ALGORITHMIQUE ET STRUCTURES DE DONNÉES

3. Graphes

Yoann Coudert--Osmont

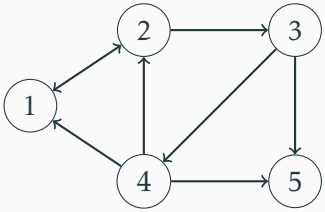
28 Janvier 2026

1. Définitions
2. Parcours de graphe
    - Parcours en largeur
    - Parcours en profondeur
3. Algorithmes de plus court chemin
    - Algorithme de Dijkstra
    - Algorithme de Bellman-Ford
4. Arbre couvrant de poids minimal
    - Algorithme de Kruskal
    - Algorithme de Prim

Définitions

Graphe (orienté)

Un graphe orienté $G = (S, A)$ est la donnée d'un ensemble de sommets S et d'un ensemble d'arcs $A \subseteq V \times V$.



$V = \{1, 2, 3, 4, 5\}$

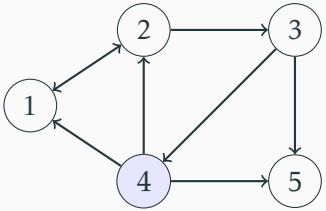
$A = \{(1, 2), (2, 1), (2, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 5)\}$

Notation

- On utilisera $n = |V|$ et $m = |A|$.

Notation

- On utilisera $n = |V|$ et $m = |A|$.
- On définit le **voisinage sortant** d'un sommet $x \in V$ comme $N^+(x) = \{y \mid (x, y) \in A\}$. Le **degré sortant** vaut $d^+(x) = |N^+(x)|$.
- On définit le **voisinage entrant** d'un sommet $x \in V$ comme $N^-(x) = \{y \mid (y, x) \in A\}$. Le **degré entrant** vaut $d^+(x) = |N^+(x)|$.

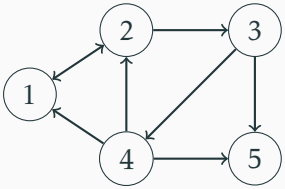


$N^+(4) = \{1, 2, 5\}, \quad d^+(4) = 3$

$N^-(4) = \{3\}, \quad d^-(4) = 1$

Représentation de graphe

Il existe deux façons classiques de représenter un graphe :



Graphe orienté

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	0	0
3	0	0	0	1	1
4	1	1	0	0	1
5	0	0	0	0	0

Matrice d'adjacence

1	2		
2	1	3	
3	4	5	
4	1	2	5
5	·		

Listes d'adjacence

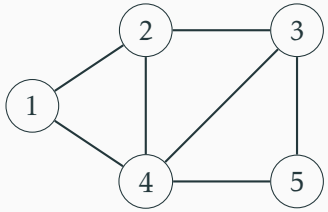
	Matrice d'adj.	Listes d'adj.
Mémoire	$\Theta(n^2)$	$\Theta(n+m)$
Test "$(x,y) \in A$?"	$\Theta(1)$	$\Theta(d^+(x))$

Graphe non orienté

Un graphe **non orienté** $G = (S, A)$ est un graphe dont chaque arc est bidirectionnel :

$$(x, y) \in A \iff (y, x) \in A$$

A est alors un ensemble de paires de sommets $\{x, y\}$ appelés **arêtes**.

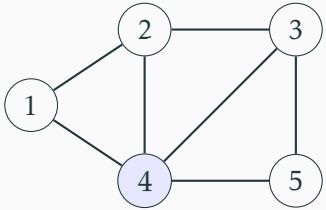


$V = \{1, 2, 3, 4, 5\}$

$A = \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\}$

Notation

On définit le **voisinage** d'un sommet $x \in V$ comme $N(x) = \{y \mid \{x, y\} \in A\}$.
Le **degré** d'un sommet est $d(x) = |N(x)|$.

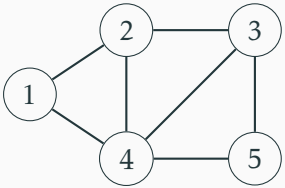


$N(4) = \{1, 2, 3, 5\}$

$d(4) = 4$

Représentation de graphe

Il existe deux façons classiques de représenter un graphe :



Graphe non orienté

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Matrice d'adjacence

1	2	4		
2	1	3	4	
3	2	4	5	
4	1	2	3	5
5	3	4		

Listes d'adjacence

	Matrice d'adj.	Listes d'adj.
Mémoire	$\Theta(n^2)$	$\Theta(n+m)$
Test "$\{x,y\} \in A$?"	$\Theta(1)$	$\Theta(d(x))$

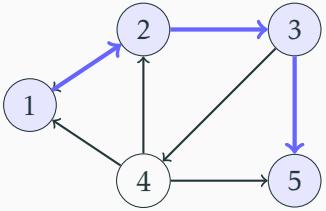
Chemin

Chemin

Dans un graphe $G = (S, A)$, un chemin c de longueur k est une séquences de $k+1$ sommets $c = \langle s_0, s_1, \ldots, s_k \rangle$ tel que :

$$\forall 0 \leqslant i < k,\ (s_i, s_{i+1}) \in A$$

Lorsque $s_0 = s$ et $s_k = t$, on dit que c est un chemin allant de s à t.

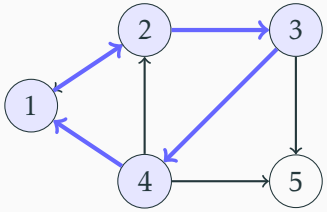


$\langle 1, 2, 3, 5 \rangle$ est un chemin de longueur 3

Cycle

Cycle

Dans un graphe $G = (S, A)$, un cycle c de longueur k est un chemin de longueur k : $c = \langle s_0, s_1, \ldots, s_k \rangle$, tel que le dernier sommet est égale au premier sommet.
C'est-à-dire, $s_0 = s_k$.

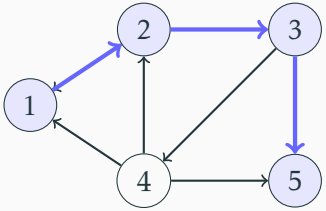


$\langle 1, 2, 3, 4, 1 \rangle$ est un cycle de longueur 4

Accessibilité

Soit $G = (S, A)$ un graphe. Un sommet t est dit accessible depuis un sommet s lorsqu'il existe un chemin allant de s à t dans G. On définit la relation d'accessibilité $\mathscr{R}_A$ par :

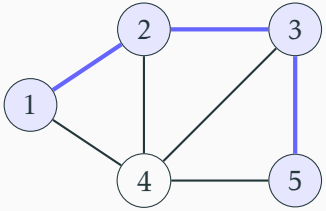
$$s \mathscr{R}_A t \iff t \text{ est accessible depuis } s$$



$\langle 1, 2, 3, 5 \rangle$ est un chemin. Donc $1 \mathscr{R}_A 5$.

Cas non orienté

Dans un graphe non orienté, si $c = \langle s_0, s_1, \ldots, s_k \rangle$ est un chemin, alors la séquence inverse $c = \langle s_k, s_{k-1}, \ldots, s_1, s_0 \rangle$ est également un chemin.



$\langle 1, 2, 3, 5 \rangle$ est un chemin. $\langle 5, 3, 2, 1 \rangle$ est son chemin inverse.

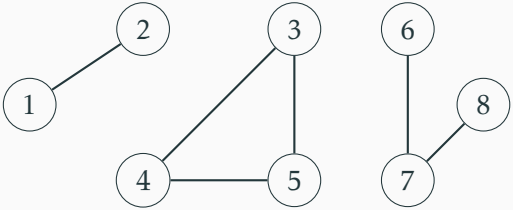
Dans un graphe non orienté, $\mathscr{R}_A$ est une relation symétrique :

$$s \mathscr{R}_A t \iff t \mathscr{R}_A s$$

Connexité

Soit $G = (S, A)$ un graphe non orienté.

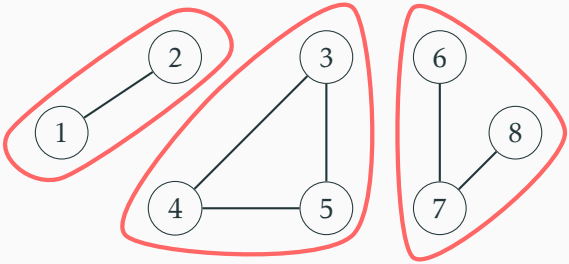
- Les classes d'équivalence de la relation $\mathscr{R}_A$ sont appelées **composantes connexes** de G.
- On dit que ***G* est connexe** s'il ne possède qu'une seule composante connexe. C'est-à-dire, lorsqu'il existe un chemin reliant n'importe quel paire de sommets.



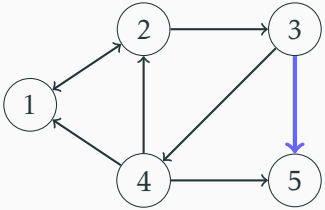
Connexité

Soit $G = (S, A)$ un graphe non orienté.

- Les classes d'équivalence de la relation $\mathscr{R}_A$ sont appelées **composantes connexes** de G.
- On dit que ***G* est connexe** s'il ne possède qu'une seule composante connexe. C'est-à-dire, lorsqu'il existe un chemin reliant n'importe quel paire de sommets.



Cas orienté



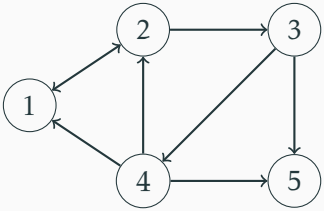
$3 \mathscr{R}_A 5$

3 n'est pas accessible depuis 5...

$\mathscr{R}_A$ n'est pas symétrique.

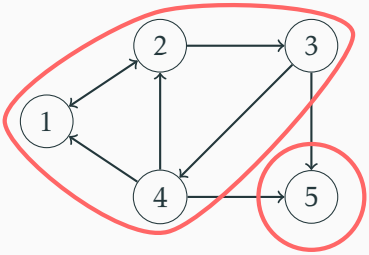
Forte connexité

On dit qu'un graphe orienté est fortement connexe si et seulement si il existe un chemin reliant n'importe quel paire de sommets.



Forte connexité

On dit qu'un graphe orienté est fortement connexe si et seulement si il existe un chemin reliant n'importe quel paire de sommets.

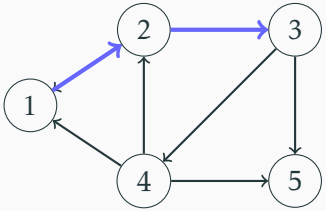


Distance entre sommets

Dans un graphe $G = (S, A)$, si t est un sommet accessible depuis s, alors l'ensemble $C(s,t)$ des chemins allant de s à t n'est pas vide. Dans ce cas, la distance $d(s,t)$ entre s et t est défini par :

$$d(s,t) = \min_{c \in C(s,t)} l(c)$$

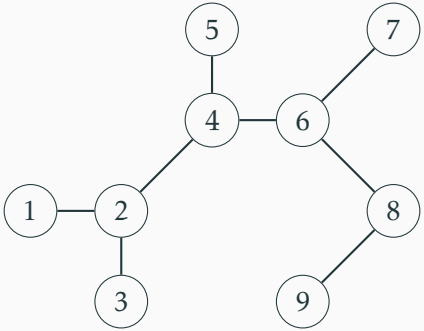
Où $l(c)$ est la longueur du chemin c.



$d(1,3) = 2$

Arbre

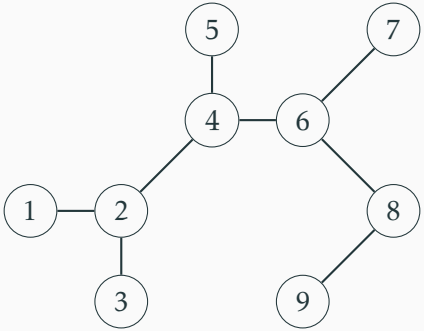
Un arbre est un graphe non orienté, connexe et sans cycles. Ses sommets sont appelés **nœuds**.



Arbre

Arbre

Un arbre est un graphe non orienté, connexe et sans cycles. Ses sommets sont appelés **nœuds**.



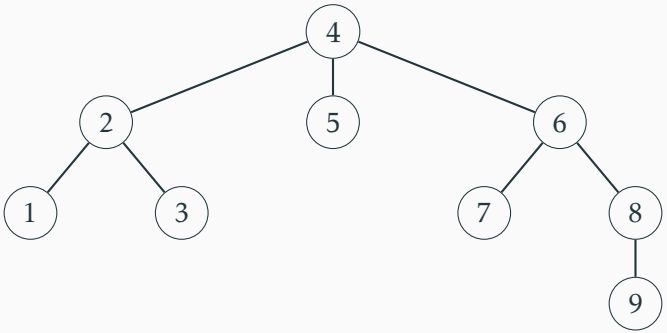
Théorème

Soit $G = (S, A)$ un arbre. Son nombre d'arêtes est $m = |A| = |S| - 1 = n - 1$.

Arbre enraciné

Un arbre enraciné est un arbre dont on a particularisé un nœud que l'on appelle racine de l'arbre.

Arbre précédent enraciné en 4 :

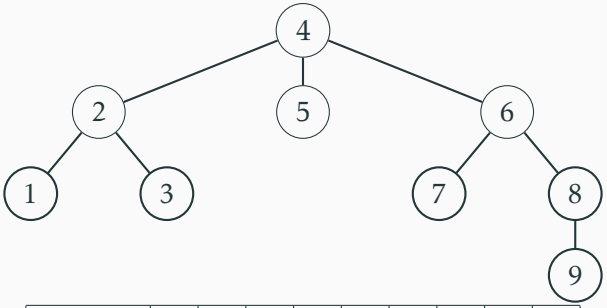


Représentation d'un arbre enraciné

Dans un arbre enraciné, pour chaque arête, on peut distinguer ses deux extrémités

- Le nœud le plus proche de la racine est appelé **père**.
- Le nœud le plus loin de la racine est appelé **fils**.

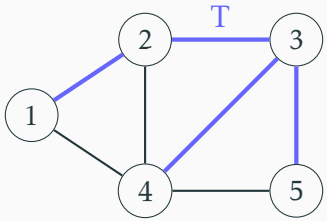
L'arbre peut être représenté par un tableau `parent` indiquant le père de chaque nœud.



nœud	1	2	3	4	5	6	7	8	9
parent	2	4	2	·	4	4	6	6	8

Arbre couvrant

Soit $G = (S, A)$ un graphe non orienté connexe. Un arbre couvrant de G, est un arbre $T = (S, A_T)$ dont les sommets sont ceux de G et donc les arêtes appartiennent à G ($A_T \subseteq A$).



Parcours de graphe

Parcours de graphe

Le but d'un parcours de graphe est d'explorer les sommets à partir d'un sommet source en parcourant les arêtes/arcs du graphe.

- **Entrée :** Un graphe G et un sommet source s.
- **Sortie :** Un arbre de parcours enraciné en s, et éventuellement d'autres informations.

Parcours de graphe

Le but d'un parcours de graphe est d'explorer les sommets à partir d'un sommet source en parcourant les arêtes/arcs du graphe.

- **Entrée :** Un graphe G et un sommet source s.
- **Sortie :** Un arbre de parcours enraciné en s, et éventuellement d'autres informations.

On distingue deux principaux types de parcours :

- Le **parcours en largeur** permettant par exemple de calculer les distances entre la source s et les autres sommets.

Parcours de graphe

Le but d'un parcours de graphe est d'explorer les sommets à partir d'un sommet source en parcourant les arêtes/arcs du graphe.

- **Entrée :** Un graphe G et un sommet source s.
- **Sortie :** Un arbre de parcours enraciné en s, et éventuellement d'autres informations.

On distingue deux principaux types de parcours :

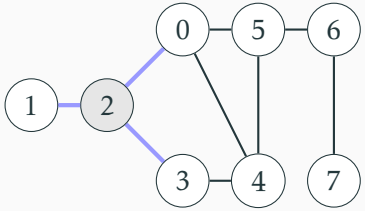
- Le **parcours en largeur** permettant par exemple de calculer les distances entre la source s et les autres sommets.
- Le **parcours en profondeur** permettant par exemple de tester la présence d'un cycle dans un graphe orienté.

Parcours de graphe

Parcours en largeur

Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.

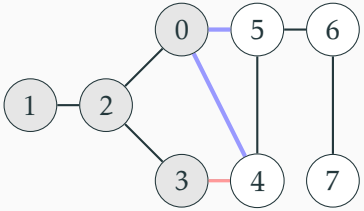


Étape 0 : $V = \{2\}$

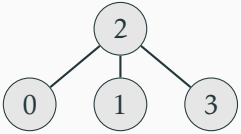
2

Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.

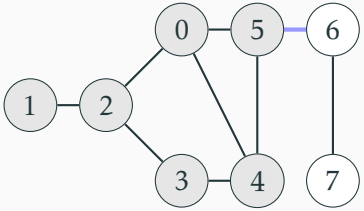


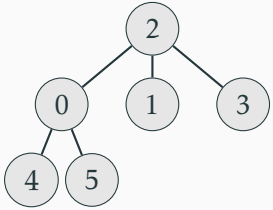
Étape 1 : $V = \{0, 1, 2, 3\}$



Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.

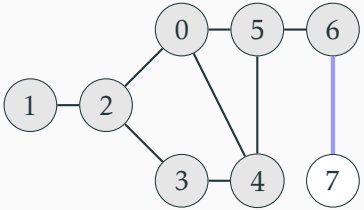




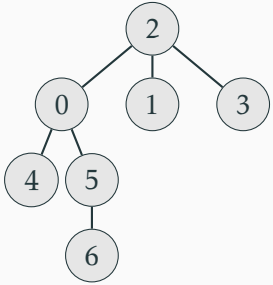
Étape 2 : $V = \{0, 1, 2, 3, 4, 5\}$

Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.

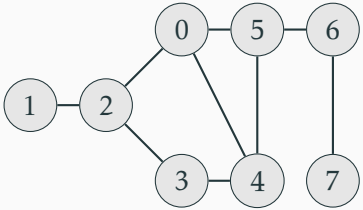


Étape 3 : $V = \{0, 1, 2, 3, 4, 5, 6\}$

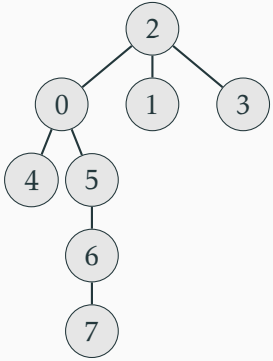


Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.

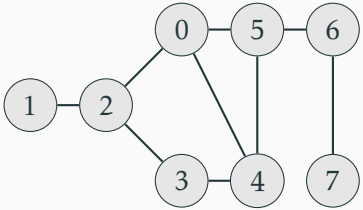


Étape 4 : $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$

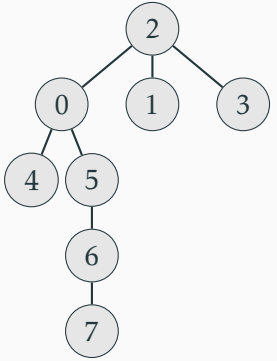


Parcours en largeur

Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.
À l'étape k, V contient tous les sommets à une distance au plus k de s.



Étape 4 : $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$



$d(2, \cdot) = 0$

$d(2, \cdot) = 1$

$d(2, \cdot) = 2$

$d(2, \cdot) = 3$

$d(2, \cdot) = 4$

Algorithme

```
1:  procedure PARCOURS_LARGEUR(G = (S, A), s)
2:      dist[u] ← +∞,    ∀u ∈ S
3:      parent[u] ← ∅,    ∀u ∈ S
4:      dist[s] ← 0
5:      F ← File vide
6:      enfiler(F, s)
7:      tant que F n'est pas vide faire
8:          u ← defiler(F)
9:          pour tout v ∈ N(u) faire
10:             si dist[v] = +∞ alors
11:                 dist[v] ← dist[u]+1
12:                 parent[v] ← u
13:                 enfiler(F, v)
14:             fin si
15:         fin pour
16:     fin tant que
17: fin procedure
```

Complexité

Cet algorithme est linéaire en la taille du graphe d'entrée :

- Avec une matrice d'adjacence, la complexité est $\Theta(n^2)$.
- Avec des listes d'adjacence, la complexité est $\Theta(n + m)$.

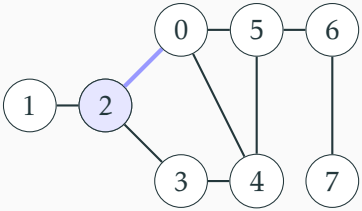
Parcours de graphe

Parcours en profondeur

Parcours en profondeur

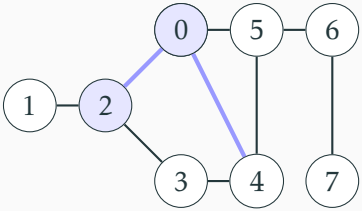
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

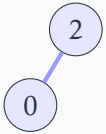
2



Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

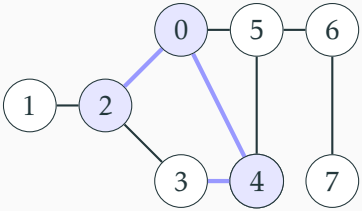


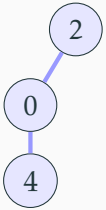


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

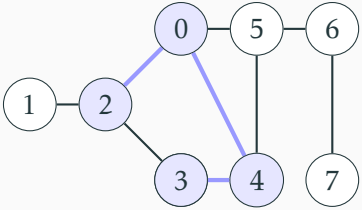
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

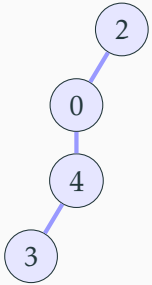




Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

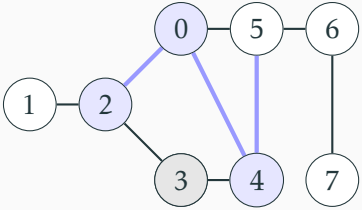


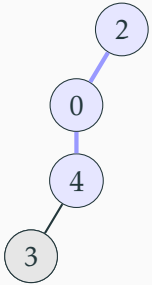


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

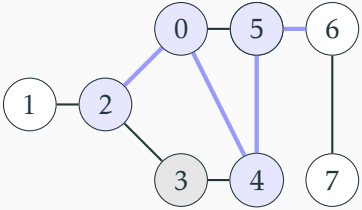
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

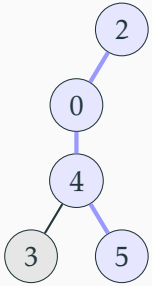




Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

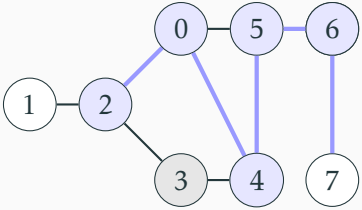


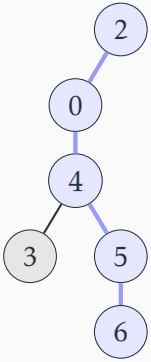


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

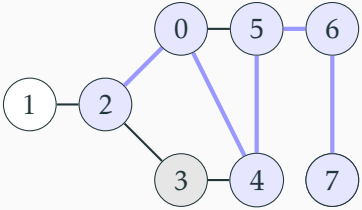
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

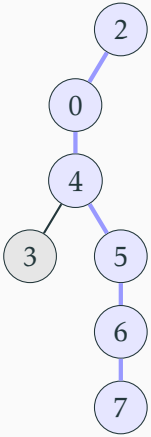




Parcours en profondeur

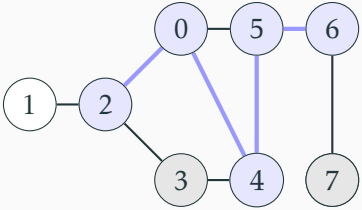
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

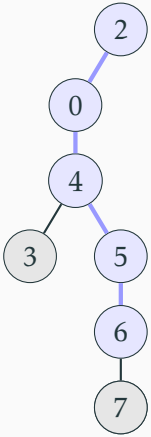




Parcours en profondeur

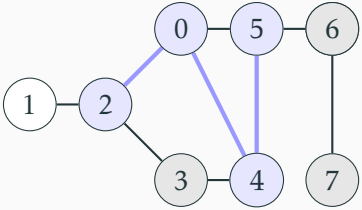
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

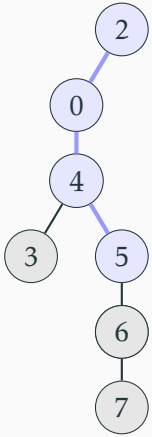




Parcours en profondeur

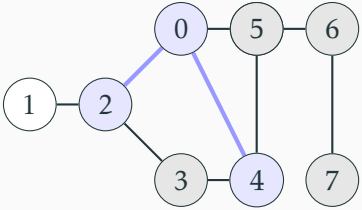
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

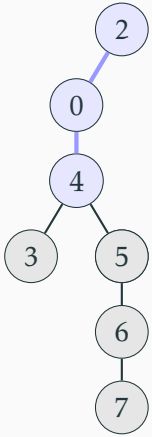




Parcours en profondeur

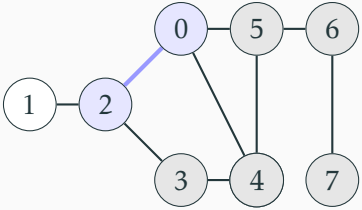
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

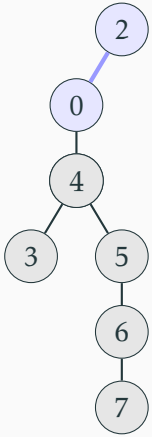




Parcours en profondeur

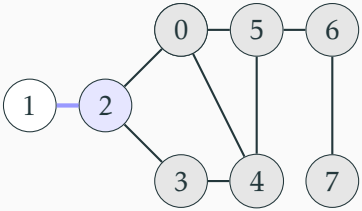
Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

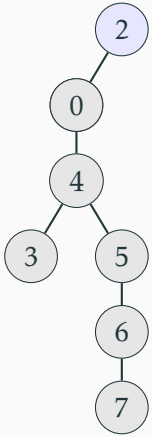




Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.
Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

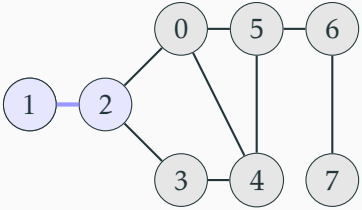


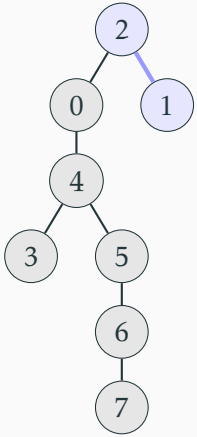


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

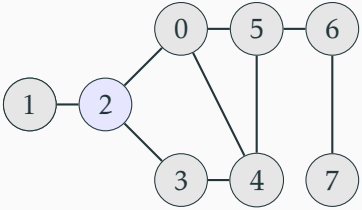


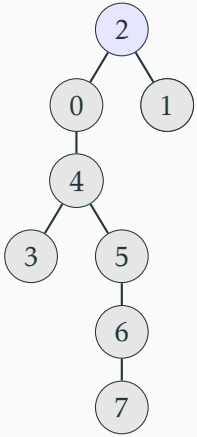


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.

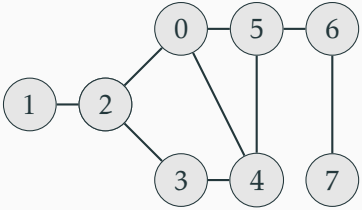


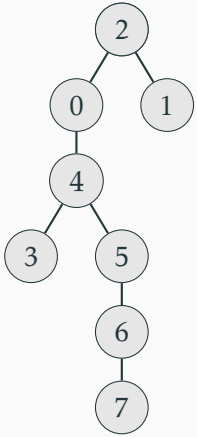


Parcours en profondeur

Le parcours en profondeur utilise une pile plutôt qu'une file.

Résultat : Au lieu de commencer par explorer les sommets les plus proches de la source s, le parcours va très vite aller visiter des sommets loin de la source s.





Algorithme

```
1:  procedure Parcours_profondeur(G = (S, A), s)
2:      etat[u] ← PAS_VU,    ∀u ∈ S
3:      parent[u] ← ∅,    ∀u ∈ S
4:      P ← Pile vide
5:      Visiter(P, ∅, s)
6:      tant que P n'est pas vide faire
7:          u ← depiler(P)
8:          si etat[u] = A_VOIR alors
9:              etat[u] ← EN_COURS
10:             empiler(P, u)
11:             pour tout v ∈ N(u) faire
12:                 si etat[v] = PAS_VU alors
13:                     Visiter(P, u, v)
14:                 fin si
15:             fin pour
16:         sinon
17:             etat[u] ← VU
18:         fin si
19:     fin tant que
20: fin procedure
```

```
1: procedure Visiter(P, u, v)
2:     etat[v] ← A_VOIR
3:     parent[v] ← u
4:     empiler(P, v)
5: fin procedure
```

Version récursive

```
1: etat[u] ← PAS_VU,   ∀u ∈ S
2: parent[u] ← ∅,   ∀u ∈ S
3:
4: procedure Parcours_profondeur(G = (S, A), u)
5:     etat[u] ← EN_COURS
6:     pour tout v ∈ N(u) faire
7:         si etat[v] = PAS_VU alors
8:             parent[v] ← u
9:             Parcours_profondeur(G, v)
10:        fin si
11:    fin pour
12:    etat[u] ← VU
13: fin procedure
14:
15: Parcours_profondeur(G, s)
```

Complexité

Cet algorithme est à nouveau linéaire en la taille du graphe d'entrée :

- Avec une matrice d'adjacence, la complexité est $\Theta(n^2)$.
- Avec des listes d'adjacence, la complexité est $\Theta(n + m)$.

Exemple d'application

On peut se servir d'un parcours en profondeur pour déterminer si un graphe orienté possède un cycle.
On peut montrer qu'un graphe possède un cycle si et seulement si on rencontre un sommet dans l'état `EN_COURS` durant le parcours.

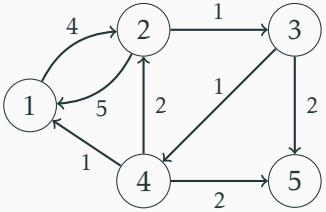
Détection de cycle

```
1:  etat[u] ← PAS_VU,    ∀u ∈ S
2:
3:  procedure Detecter_cycle(G = (S, A), u)
4:      etat[u] ← EN_COURS
5:      pour tout v ∈ N⁺(u) faire
6:          si etat[v] = PAS_VU alors
7:              si Detecter_cycle(G, v) = VRAI alors
8:                  retourner VRAI
9:              fin si
10:         sinon si etat[v] = EN_COURS alors
11:             retourner VRAI
12:         fin si
13:     fin pour
14:     etat[u] ← VU
15:     retourner FAUX
16: fin procedure
```

Algorithmes de plus court chemin

Graphe pondéré

Un graphe pondéré est un triplet $G = (S, A, w)$ où (S, A) est un graphe (orienté ou non) et $w : A \to \mathbb{R}$ est une application associant un poids $w((s,t))$ à chacun des arcs du graphes. Si G est non orienté, $w((t,s)) = w((s,t))$.

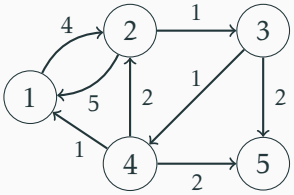


$w(1,2) = 4$

$w(2,3) = 1$

Représentation de graphe

Il existe deux façons classiques de représenter un graphe pondéré :



Graphe orienté

	1	2	3	4	5
1	0	4	∞	∞	∞
2	5	0	1	∞	∞
3	∞	∞	0	1	2
4	1	2	∞	0	2
5	∞	∞	∞	∞	0

Matrice d'adjacence

1	2, 4		
2	1, 4	3, 1	
3	4/1	5/2	
4	1/1	2/2	5/2
5	·		

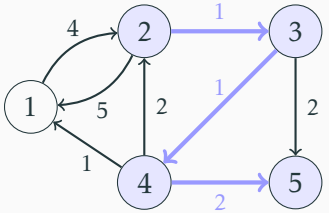
Listes d'adjacence

	Matrice d'adj.	Listes d'adj.
Mémoire	$\Theta(n^2)$	$\Theta(n+m)$
Test "$(x,y) \in A$?"	$\Theta(1)$	$\Theta(d^+(x))$

Poids d'un chemin

Dans un graphe pondéré $G = (S, A, w)$, on définit le poids $W(c)$ d'un chemin $c = \langle s_0, \ldots, s_k \rangle$ par :

$$W(c) = \sum_{i=1}^{k} w(s_{i-1}, i)$$



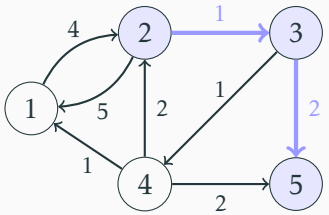
$W(\langle 2, 3, 4, 5 \rangle) = 1 + 1 + 2 = 4$

Distance pondérée entre sommets

Dans un graphe pondéré $G = (S, A, w)$, si t est un sommet accessible depuis s, alors l'ensemble $C(s,t)$ des chemins allant de s à t n'est pas vide. Dans ce cas, la distance pondérée $\delta(s,t)$ entre s et t est défini par :

$$\delta(s,t) = \min_{c \in C(s,t)} W(c)$$

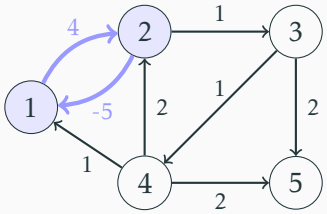
Lorsque t n'est pas accessible depuis s, on pose $\delta(s,t) = +\infty$.



$\delta(2,5) = 1 + 2 = 3$

Cycle de poids négatif ?

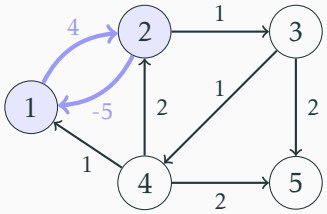
Que se passe-t-il lorsqu'un cycle de poids négatif existe ?



$W(\langle 1,2,1\rangle) = 4 - 5 = -1$

Cycle de poids négatif ?

Que se passe-t-il lorsqu'un cycle de poids négatif existe ?

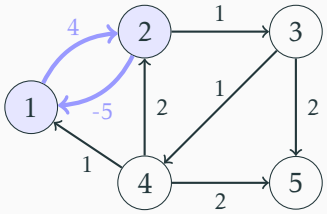


$W(\langle 1,2,1\rangle) = 4 - 5 = -1$

Quelles est la distance pondérée entre 4 et 3 ?

Cycle de poids négatif ?

Que se passe-t-il lorsqu'un cycle de poids négatif existe ?



$W(\langle 1,2,1\rangle) = 4 - 5 = -1$

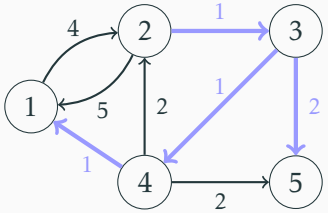
Quelles est la distance pondérée entre 4 et 3 ?

$$\forall k \geqslant 0, \quad W\left(\langle 4,1\rangle \cdot \langle 1,2,1\rangle^k + \langle 1,2,3\rangle\right) = 1 + k \cdot (-1) + (4+1) = 6 - k$$

$$\delta(4,3) = -\infty$$

Algorithme de plus court chemin

Un algorithme de plus court chemin est un algorithme qui étant donné un graphe pondéré $G = (S, A, w)$ et un sommet source $s \in S$, calcule les distances pondérées entre s et tous les autres sommets de G. De plus, un tel algorithme calcule un arbre enraciné en s de tel sorte que la distance pondérée du chemin entre s et t dans cet arbre soit égale $\delta(s, t)$.



$\delta(2,1) = 3$
$\delta(2,2) = 0$
$\delta(2,3) = 1$
$\delta(2,4) = 2$
$\delta(2,5) = 3$

Algorithmes de plus court chemin

Algorithme de Dijkstra

Rappel

On rappelle qu'un parcours en largeur permet de calculer la distance (non pondérée) entre un sommet source s et tous les autres sommets accessibles depuis s. L'algorithme de Dijkstra est une version modifiée du parcours en largeur.

Rappel

On rappelle qu'un parcours en largeur permet de calculer la distance (non pondérée) entre un sommet source s et tous les autres sommets accessibles depuis s. L'algorithme de Dijkstra est une version modifiée du parcours en largeur.

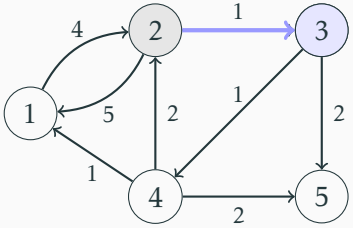
Le parcours en largeur maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.

Rappel

On rappelle qu'un parcours en largeur permet de calculer la distance (non pondérée) entre un sommet source s et tous les autres sommets accessibles depuis s. L'algorithme de Dijkstra est une version modifiée du parcours en largeur.

L'algorithme de Dijkstra maintient un ensemble de sommets visités V.
Initialement $V = \{s\}$.
~~À chaque étape, on ajoute tous les voisins de V qui ne sont pas encore dans V.~~
À chaque étape, on ajoute **le voisin** de V **le plus proche de** s qui n'est pas encore dans V.

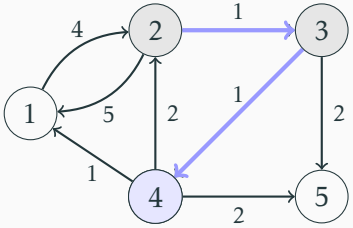
Déroulement de l'algorithme de Dijkstra



sommet	1	2	3	4	5
dist	5	0	1	$+\infty$	$+\infty$

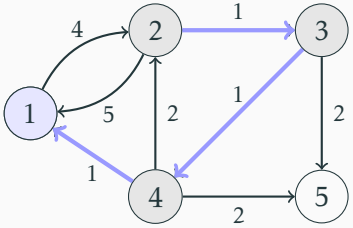
2

Déroulement de l'algorithme de Dijkstra

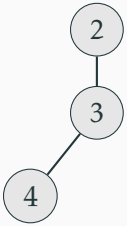


sommet	1	2	3	4	5
dist	5	0	1	2	3

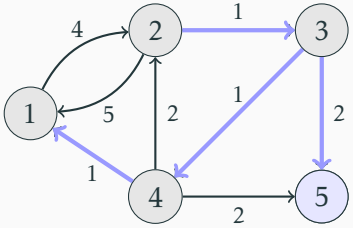
Déroulement de l'algorithme de Dijkstra



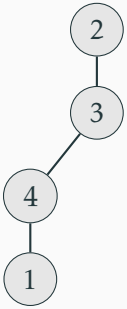
sommet	1	2	3	4	5
dist	3	0	1	2	3



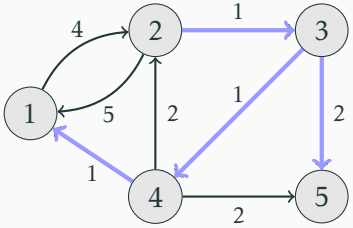
Déroulement de l'algorithme de Dijkstra



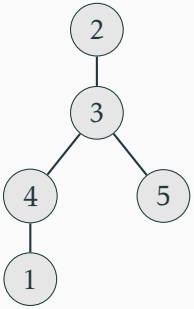
sommet	1	2	3	4	5
dist	3	0	1	2	3



Déroulement de l'algorithme de Dijkstra



sommet	1	2	3	4	5
dist	3	0	1	2	3



Algorithme de Dijkstra

```
1:  procedure DIJKSTRA(G = (S, A, w), s)
2:      dist[u] ← +∞,    ∀u ∈ S
3:      parent[u] ← ∅,    ∀u ∈ S
4:      F ← File de priorité vide
5:      Visiter(F, ∅, s, 0)
6:      tant que F n'est pas vide faire
7:          (d, u) ← defiler_min(F)
8:          si d ≠ dist[u] alors
9:              continuer
10:         fin si
11:         pour tout v ∈ N⁺(u) faire
12:             d' ← d + w(u, v)
13:             si d' < dist[v] alors
14:                 Visiter(F, u, v, d')
15:             fin si
16:         fin pour
17:     fin tant que
18: fin procedure
```

```
1: procedure VISITER(F, u, v, d)
2:     dist[v] = d
3:     parent[v] = u
4:     enfiler(F, (d, v))
5: fin procedure
```

Complexité

L'algorithme fait au plus $m = |A|$ appels à `enfiler`(F, (d, v)) et donc au plus m appels à `defiler`(F).

Complexité

L'algorithme fait au plus $m = |A|$ appels à `enfiler`(F, (d, v)) et donc au plus m appels à `defiler`(F).

Implémentation	Enfiler	Défiler
Liste	$\Theta(1)$	$\Theta(\|F\|)$
Liste triée	$\Theta(\|F\|)$	$\Theta(1)$
Tas binaire	$\Theta(\log \|F\|)$	$\Theta(\log \|F\|)$

Complexité

L'algorithme fait au plus $m = |A|$ appels à `enfiler`(F, (d, v)) et donc au plus m appels à `defiler`(F).

Implémentation	Enfiler	Défiler
Liste	$\Theta(1)$	$\Theta(\|F\|)$
Liste triée	$\Theta(\|F\|)$	$\Theta(1)$
Tas binaire	$\Theta(\log\|F\|)$	$\Theta(\log\|F\|)$

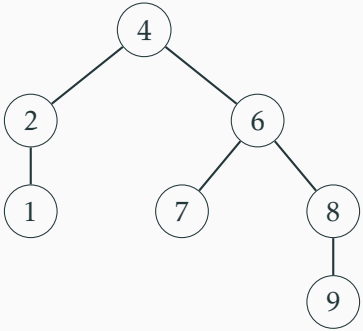
Durant l'exécution de Dijkstra la taille de F est un $\mathcal{O}(m)$. Donc :

$$C_{\text{Dijkstra}}(m) = \mathcal{O}\left(m \cdot \left(C_{\text{Enfiler}}(m) + C_{\text{Défiler}}(m)\right)\right)$$

En utilisant une liste triée ou non, $C_{\text{Dijkstra}}(m) = \mathcal{O}(m^2)$.
En revanche, avec un **tas binaire**, on obtient $\boldsymbol{C_{\text{Dijkstra}}(m) = \mathcal{O}(m \cdot \log(m))}$

Arbre binaire

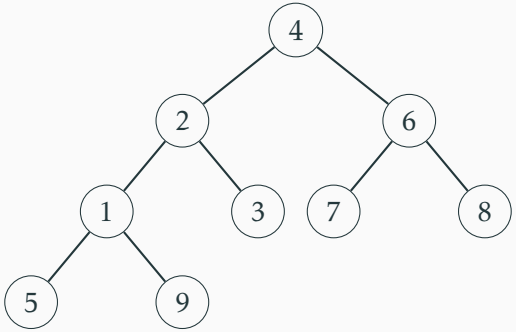
Un arbre binaire est un arbre enraciné dont chaque nœud possède au plus deux fils.



Arbre binaire complet

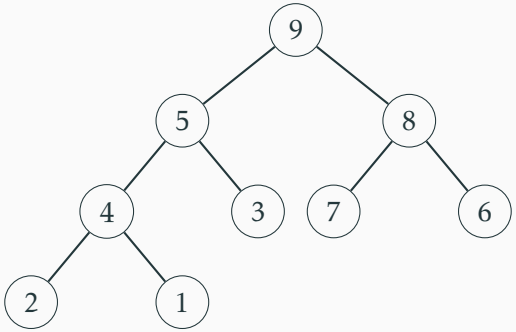
Arbre binaire complet

Un arbre binaire complet est un arbre binaire dont tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier niveau ne l'est pas totalement alors, il doit être rempli de gauche à droite.



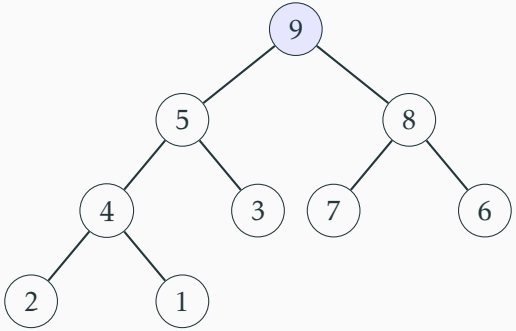
Tas binaire

Un tas binaire est un arbre binaire complet dans lequel chaque nœud est plus grand que c'est fils



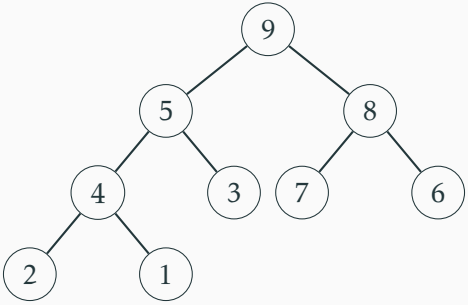
Valeur max d'un tas binaire

La plus grande valeur d'un tas binaire se trouve toujours à la racine de l'arbre.
Accéder à la plus grande valeur d'un tas binaire se fait donc en $\Theta(1)$.



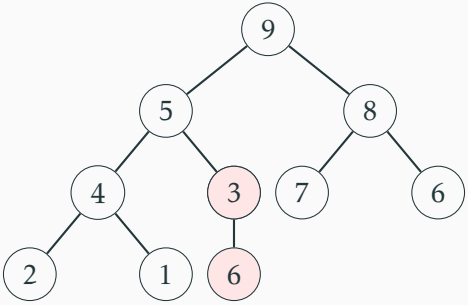
Ajout d'un nouveau nœud

On souhaite ajouter un nouveau nœud de valeur 6 à notre tas binaire :



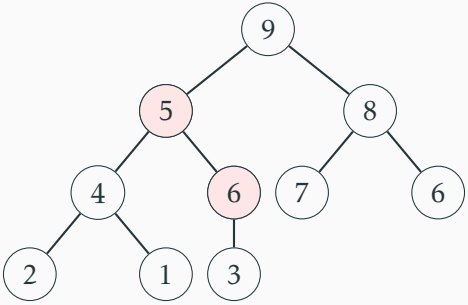
Ajout d'un nouveau nœud

On souhaite ajouter un nouveau nœud de valeur 6 à notre tas binaire :



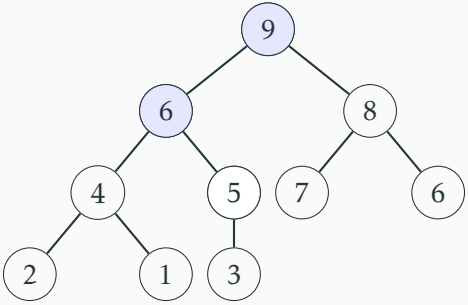
Ajout d'un nouveau nœud

On souhaite ajouter un nouveau nœud de valeur 6 à notre tas binaire :



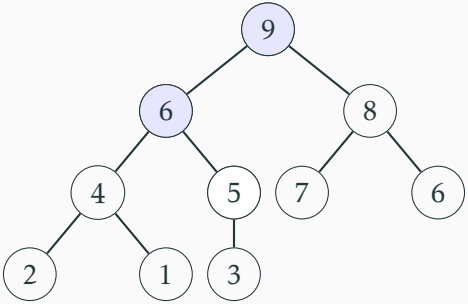
Ajout d'un nouveau nœud

On souhaite ajouter un nouveau nœud de valeur 6 à notre tas binaire :



Ajout d'un nouveau nœud

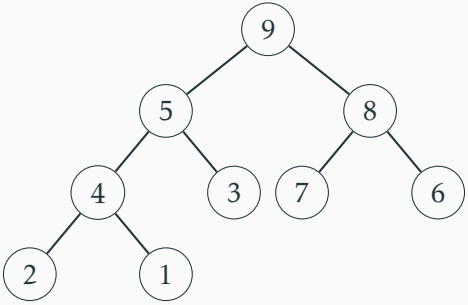
On souhaite ajouter un nouveau nœud de valeur 6 à notre tas binaire :



Le nombre d'étape nécessaire pour insérer un nouveau nœud est au plus la hauteur H de l'arbre.
Or dans un arbre binaire complet : $H = \lfloor \log_2(n) \rfloor$. L'**ajout d'un nouveau nœud** a donc une complexité en $\boldsymbol{\Theta(\log_2(n))}$.

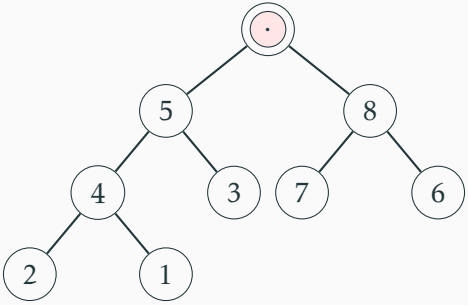
Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



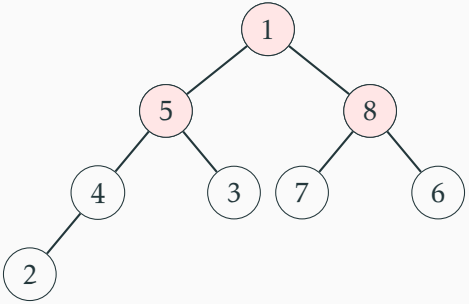
Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



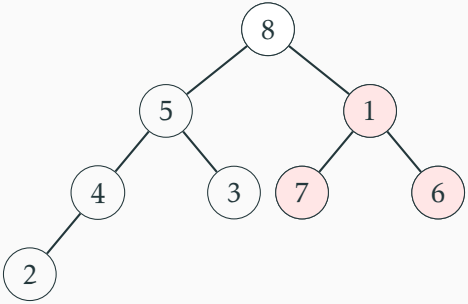
Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



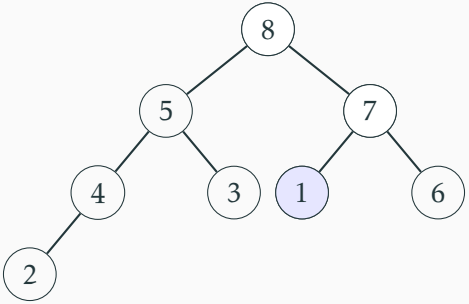
Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



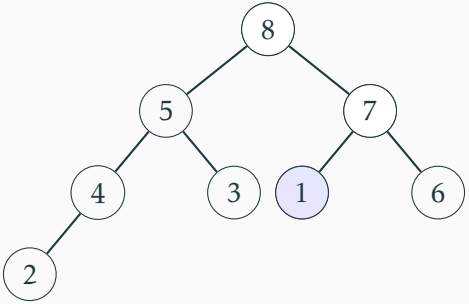
Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



Suppression de la racine

On souhaite supprimer la racine de notre tas binaire.



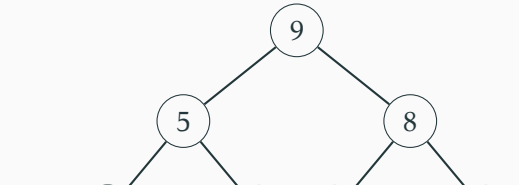
Le nombre d'étape nécessaire pour supprimer la racine est au plus la hauteur H de l'arbre.
Or dans un arbre binaire complet : $H = \lfloor \log_2(n) \rfloor$. **La suppression de la racine** a donc une complexité en $\boldsymbol{\Theta(\log_2(n))}$.

Conclusion

Un tas binaire permet bien d'implémenter une file de priorité avec ajout d'un nouvel élément et suppression du maximum en $\Theta(\log(n))$.

Représentation d'un tas binaire

En pratique on utilise un tableau pour représenter un tas binaire :

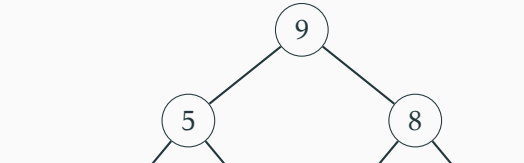


Indice	0	1	2	3	4	5	6	7	8	9
Tas T	·	9	5	8	4	3	7	6	2	1

La racine se trouve à l'indice 1 du tableau, puis on ajoute les niveaux les uns à la suite des autres.

Représentation d'un tas binaire

En pratique on utilise un tableau pour représenter un tas binaire :



Indice	0	1	2	3	4	5	6	7	8	9
Tas T	·	9	5	8	4	3	7	6	2	1

Les enfants de `T[i]` sont `T[2*i]` et `T[2*i+1]`. Le père de `T[i]` est `T[i/2]`.

Implémentation de l'insertion

Une implémentation possible de l'insertion d'une valeur x dans un tas T de taille n en C++ est la suivante :

```
void ajouter(int* T, int n, int x) {
    T[n] = x;
    while(n > 1 && T[n] > T[n/2]) {
        std::swap(T[n], T[n/2]);
        n /= 2;
    }
}
```

Algorithmes de plus court chemin

Algorithme de Bellman-Ford

Nécessité d'un autre algorithme

L'algorithme de Dijkstra ne fonctionne que lorsque le poids des arcs est positif.
En effet, l'algorithme de Dijkstra suppose que l'on visite les sommets par ordre croissant de distance à la source. Or, s'il existe des arcs de poids négatifs, on peut découvrir plus tard un sommet qui est en réalité plus proche de la source (exemple au tableau).
Lorsqu'un graphe pondéré possède des arcs de poids négatif, il faut trouver un autre algorithme.

Idées de l'algorithme de Bellman-Ford

S'il n'existe pas de cycle de poids négatif, alors un plus court chemin ne passe jamais deux fois par un même sommet.

$\Longrightarrow$ Un plus court chemin est de taille au plus $n-1$.

Idées de l'algorithme de Bellman-Ford

S'il n'existe pas de cycle de poids négatif, alors un plus court chemin ne passe jamais deux fois par un même sommet.
$\Longrightarrow$ Un plus court chemin est de taille au plus $n-1$.

On peut s'appuyer sur de la programmation dynamique en notant $D(t,k)$, le plus petit poids d'un chemin de taille au plus k entre la source s et t. On a :

$$\begin{cases} D(s,0) &= 0 \\ D(t,0) &= +\infty, \quad \forall t \neq s \end{cases}$$

Puis :

$$D(t,k+1) = \min\left\{D(t,k), \min_{u \in N^-(t)} [D(u,k) + w(u,t)]\right\}$$

Enfin la distance pondérée entre s et t, est :

$$d(s,t) = D(t,n-1)$$

Algorithme de Bellman-Ford

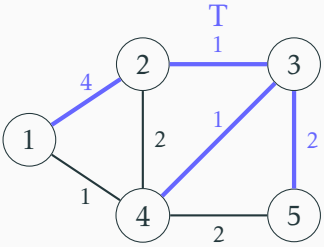
1: **procedure** BELLMAN_FORD($G = (S, A, w)$, s)
2: $\quad$ dist$[u] \leftarrow +\infty, \quad \forall u \in S$
3: $\quad$ parent$[u] \leftarrow \emptyset, \quad \forall u \in S$
4: $\quad$ dist$[s] \leftarrow 0$
5: $\quad$ **pour** $k \leftarrow 1$ à $|S| - 1$ **faire**
6: $\quad\quad$ **pour tout** $(u, v) \in A$ **faire**
7: $\quad\quad\quad$ $d \leftarrow$ dist$[u] + w(u, v)$
8: $\quad\quad\quad$ **si** $d <$ dist$[v]$ **alors**
9: $\quad\quad\quad\quad$ dist$[v] \leftarrow d$
10: $\quad\quad\quad\quad$ parent$[v] \leftarrow u$
11: $\quad\quad\quad$ **fin si**
12: $\quad\quad$ **fin pour**
13: $\quad$ **fin pour**
14: **fin procedure**

Arbre couvrant de poids minimal

Poids d'un arbre couvrant

Soit $G = (S, A, w)$ un graphe non orienté pondéré et $T = (S, A_T)$ un arbre couvrant de G. Alors, on définit le poids de T comme :

$$W(T) = \sum_{(u,v) \in A_T} w(u,v)$$



$W(T) = 4 + 1 + 1 + 2 = 8$

Problème

Le problème de l'arbre couvrant de poids minimal consiste à trouver un arbre couvrant ayant le plus petit poids possible.

Problème

Le problème de l'arbre couvrant de poids minimal consiste à trouver un arbre couvrant ayant le plus petit poids possible.

Exemple de cas concret

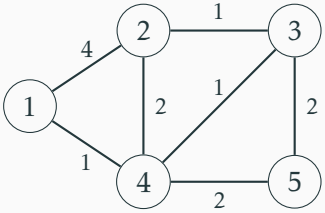
On souhaite relier des villes par des routes. On assigne un coût à la construction d'une route entre chaque paire de ville. L'arbre couvrant de poids minimal permet de relier toutes les villes avec le plus petit coût total possible.

Arbre couvrant de poids minimal

Algorithme de Kruskal

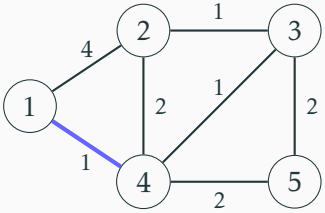
Idées de l'algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton qui trie les arêtes par ordre croissant de poids puis tente d'ajouter les arêtes une à une à notre arbre couvrant.



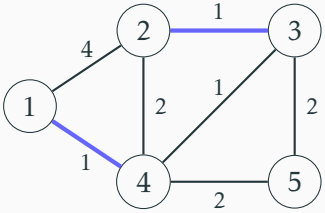
Idées de l'algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton qui trie les arêtes par ordre croissant de poids puis tente d'ajouter les arêtes une à une à notre arbre couvrant.



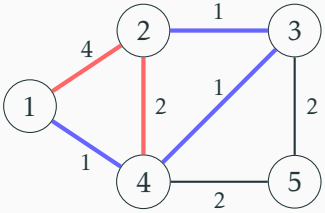
Idées de l'algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton qui trie les arêtes par ordre croissant de poids puis tente d'ajouter les arêtes une à une à notre arbre couvrant.



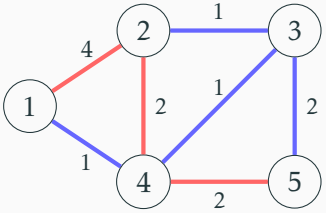
Idées de l'algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton qui trie les arêtes par ordre croissant de poids puis tente d'ajouter les arêtes une à une à notre arbre couvrant.



Idées de l'algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton qui trie les arêtes par ordre croissant de poids puis tente d'ajouter les arêtes une à une à notre arbre couvrant.



$W(T) = 5$

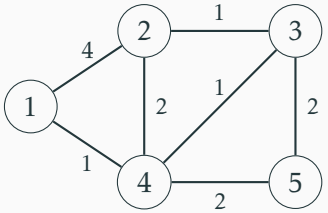
Algorithme de Kruskal

1: **procedure** KRUSKAL($G = (S, A, w)$)
2: $A_T \leftarrow \emptyset$
3: U $\leftarrow$ *Union_Find* de taille $|S|$
4: trier les arêtes de A par poids croissant
5: **pour tout** $(u, v) \in A$ par ordre croissant de poids **faire**
6: **si** trouver(U, u) $\neq$ trouver(U, v) **alors**
7: unir(U, u, v)
8: $A_T \leftarrow A_T \cup \{(u, v)\}$
9: **fin si**
10: **fin pour**
11: **retourner** A_T
12: **fin procedure**

Union-Find

Un Union-Find est une structure de données représentant une partition d'un ensemble en plusieurs classe d'équivalence. La structure supporte deux opérations :

- **Trouver** : retourne le représentant de la classe d'équivalence d'un élément. Cette opération est principalement utiliser pour savoir si deux élément u et v appartiennent à la même classe d'équivalence via le test `Trouver(u) = Trouver(v)`.
- **Unir** : réunit deux classes d'équivalence en une seule.

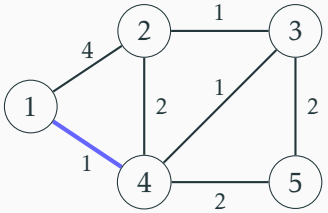


$U = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$

Union-Find

Un Union-Find est une structure de données représentant une partition d'un ensemble en plusieurs classe d'équivalence. La structure supporte deux opérations :

- **Trouver** : retourne le représentant de la classe d'équivalence d'un élément. Cette opération est principalement utiliser pour savoir si deux élément u et v appartiennent à la même classe d'équivalence via le test `Trouver(u) = Trouver(v)`.
- **Unir** : réunit deux classes d'équivalence en une seule.

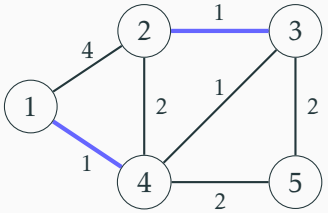


$U = \{\{1,4\},\{2\},\{3\},\{5\}\}$

Union-Find

Un Union-Find est une structure de données représentant une partition d'un ensemble en plusieurs classe d'équivalence. La structure supporte deux opérations :

- **Trouver** : retourne le représentant de la classe d'équivalence d'un élément. Cette opération est principalement utiliser pour savoir si deux élément u et v appartiennent à la même classe d'équivalence via le test `Trouver(u) = Trouver(v)`.
- **Unir** : réunit deux classes d'équivalence en une seule.

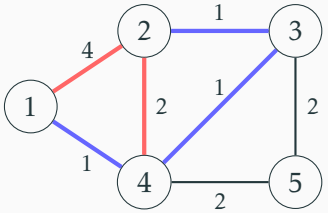


$U = \{\{1,4\},\{2,3\},\{5\}\}$

Union-Find

Un Union-Find est une structure de données représentant une partition d'un ensemble en plusieurs classe d'équivalence. La structure supporte deux opérations :

- **Trouver** : retourne le représentant de la classe d'équivalence d'un élément. Cette opération est principalement utiliser pour savoir si deux élément u et v appartiennent à la même classe d'équivalence via le test `Trouver(u) = Trouver(v)`.
- **Unir** : réunit deux classes d'équivalence en une seule.

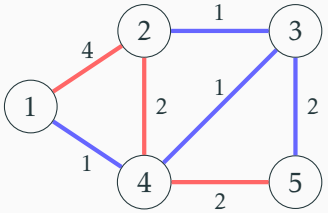


$U = \{\{1, 2, 3, 4\}, \{5\}\}$

Union-Find

Un Union-Find est une structure de données représentant une partition d'un ensemble en plusieurs classe d'équivalence. La structure supporte deux opérations :

- **Trouver** : retourne le représentant de la classe d'équivalence d'un élément. Cette opération est principalement utiliser pour savoir si deux élément u et v appartiennent à la même classe d'équivalence via le test `Trouver(u) = Trouver(v)`.
- **Unir** : réunit deux classes d'équivalence en une seule.



Complexité

La complexité des opérations sur un Union-Find sont en $\Theta(\alpha(n))$ (voir `https://fr.wikipedia.org/wiki/Union-find`). En pratique $\forall n, \alpha(n) \leqslant 5$.
On peut considérer que les opérations sur un Union-Find sont en $\Theta(1)$.

Complexité

La complexité des opérations sur un Union-Find sont en $\Theta(\alpha(n))$ (voir `https://fr.wikipedia.org/wiki/Union-find`). En pratique $\forall n, \alpha(n) \leqslant 5$.
On peut considérer que les opérations sur un Union-Find sont en $\Theta(1)$.

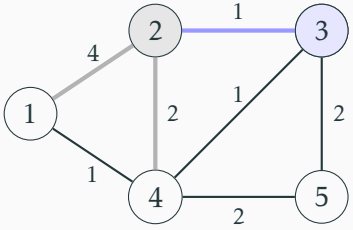
Toute la complexité de l'algorithme de Kruskal réside dans le tri des arêtes.
On obtient une complexité en $\boldsymbol{\Theta(m \cdot \log(m))}$.

Arbre couvrant de poids minimal

Algorithme de Prim

Idées de l'algorithme de Prim

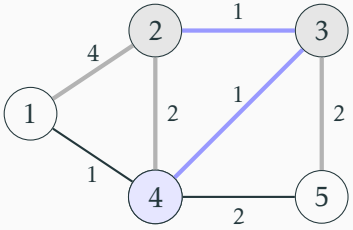
L'algorithme de Prim consiste à faire croître une composante connexe en partant d'une source $\{s\}$ et en ajoutant étape par étape le sommet le plus proche de la composante connexe que l'on construit.



sommet	1	2	3	4	5
dist	4	0	1	2	$+\infty$

Idées de l'algorithme de Prim

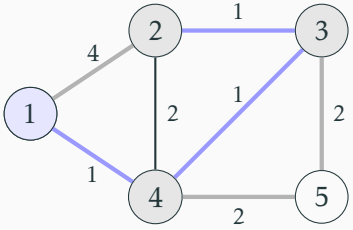
L'algorithme de Prim consiste à faire croître une composante connexe en partant d'une source $\{s\}$ et en ajoutant étape par étape le sommet le plus proche de la composante connexe que l'on construit.



sommet	1	2	3	4	5
dist	4	0	0	1	2

Idées de l'algorithme de Prim

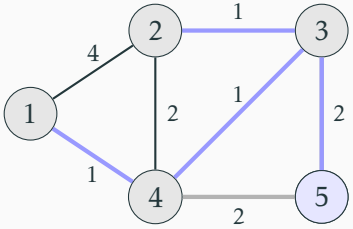
L'algorithme de Prim consiste à faire croître une composante connexe en partant d'une source $\{s\}$ et en ajoutant étape par étape le sommet le plus proche de la composante connexe que l'on construit.



sommet	1	2	3	4	5
dist	1	0	0	0	2

Idées de l'algorithme de Prim

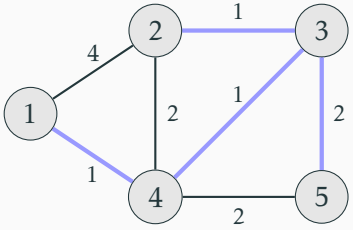
L'algorithme de Prim consiste à faire croître une composante connexe en partant d'une source $\{s\}$ et en ajoutant étape par étape le sommet le plus proche de la composante connexe que l'on construit.



sommet	1	2	3	4	5
dist	0	0	0	0	2

Idées de l'algorithme de Prim

L'algorithme de Prim consiste à faire croître une composante connexe en partant d'une source $\{s\}$ et en ajoutant étape par étape le sommet le plus proche de la composante connexe que l'on construit.



sommet	1	2	3	4	5
dist	0	0	0	0	0

Algorithme de Prim

```
1:  procedure Prim(G = (S, A, w), s)
2:      dist[u] ← +∞,    ∀u ∈ S
3:      parent[u] ← ∅,    ∀u ∈ S
4:      F ← File de priorité vide
5:      Visiter(F, ∅, s, 0)
6:      tant que F n'est pas vide faire
7:          (d, u) ← defiler_min(F)
8:          si dist[u] = 0 alors
9:              continuer
10:         fin si
11:         dist[u] ← 0
12:         pour tout v ∈ N(u) faire
13:             si w(u, v) < dist[v] alors
14:                 Visiter(F, u, v, w(u, v))
15:             fin si
16:         fin pour
17:     fin tant que
18: fin procedure
```

```
1: procedure Visiter(F, u, v, d)
2:     dist[v] = d
3:     parent[v] = u
4:     enfiler(F, (d, v))
5: fin procedure
```

Complexité

L'algorithme est très similaire à l'algorithme de Dijkstra et effectue un $\Theta(m)$ opérations sur une file de priorité. Il en résulte une complexité de l'algorithme de Prim en $\boldsymbol{\Theta(m \cdot \log(m))}$ lorsqu'on utilise un tas binaire.