.       v ← Enlever v ∈ Q avec min distances[v]
9.       si distances[v] = + ∞ break
10.      pour toute arête sortante e = (v,w) depuis v
11.          d ← distances[v]+e.distance
12.          si d < distances[w]
13.              parents[w] ← v
14.              distances[w] ← d
15.              Q.insérer(w)
16. retourner (distances, parents)
```

Algorithme de Dijkstra

Initialement, les parents sont indéfinis

- Le cœur de l'algorithme repose sur la boucle à la ligne 7
- Boucle itère sur les sommets insérés dans la file prioritaire Q afin de mettre à jour le tableau de distances
- Pour chaque sommet v, on itère sur les arêtes sortantes e = (v, w) pour visiter tous les sommets w accessibles à partir de v

```
1. DIJKSTRA(G = (V,E); s ∈ V)
2.   pour tout v ∈ V
3.       distances[v] ← +∞
4.       parents[v] ← indéfini
5.   distances[s] ← 0
6.   Q ← créer FilePrioritaire(V)
7.   tant que ¬Q.vide()
8.       v ← Enlever v ∈ Q avec min distances[v]
9.       si distances[v] = + ∞ break
10.      pour toute arête sortante e = (v,w) depuis v
11.          d ← distances[v]+e.distance
12.          si d < distances[w]
13.              parents[w] ← v
14.              distances[w] ← d
15.              Q.insérer(w)
16. retourner (distances, parents)
```

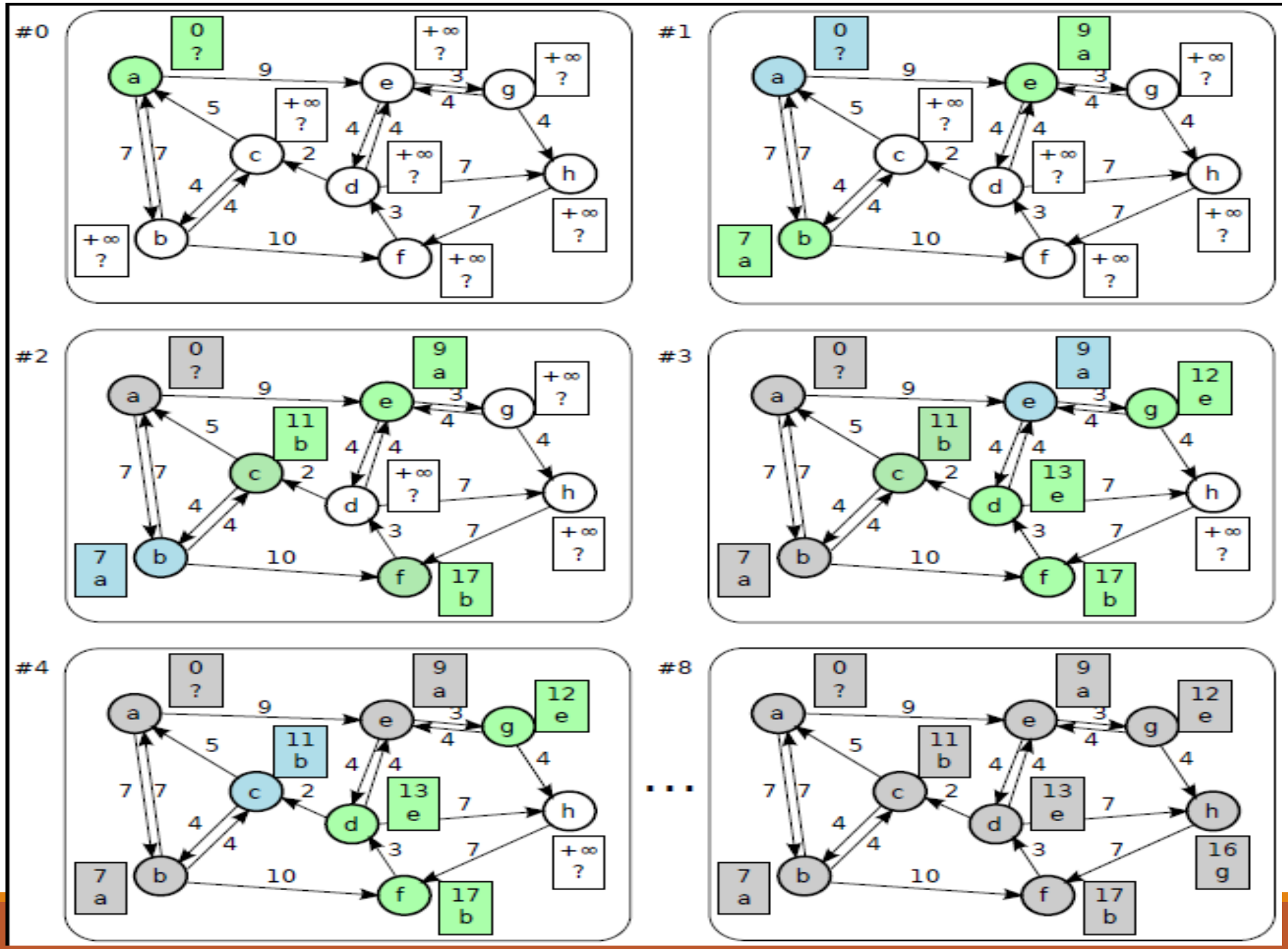
Algorithme de Dijkstra

La ligne 11 calcule la distance d pour se rendre au sommet w en passant par v. Si cette distance est meilleure que celle déjà présente dans la case distances[i], alors un meilleur chemin a été trouvé. On met à jour parents[i] et distances[i]. Il faudra revérifier si d'autres distances vers des sommets à partir de w peuvent être améliorées. Ainsi, à la ligne 15, on ajoute w à Q.

```
1. DIJKSTRA(G = (V,E); s ϵ V)
2.   pour tout v ϵ V
3.       distances[v] ← +∞
4.       parents[v] ← indéfini
5.   distances[s] ← 0
6.   Q ← créer FilePrioritaire(V)
7.   tant que ¬Q.vide()
8.       v ← Enlever v ϵ Q avec min distances[v]
9.       si distances[v] = + ∞ break
10.      pour toute arête sortante e = (v,w) depuis v
11.          d ← distances[v]+e.distance
12.          si d < distances[w]
13.              parents[w] ← v
14.              distances[w] ← d
15.              Q.insérer(w)
16. retourner (distances, parents)
```

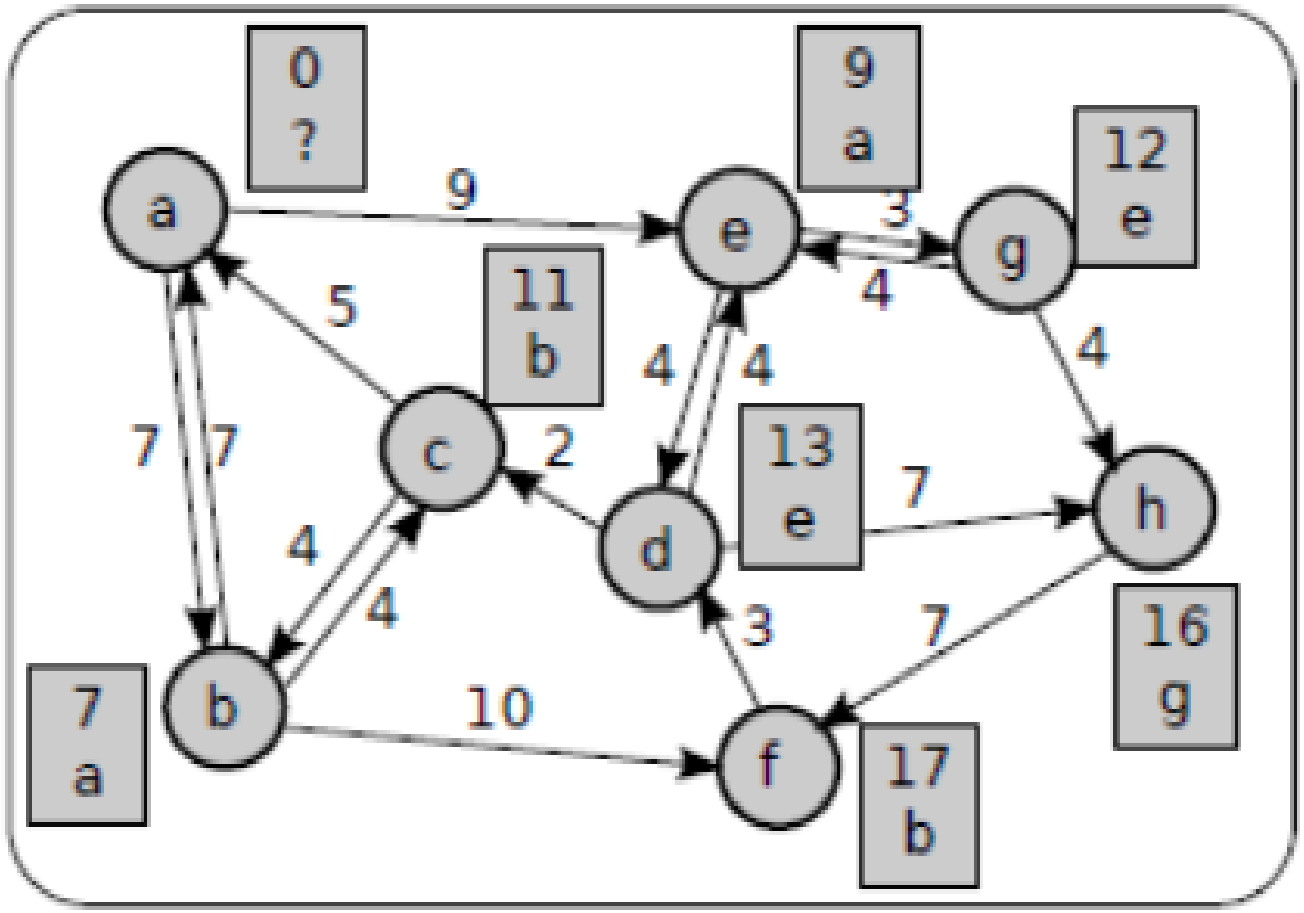
Algorithme de Dijkstra

```
1. DIJKSTRA(G = (V,E); s ϵ V)
2.   pour tout v ϵ V
3.       distances[v] ← +∞
4.       parents[v] ← indéfini
5.   distances[s] ← 0
6.   Q ← créer FilePrioritaire(V)
7.   tant que ¬Q.vide()
8.       v ← Enlever v ϵ Q avec la plus petite valeur distances[v]
9.       si distances[v] = + ∞ break
10.      pour toute arête sortante e = (v,w) depuis le sommet v
11.          d ← distances[v]+e.distance
12.          si d < distances[w]
13.              parents[w] ← v
14.              distances[w] ← d
15.              Q.insérer(w)
16. retourner (distances, parents)
```


#0
#1
#2
#3
#4
#8
...

Algorithme de Dijkstra

- Chemin de a à h
  - distance est 16
  - h←g←e←a



Analyse de Dijkstra

- Complexité temporelle de l'algorithme de Dijkstra s'exprime en fonction du nombre de sommets |V| et d'arêtes |E|
- Dans le pire cas, l'algorithme doit visiter une fois tous les sommets et toutes les arêtes

Analyse de Dijkstra

- La file prioritaire Q joue un rôle central
- Tous les sommets dans V doivent être extrait de Q
- Les sommets peuvent être réordonnés (remontés en priorité) dans Q à chaque fois qu'une arête permet la découverte d'un meilleur coût (ligne 12)
- Ainsi, on a n retraits du sommet minimum et m ré ordonnancements

Analyse de Dijkstra

- La complexité temporelle dépend de l'implémentation de la file prioritaire Q
- File prioritaire basée sur un monceau
- L'insertion et l'enlèvement de l'élément minimal
  - O(log|V|)
- Complexité temporelle de Dijkstra
  - **O(|V|log|V|+|E|log|V|)**

Analyse de Dijkstra

- Utiliser une structure de données plus efficace pour la file prioritaire
  - Monceau de Fibonacci
    - Insertion en temps constant et l'enlèvement en temps logarithmique

Ainsi, la complexité temporelle descend à

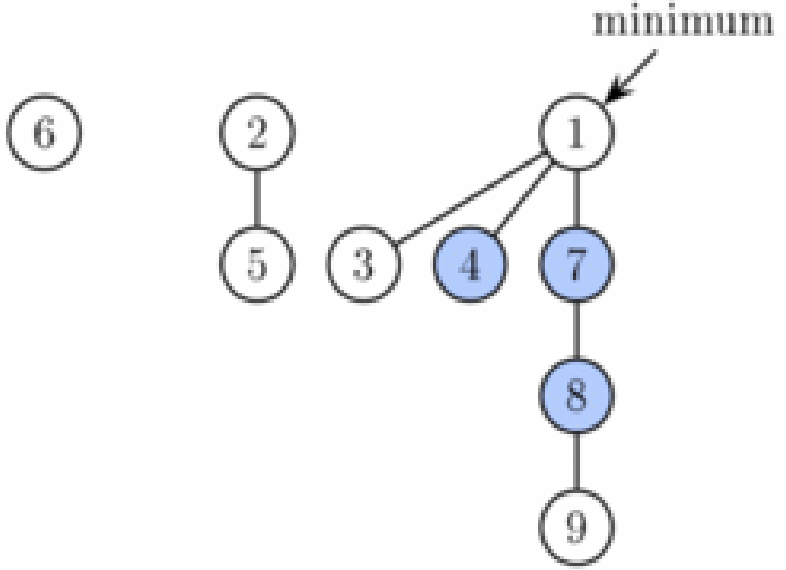
O(|V|**log**|V|**+** |E|**)**

Monceau de Fibonacci

- Monceau de Fibonacci est une structure de données similaire au monceau binomial, mais avec un meilleur temps d'exécution amorti

- Le nom de monceau de Fibonacci vient des nombres de Fibonacci, qui sont utilisés pour calculer son temps d'exécution.

Monceau de Fibonacci



- Monceau de Fibonacci est un ensemble d'arbres satisfaisant la propriété de tas-minimum
  - Clé d'un fils est toujours supérieure ou égale à la clé de son père
  - Clé minimum est toujours à la racine d'un des arbres

- Opérations insertion, trouver le minimum, supprimer le minimum + **union**

Opérations

- Opérations insertion, trouver le minimum et union ont toutes un coût amorti constant

- Opérations supprimer le minimum ont un coût amorti en O(log n)

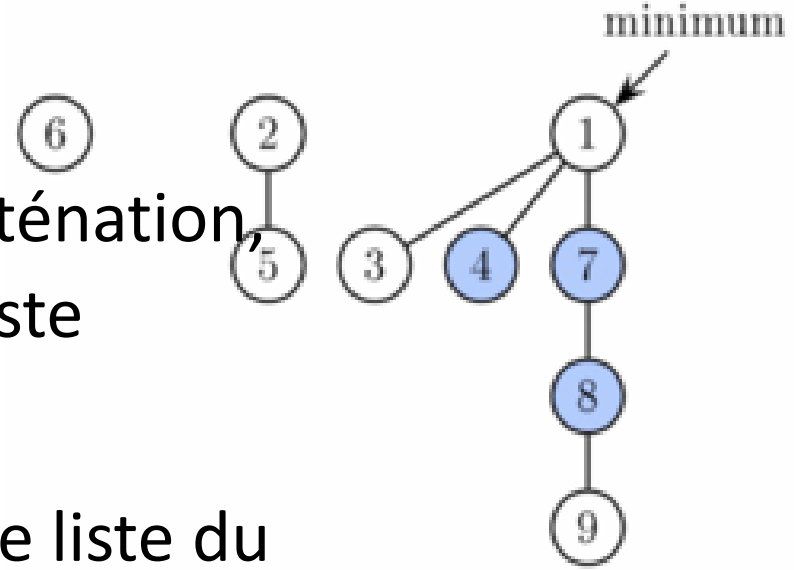
Opérations

- Union de deux tas est effectuée simplement en concaténant les deux listes d'arbres
  - Cependant, à un moment donné il est nécessaire d'introduire un certain ordre dans la structure du monceau de manière à obtenir le temps d'exécution voulu

Opérations

- Ordonner la structure
  - Garde les degrés des nœuds (nombre de fils) en dessous d'une valeur assez basse (au plus en O(log n))
  - Garder la taille d'un sous-arbre d'un nœud de degré k est au moins Fk + 2, où Fk est le kème nombre de la suite de Fibonacci
    - K = 2 => Fk+2 = F2+2 = 3
    - Nombre d'arbres est diminué par l'opération supprimer le minimum, dans laquelle on relie les arbres

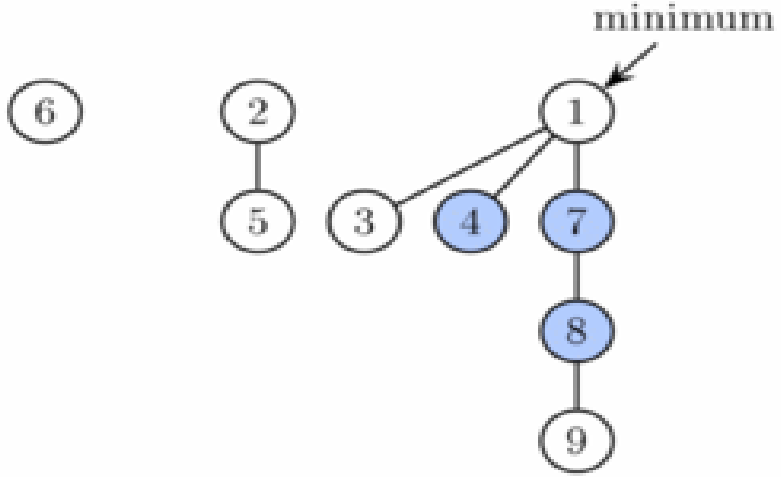
Implémentation des opérations



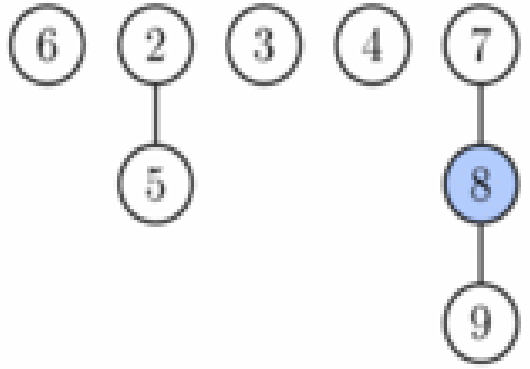
- Pour permettre la suppression rapide et la concaténation, les racines de tous les arbres sont liées par une liste doublement chaînée circulaire
  - Les fils de chaque nœud sont aussi liés par une liste du même type
  - Pour chaque nœud, on conserve des informations sur son nombre de fils et sur son marquage éventuel
  - On garde un pointeur sur la racine contenant la clé minimale

Implémentation des opérations

- L'opération trouver le minimum est triviale
  - On garde un pointeur sur son nœud
- Supprimer le minimum
  - On prend la racine contenant l'élément minimum et on la supprime. Ses fils vont devenir les racines de nouveaux arbres

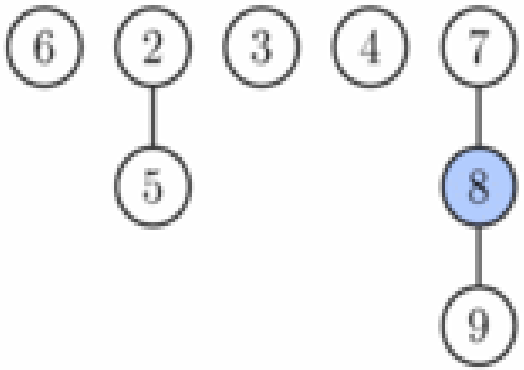


Après la première phase de l'extraction du minimum.

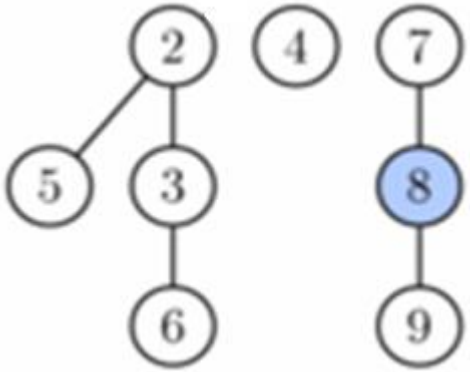


Implémentation des opérations

- Supprimer le minimum, Seconde phase
- On décroît le nombre de racines en connectant successivement les racines de même degré
  - Quand deux racines *u* et *v* ont le même degré, on met celle avec la plus grande clé des deux en fils de celle ayant la plus petite clé. Cette dernière voit son degré augmenter de un. On répète ceci jusqu'à ce que chaque racine ait un degré différent. Pour trouver efficacement les arbres de même degré, on utilise un tableau de longueur $O(\log n)$ dans lequel on garde un pointeur sur une racine de chaque degré. Quand on trouve une seconde racine avec le même degré, on connecte les deux et on met à jour le tableau
  - Le temps d'exécution réel est en $O(\log n + r)$, où r est le nombre de racines au début de la seconde phase.

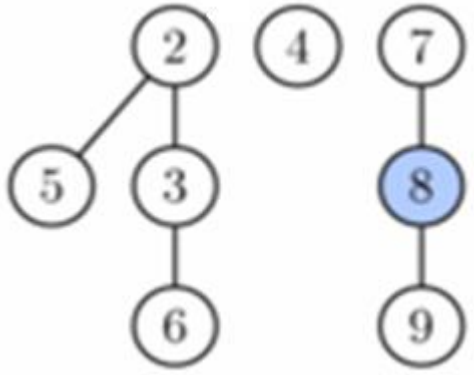


Nœuds 3 et 6 sont connectés, puis le résultat est connecté avec l'arbre de racine 2.

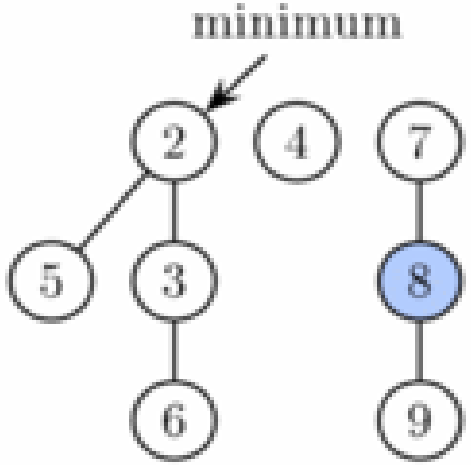


Implémentation des opérations

- Supprimer le minimum
  - Dans la troisième phase, on regarde chacune des racines restantes et on trouve le minimum, ce qui est fait en $O(\log n)$
  - À la fin, on aura au plus $O(\log n)$ racines (parce que chacune a un degré différent)



Enfin, le nouveau minimum est trouvé



Implémentation des opérations

- Union
  - est implémentée simplement en concaténant les listes des racines des arbres des deux tas. Cette opération peut être effectuée en temps constant

Résumé des temps d'exécution

	Liste chaînée	Tas	Tas de Fibonacci
insert	**O(1)**	**O(log n)**	**O(1)**
accessMin	**O(n)**	**O(1)**	**O(1)**
deleteMin	**O(n)**	**O(log n)**	**O(log n), temps amorti**
merge	**O(1)**	**O(m log(n+m))**	**O(1)**

Analyse de Dijkstra

- La complexité spatiale de Dijkstra est de O(|V|)
- Mémoire temporaire requise pour exécuter Dijkstra se limite aux deux tableaux distances et parents ayant chacun la taille du nombre de sommets

Adaptations à l'algorithme de Dijkstra

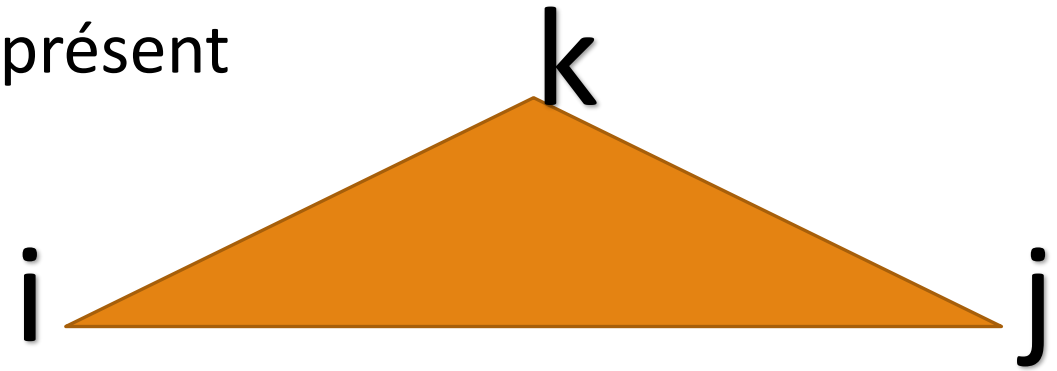
- Selon les applications, il est possible de faire quelques améliorations et adaptations
- Si on s'intéresse uniquement à une source et une seule destination, il est possible d'ajouter une condition à l'algorithme pour arrêter le calcul dès que la destination est visitée
  - Évite d'avoir à visiter tout le graphe en entier

Algorithme de Floyd-Warshall

- Calcule les meilleurs chemins pour toutes les paires de sommets d’un graphe
- Calculer de façon itérative la distance minimale entre chaque paire de sommets
- Les coûts des arêtes initialisent les distances connues

Algorithme de Floyd-Warshall

- Dans les 3 boucles imbriquées, on vérifie si le chemin passant par k pour se rendre de i à j est plus court que le meilleur chemin connu jusqu'à présent



- Si c'est le cas, on met à jour la nouvelle distance minimale trouvée
- Pour chaque paire, on note la direction à prendre

Algorithme de Floyd-Warshall

```
1. FLOYD_WARSHALL(G = (V, E))
2.    distances  ← créer tableau |V|*|V|initialisé à  + ∞
3.    directions  créer tableau |V|*|V| initialisé à « ?»
4.     pour tout v ϵ V
5.         distances[v][v] ← 0
6.         directions[v][v] ← v
7.      pour tout e ϵ E
8.          distances[e.out][e.in] ← e.cout
9.          directions[e.out][e.in] ←  e.in
10.     pour k = 0 à |V|-1
11.         pour i = 0 à |V|-1
12.             pour j = 0 à |V|-1
13.                 si distances[i][k]+distances[k][ j] < distances[i][ j]
14.                     distances[i][ j] ←  distances[i][k]+distances[k][ j]
15.                     directions[i][ j] ←  directions[i][k]
16. retourner directions
```