Amphi de révision : Algorithmique et complexité

Mattéo Rizza Murgier

19 janvier 2026

Infos sur l'examen

- L'examen dure 3h ;
- Vous avez le droit à des documents **manuscrits** ;
- Les chapitres 1 à 6, ainsi que les chapitres de pré-requis sont au programme de l'examen.

Attention

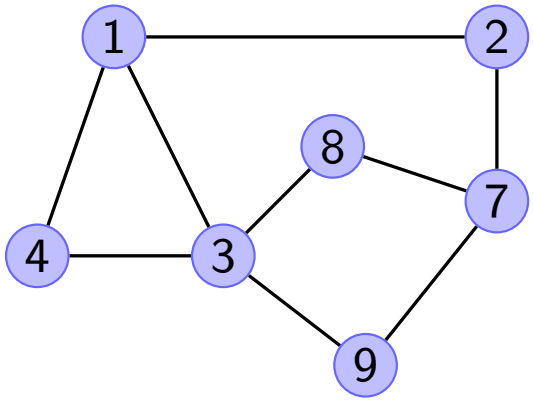
La structure et le contenu du cours ont changé par rapport à l'année dernière. Certains concepts ont été ajoutés au cours et seront mis en évidence dans ce qui suit.

1 Algorithmes de graphes

Définition - Graphe non-orienté

On note $G = (V, E)$ où :

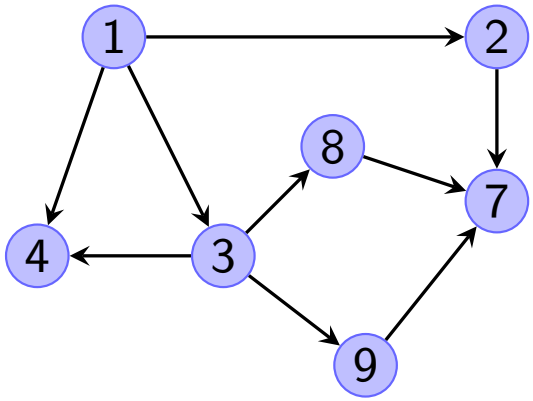
- V est un ensemble de nœuds ;
- E est un ensemble d'arêtes ;
- Une arête $e = (u, v)$ est une paire d'éléments de V.



Définition - Graphe orienté

On note $G = (V, E)$ où :

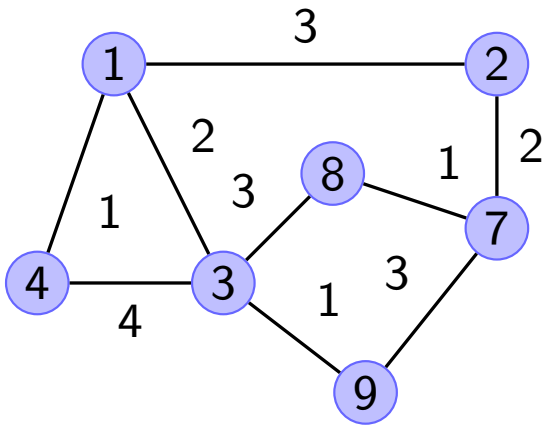
- V est un ensemble de nœuds ;
- E est un ensemble d'arcs ;
- Une arête $e = (u, v)$ est un couple d'éléments de V.



Définition - Graphe pondéré

On note $G = (V, E, w)$ où :

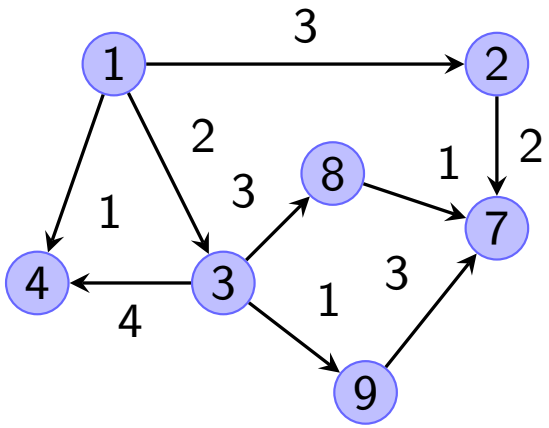
- V est un ensemble de nœuds ;
- E est un ensemble d'arêtes ;
- Une arête $e = (u, v)$ est une paire d'éléments de V.
- $w : E \to \mathbb{R}$ est une fonction de pondération



Définition - Graphe orienté pondéré

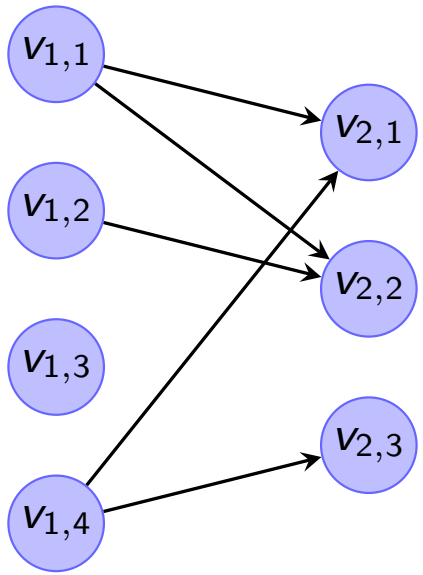
On note $G = (V, E, w)$ où :

- V est un ensemble de nœuds ;
- E est un ensemble d'arêtes ;
- Une arête $e = (u, v)$ est une paire d'éléments de V.
- $w : E \to \mathbb{R}$ est une fonction de pondération



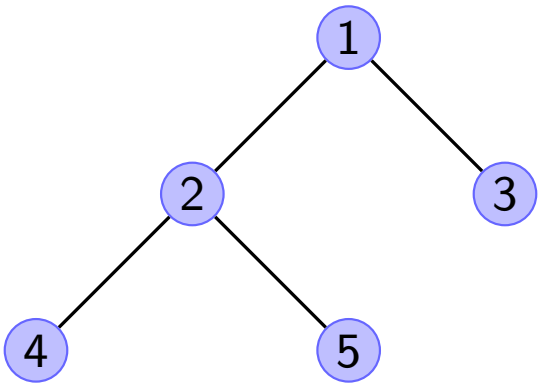
Graphe biparti

Graphe dont l'ensemble des sommets V peut être partitionné en deux ensembles V_1 et V_2 tels que chaque arête a une extrémité dans V_1 et l'autre dans V_2.



Arbre

Graphe acyclique et connexe (il existe un chemin entre toute paire de sommets).



Idée de l'algorithme

Lorsqu'on visite un nœud, on le marque comme visité, puis on visite immédiatement le premier de ses voisins qui n'a pas encore été vu, et ainsi de suite.

Complexité : $O(|V| + |E|)$ avec une liste d'adjacence

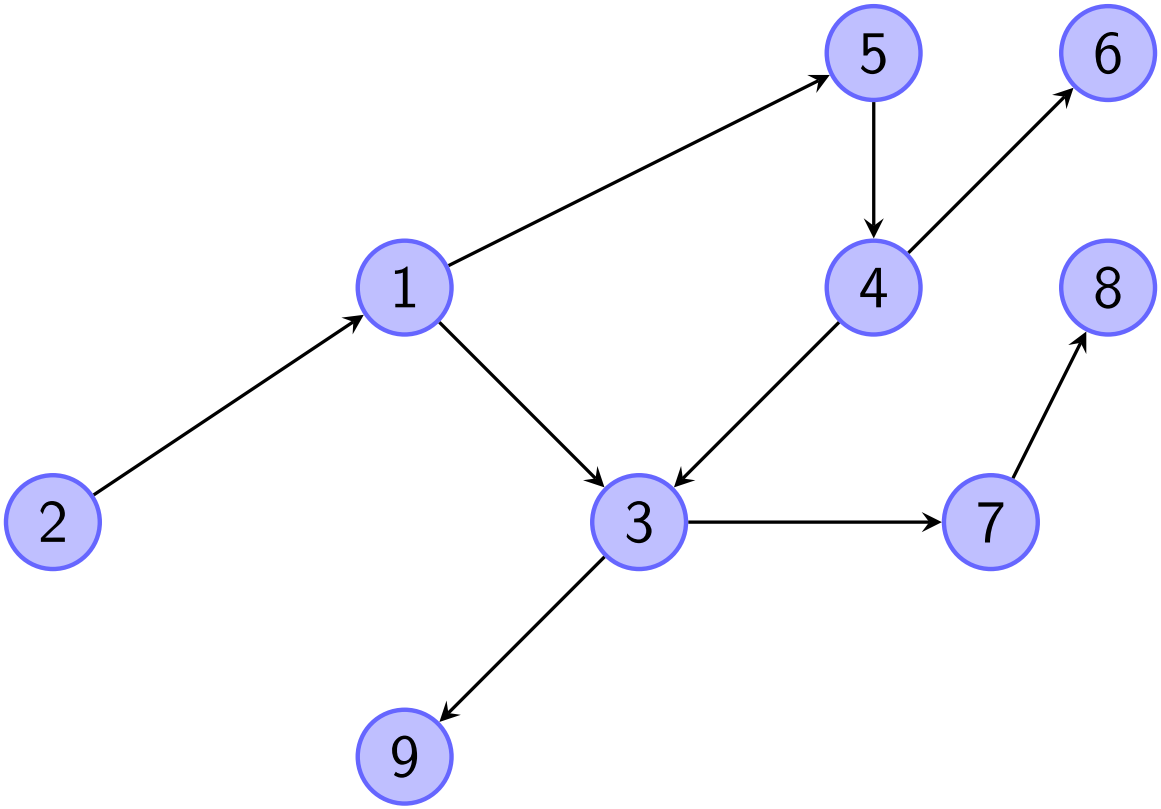
On peut soit utiliser une structure de pile (dernier arrivé, premier sorti), soit écrire une fonction récursive.

Parcours en profondeur

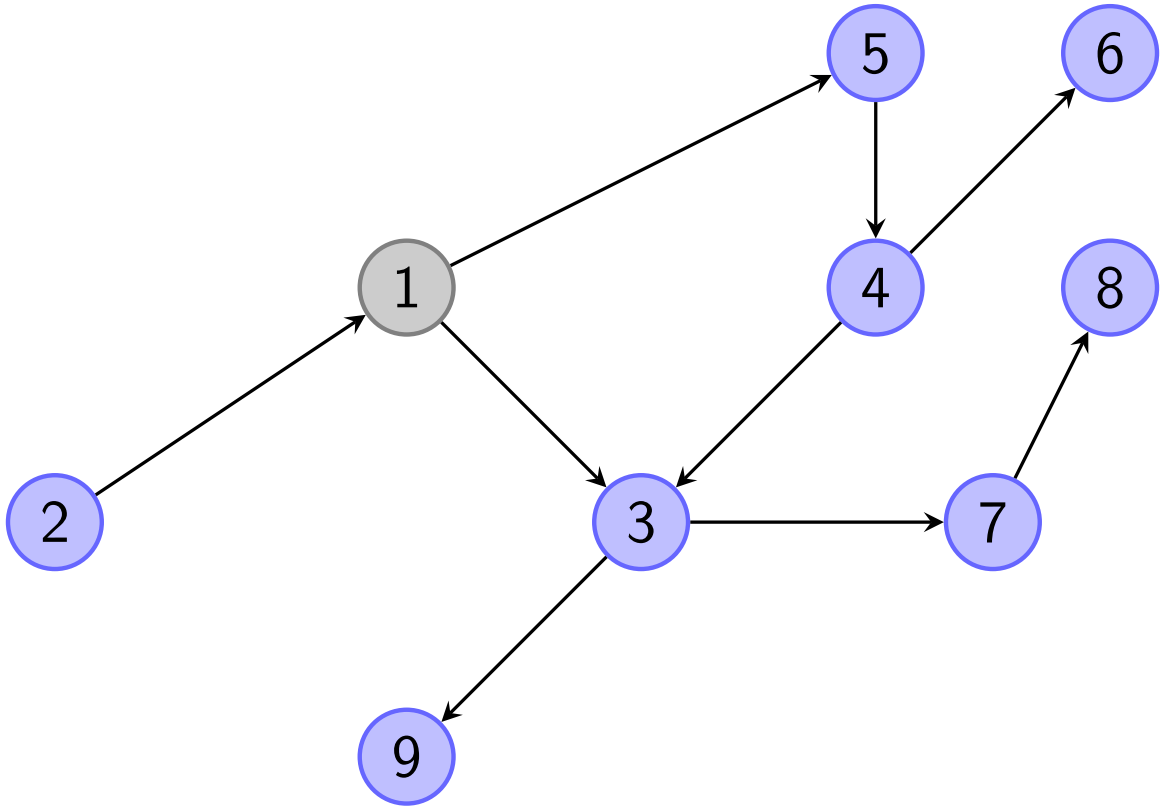
Pour un graphe sous forme de listes d'adjacence :

```
def dfs(start_node, graph):

    visited = {}
    def visit(node):
        visited[node] = True

        for v in graph[node]:
            if v not in visited:
                visit(v)

    visit(start_node)
```

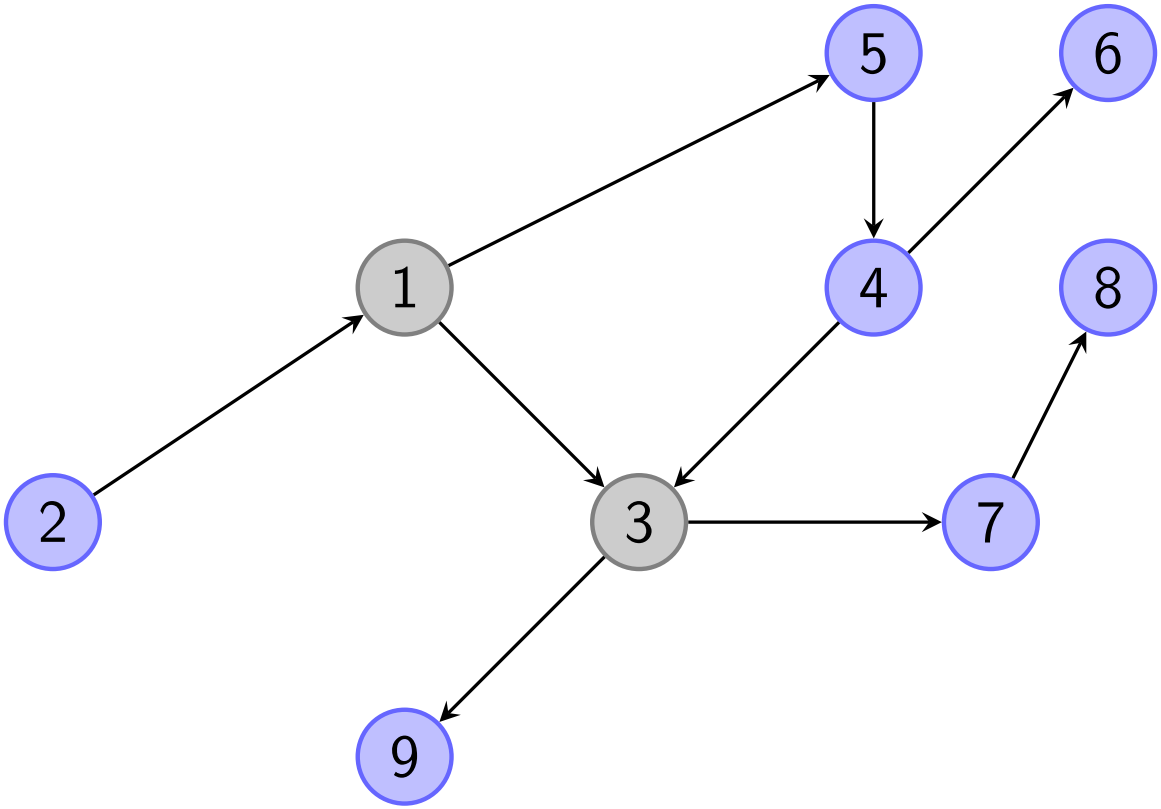
Parcours en profondeur



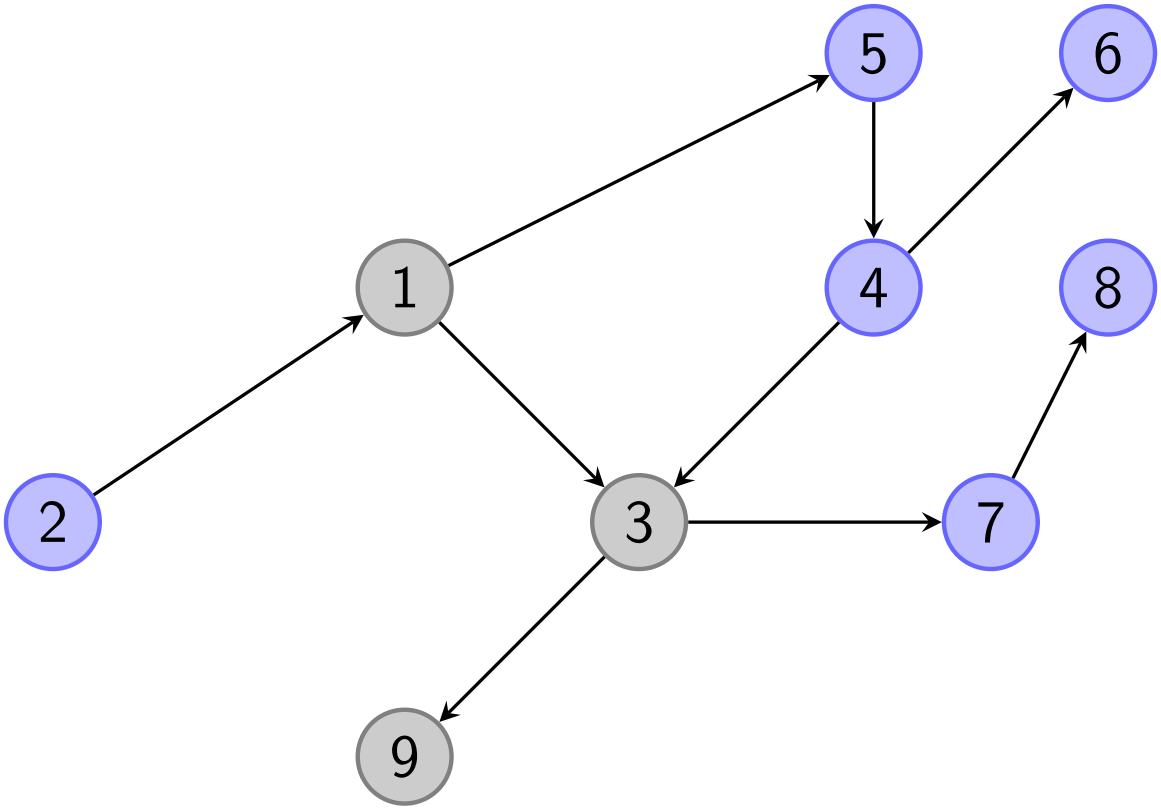
Parcours en profondeur



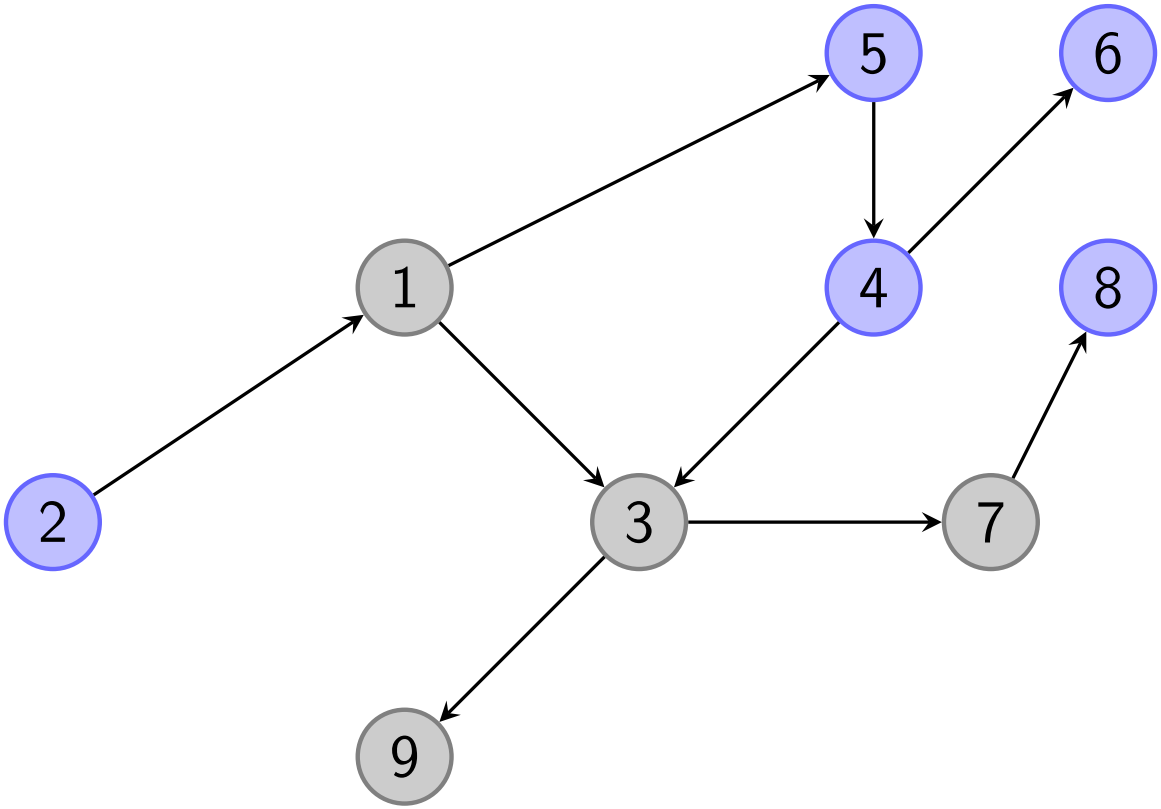
Parcours en profondeur



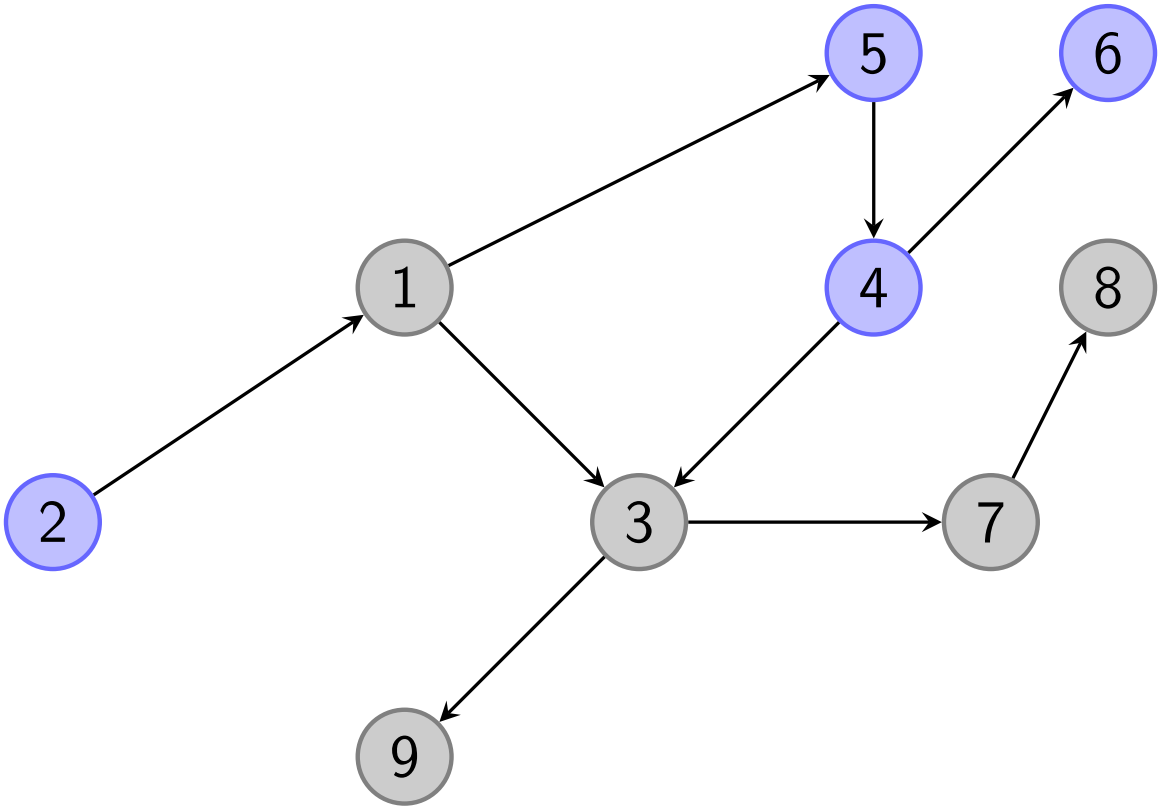
Parcours en profondeur



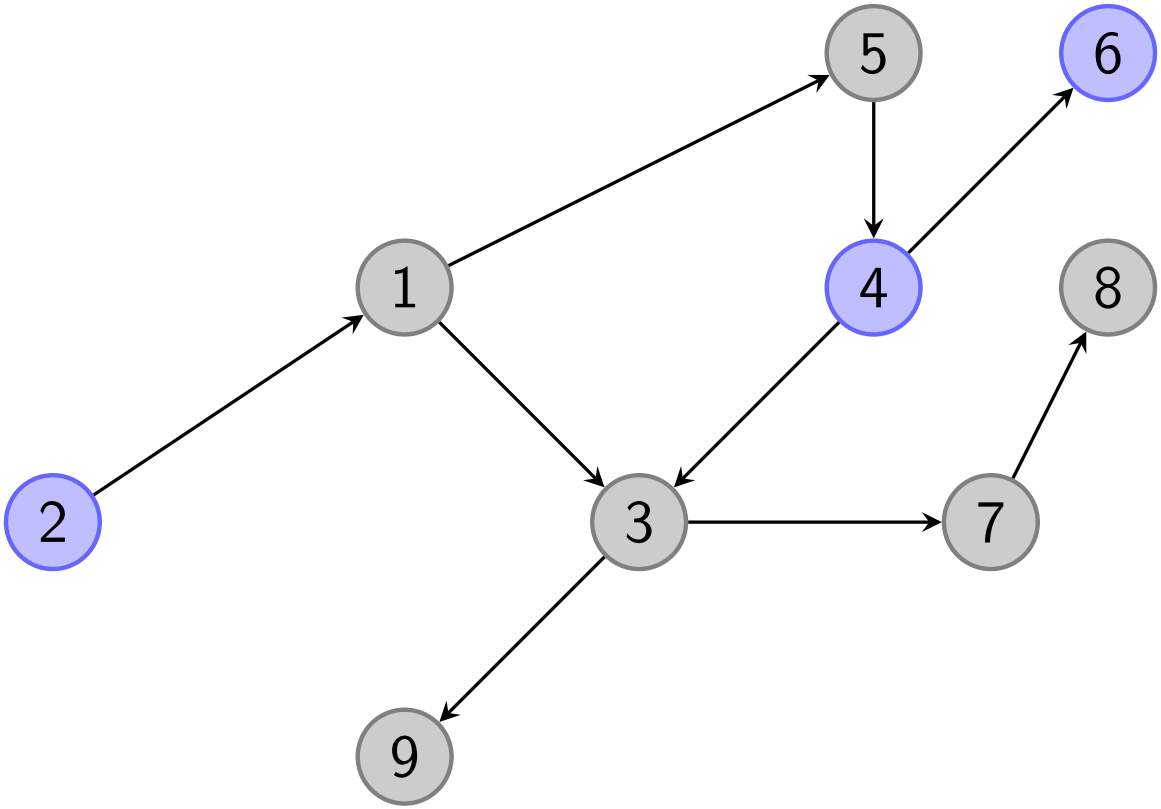
Parcours en profondeur



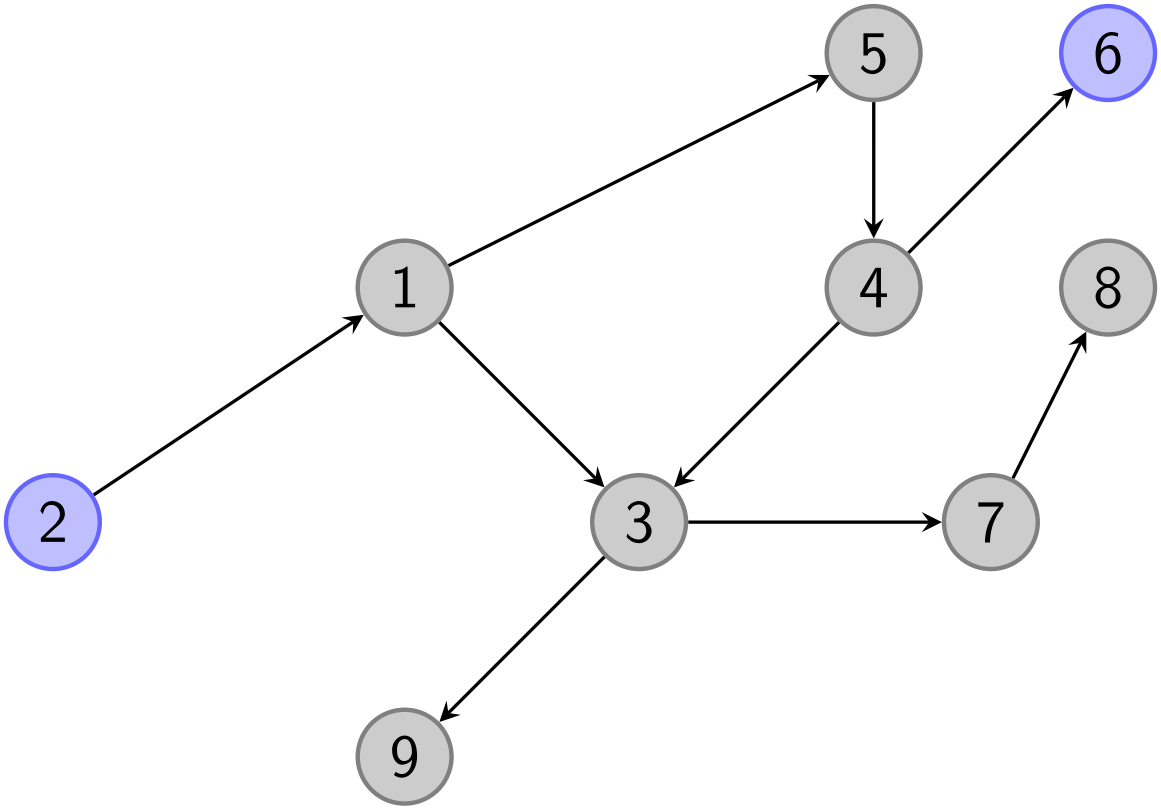
Parcours en profondeur



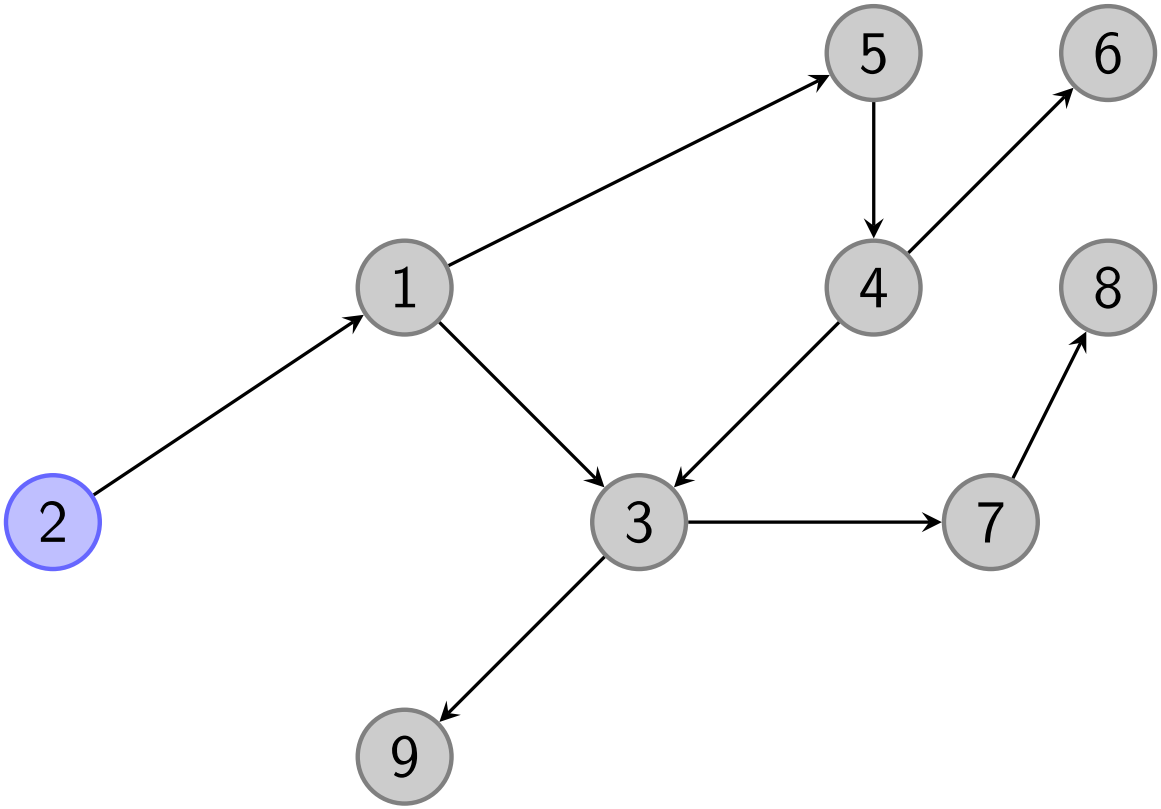
Parcours en profondeur



Parcours en profondeur



Parcours en profondeur



Principe de l'algorithme

Lorsqu'on visite un nœud, on le marque comme visité, puis on visite successivement l'ensemble de ses voisins avant de passer à la suite.

Complexité : $O(|V| + |E|)$ avec une liste d'adjacence

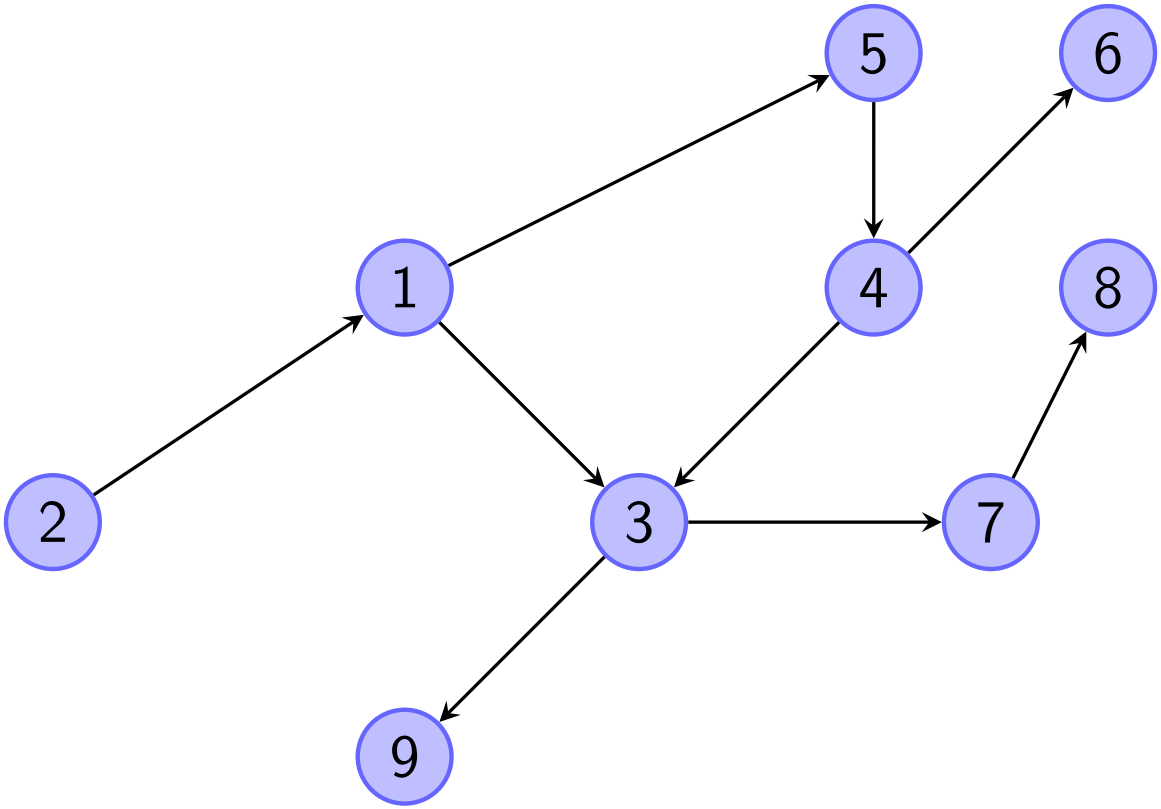
On écrit une fonction itérative en utilisant une structure de file (premier arrivé, premier sorti).

Parcours en largeur

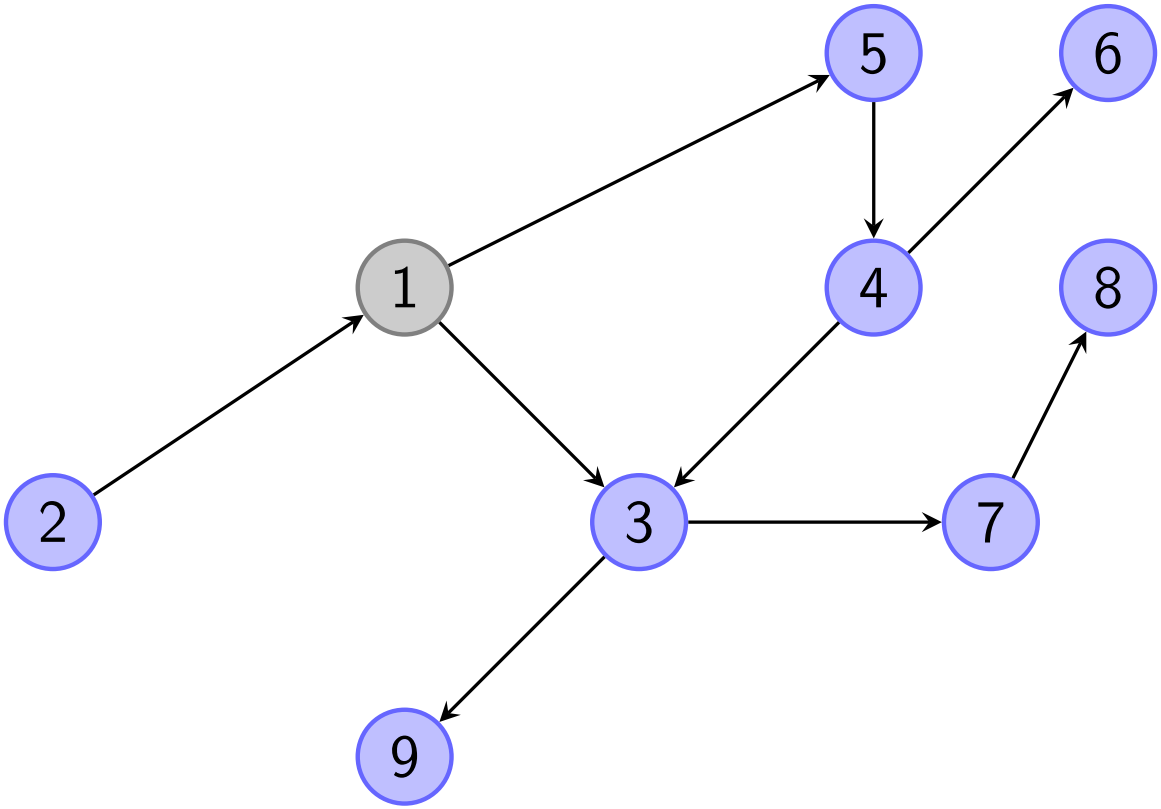
Pour un graphe sous forme de listes d'adjacence :

```
def bfs(start_node, graph):
    visited = {start_node: True}
    queue = deque([start_node])

    while queue:
        node = queue.popleft()

        for v in graph[node]:
            if v not in visited:
                visited[v] = True
                queue.append(v)
```

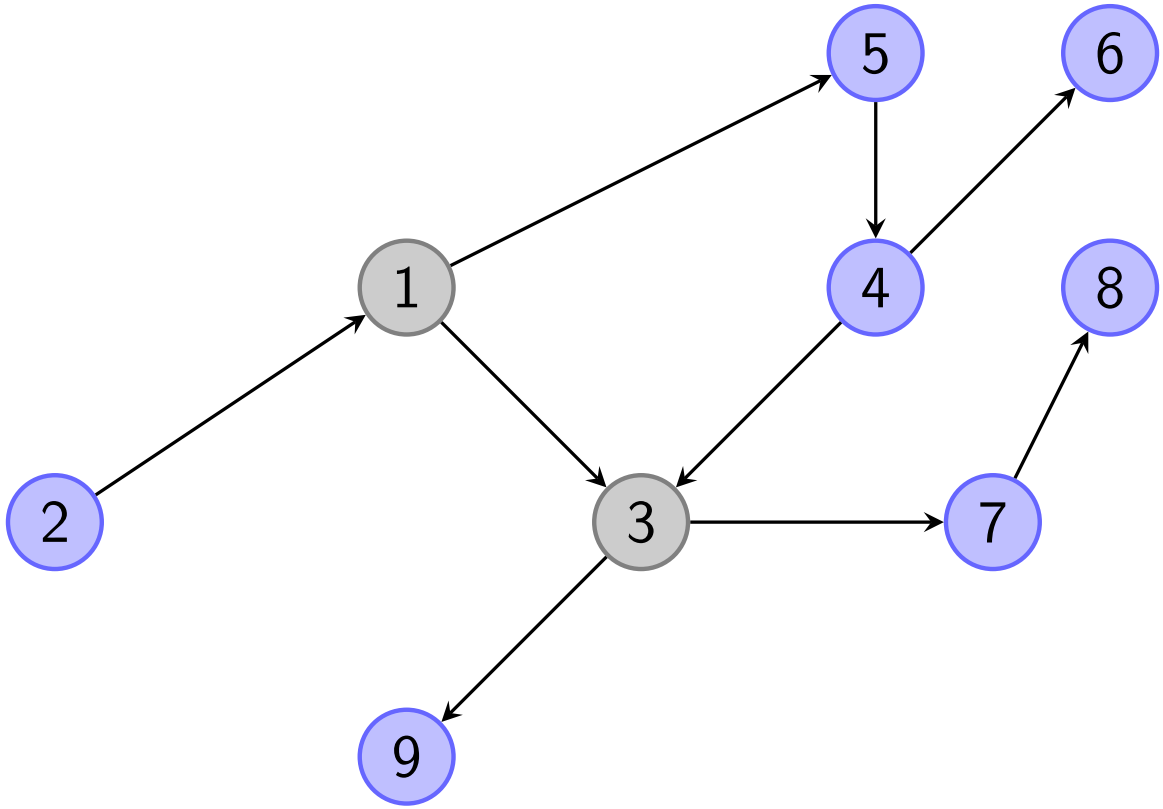
Parcours en largeur



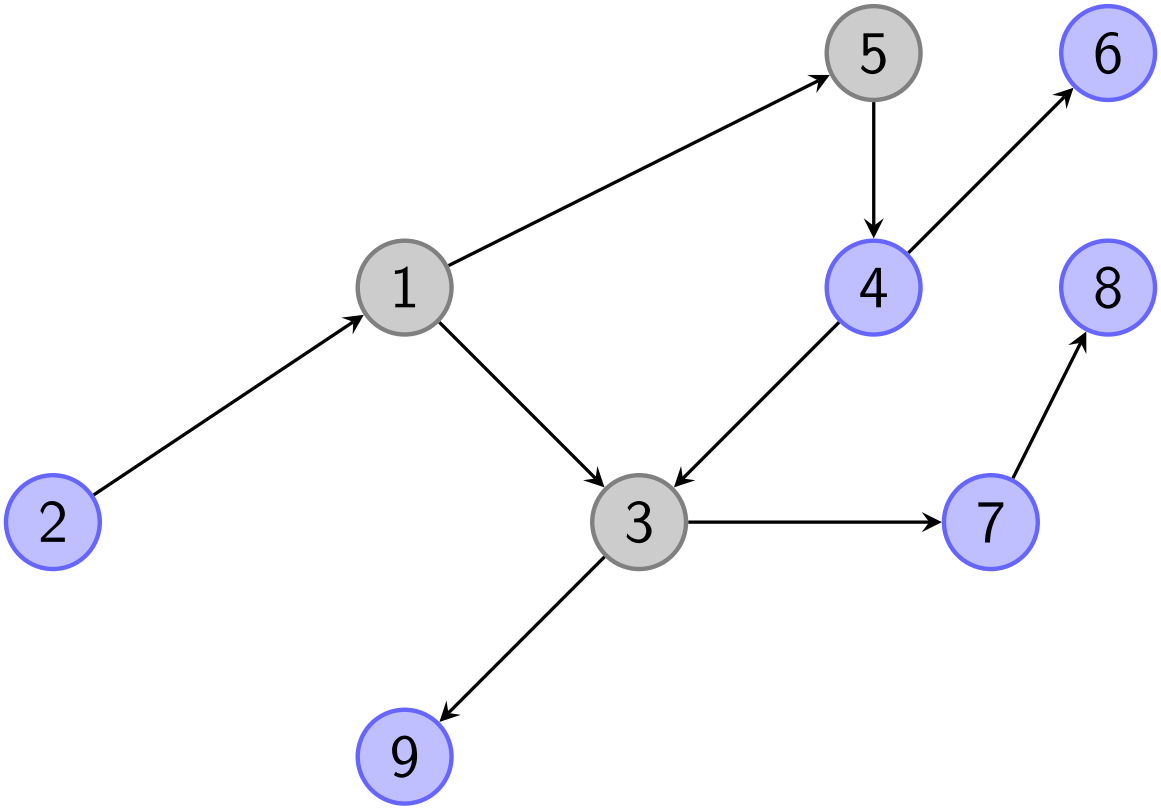
Parcours en largeur



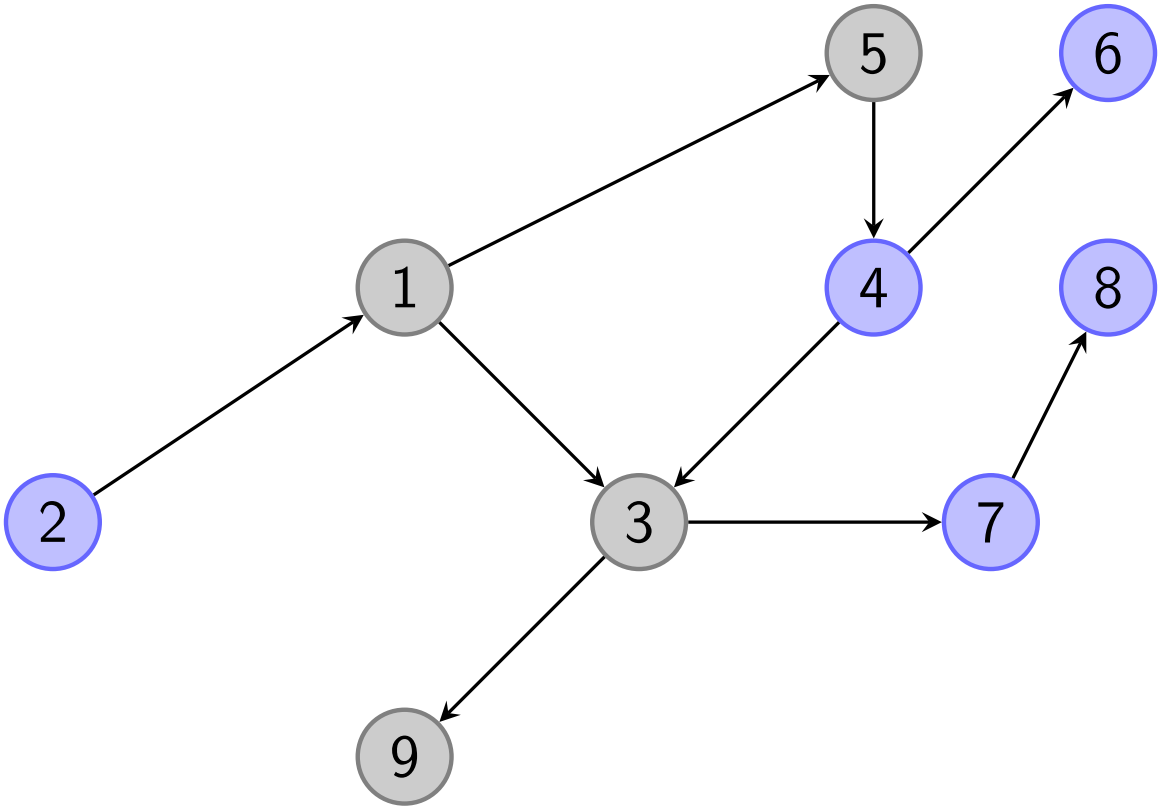
Parcours en largeur



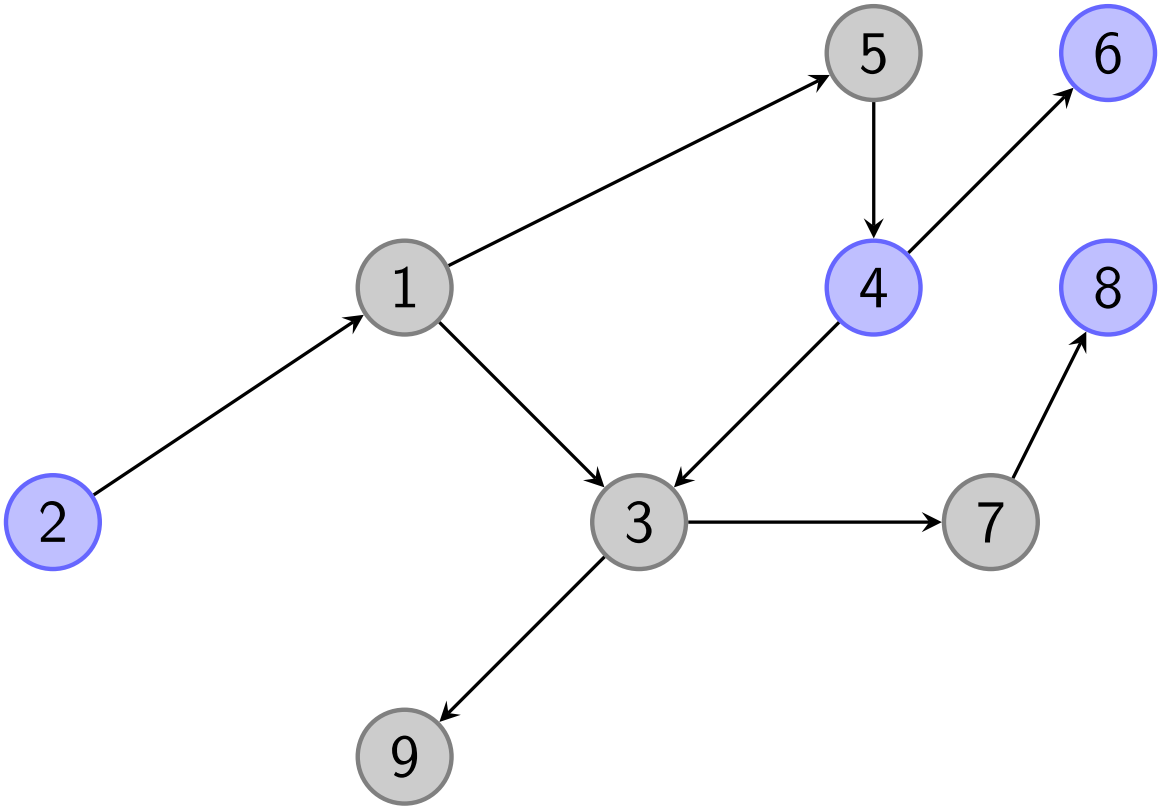
Parcours en largeur



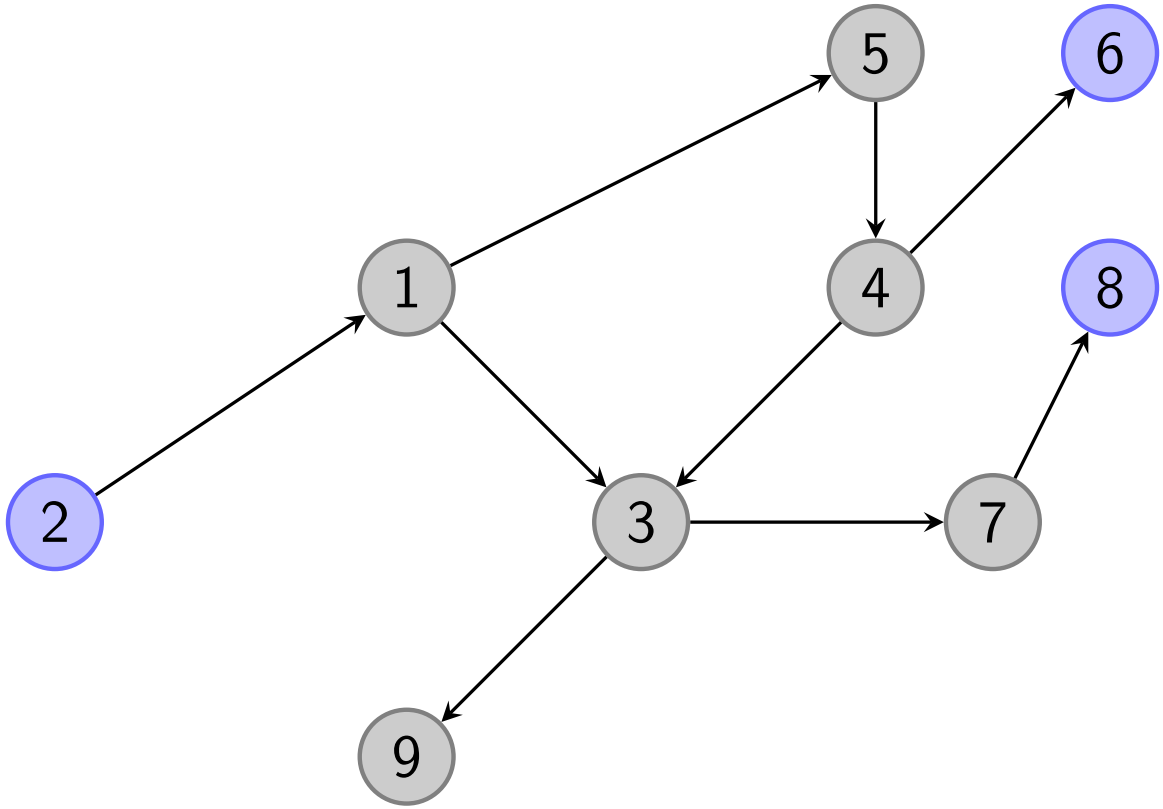
Parcours en largeur



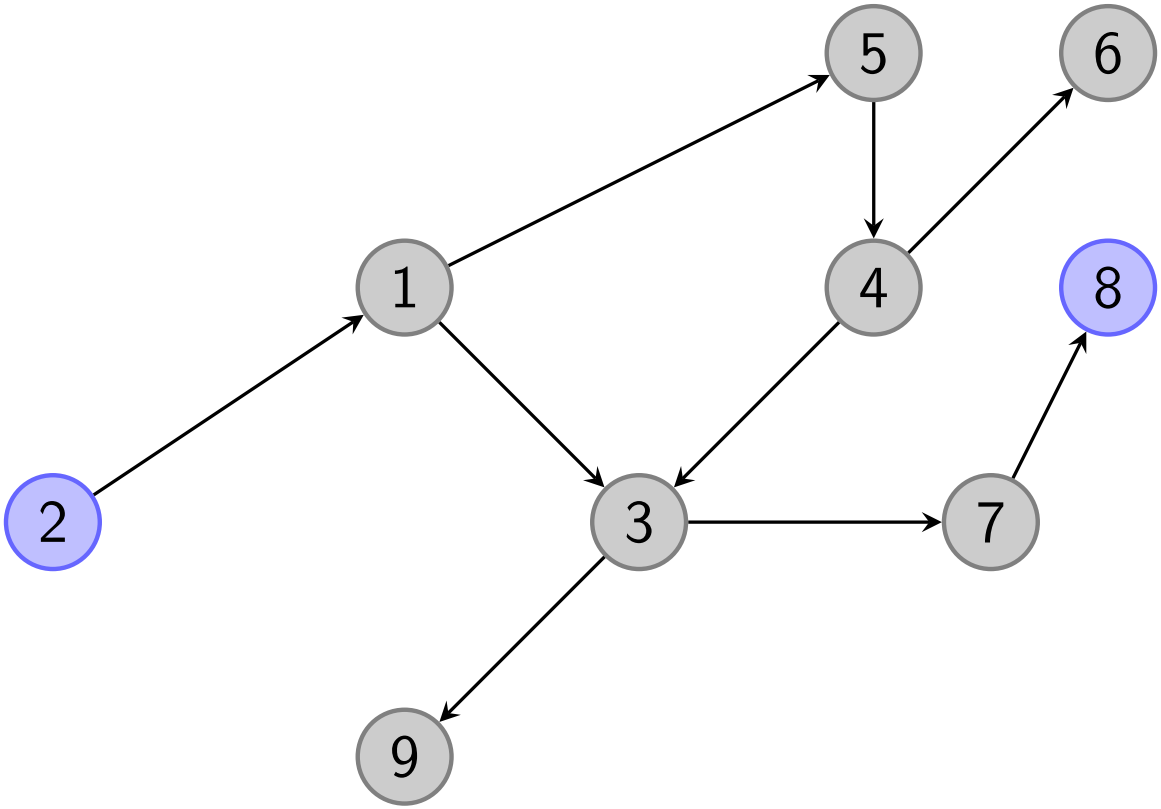
Parcours en largeur



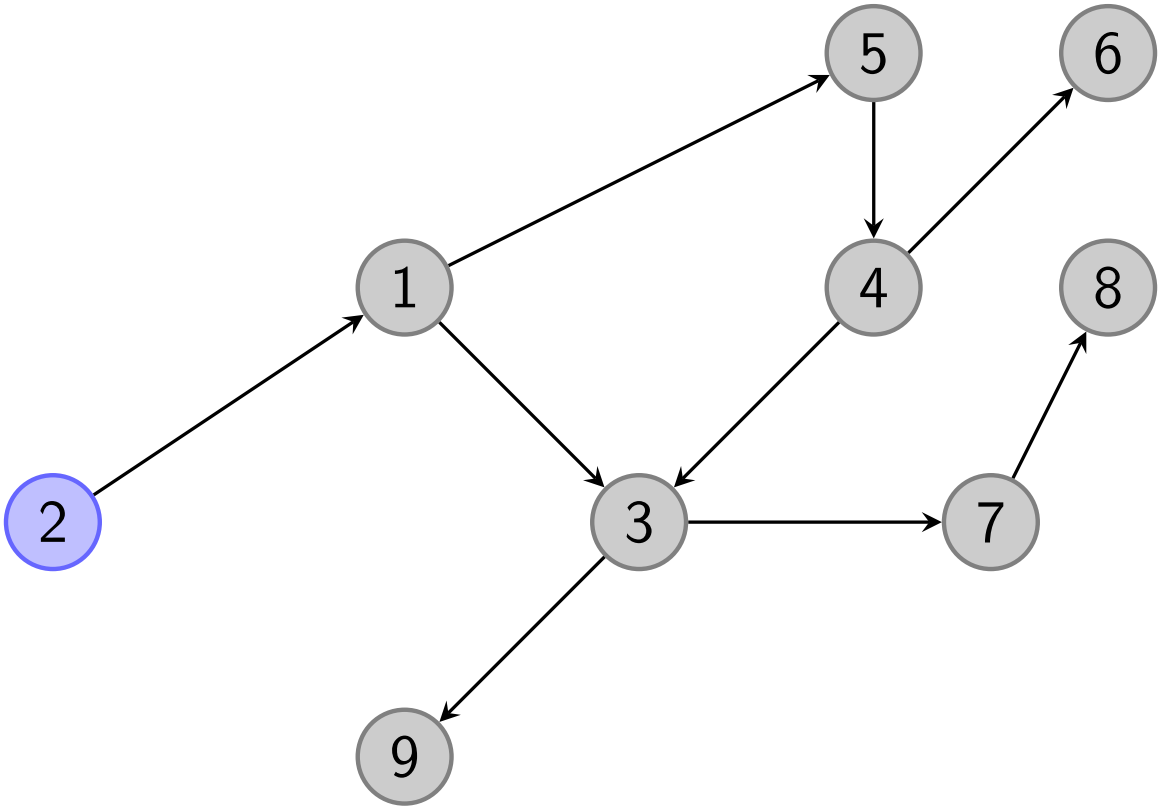
Parcours en largeur



Parcours en largeur



Parcours en largeur



Optimalité de BFS

Pour des graphes **non-pondérés**, le parcours en largeur nous donne le plus court chemin entre le nœud de départ et tous les autres nœuds du graphe.

Mais comment trouver le plus court chemin entre deux nœuds dans un graphe pondéré ?

Optimalité de Dijkstra

Pour des graphes **pondérés** à poids **positifs**, l'algorithme de Dijkstra nous donne le plus court chemin entre le nœud de départ et tous les autres nœuds du graphe.

Attention !

Si le graphe contient des arêtes de poids strictement négatif, l'algorithme de Dijkstra peut retourner un chemin non-optimal.

Algorithme de Dijkstra

Principe de l'algorithme

On fait un parcours en largeur en visitant les voisins dans un ordre qui dépend du poids des arêtes. En pratique, on remplace la file classique par une file de priorité (tas min ou tas de Fibonacci).

Complexité : $O(|E|log(|V|))$ avec un tas binaire

Algorithme de Dijkstra

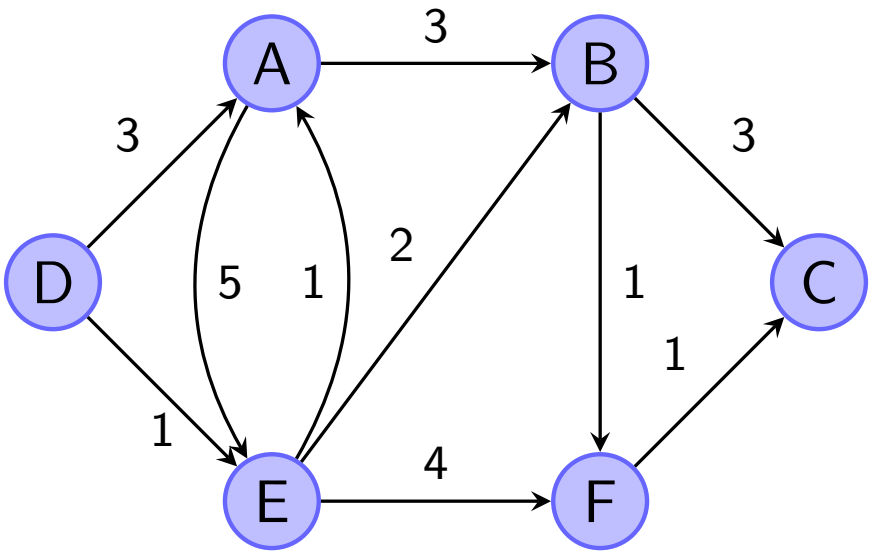
Entrées : Un graphe $G = (E, V, w)$, un sommet de départ s

Algorithme :

$distance[u] \leftarrow +\infty$ pour tout sommet $u \in V$
$distance[s] \leftarrow 0$
$parent[u] \leftarrow u$ pour tout $u \in V$
$F \leftarrow$ File de priorité vide ▷ Frontière
Enfiler(F, $(0, s)$)

tant que non EstVide(F) **faire**
    $u \leftarrow$ Defiler(F) ▷ Sommet avec $distance[u]$ minimum
    pour chaque voisin v de u **faire**
        si $distance[u] + w(u, v) < distance[v]$ **alors**
            $distance[v] \leftarrow distance[u] + w(u, v)$
            $parent[v] \leftarrow u$
            Mise à jour de v dans F avec la priorité $distance[v]$
retourner $(distance, parent)$

Algorithme de Dijkstra

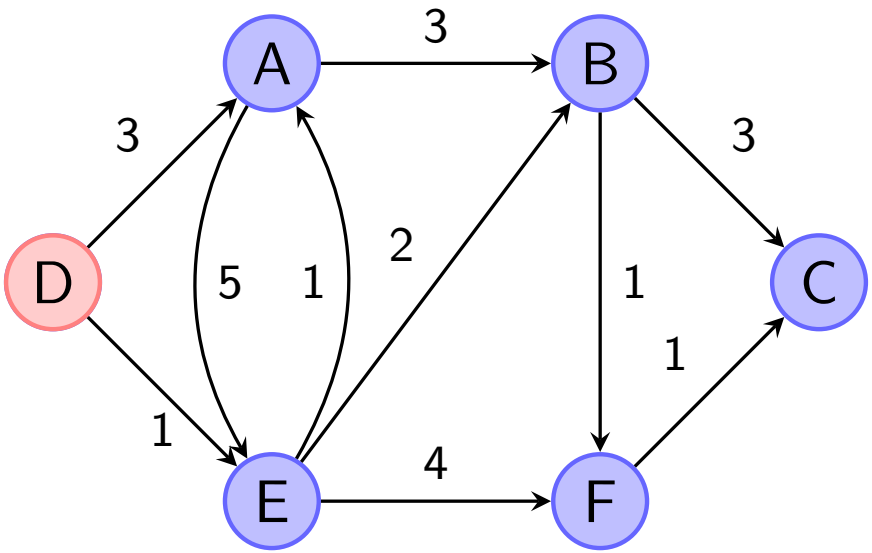


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{D\}$

u =

Algorithme de Dijkstra

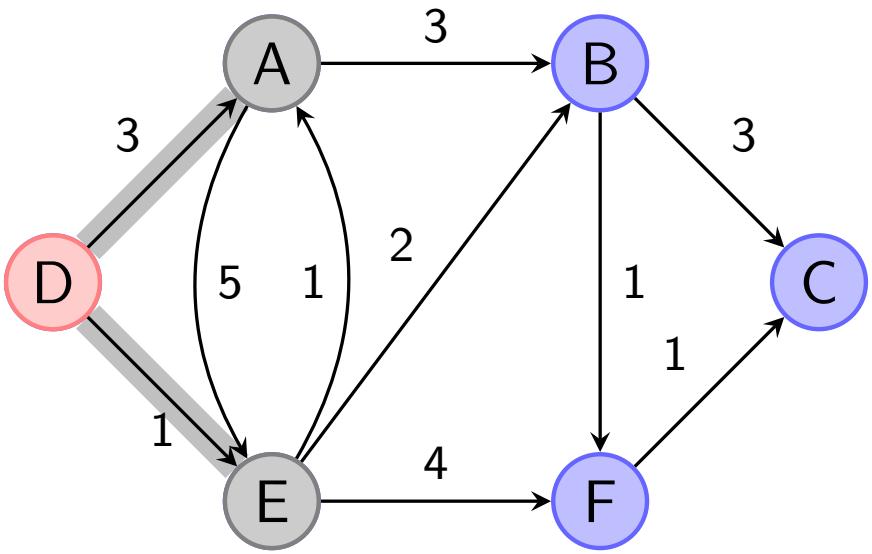


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{\}$

u = D

Algorithme de Dijkstra

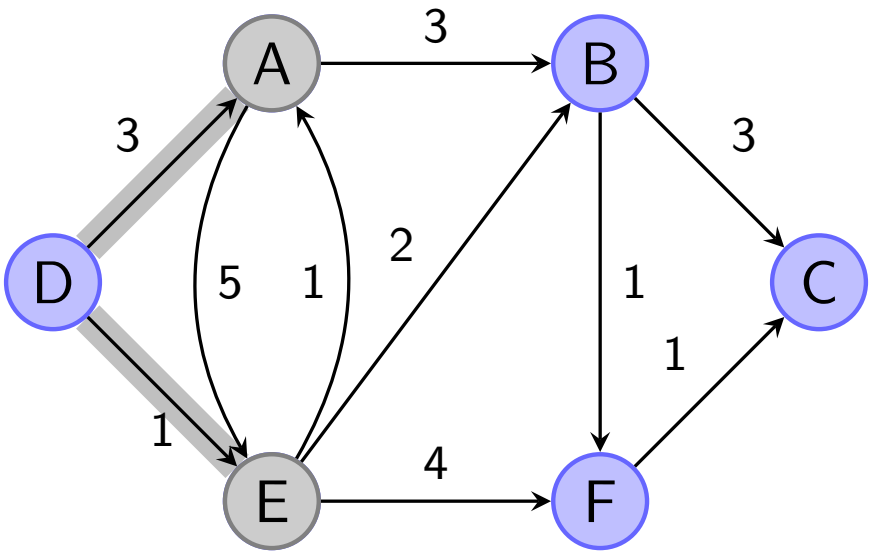


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{A, E\}$

u = D

Algorithme de Dijkstra

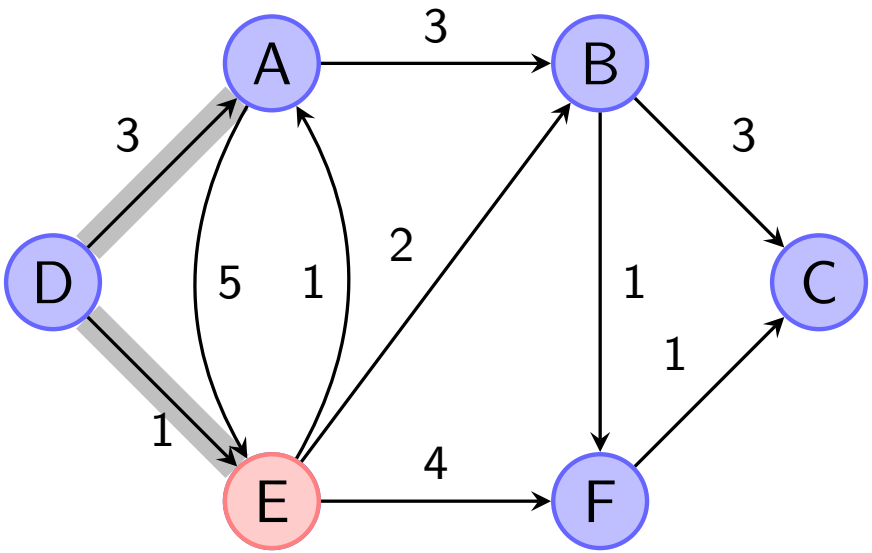


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A, E\}$

u = D

Algorithme de Dijkstra

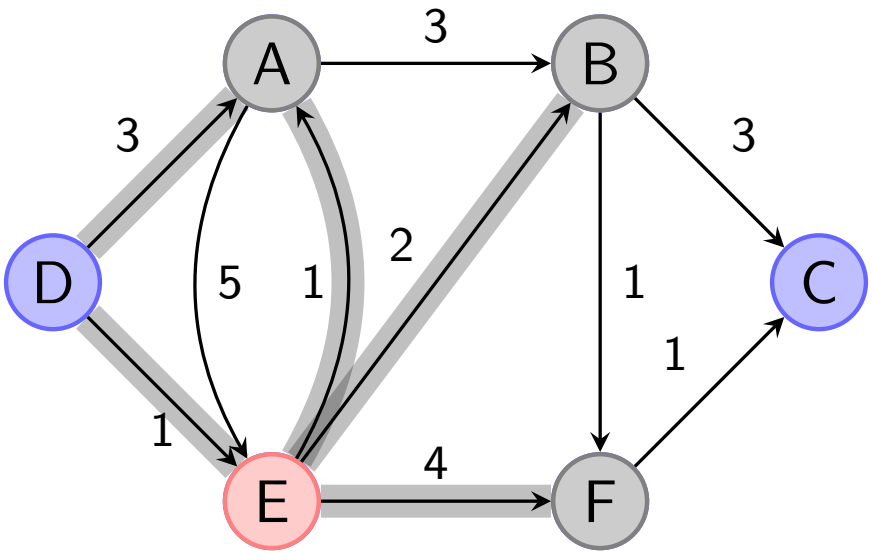


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A\}$

u = E

Algorithme de Dijkstra

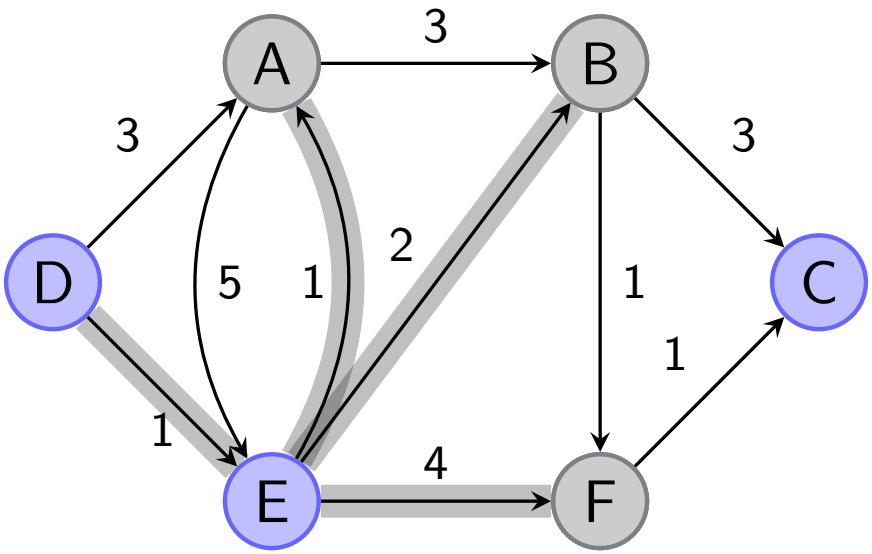


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A, B, F\}$

u = E

Algorithme de Dijkstra

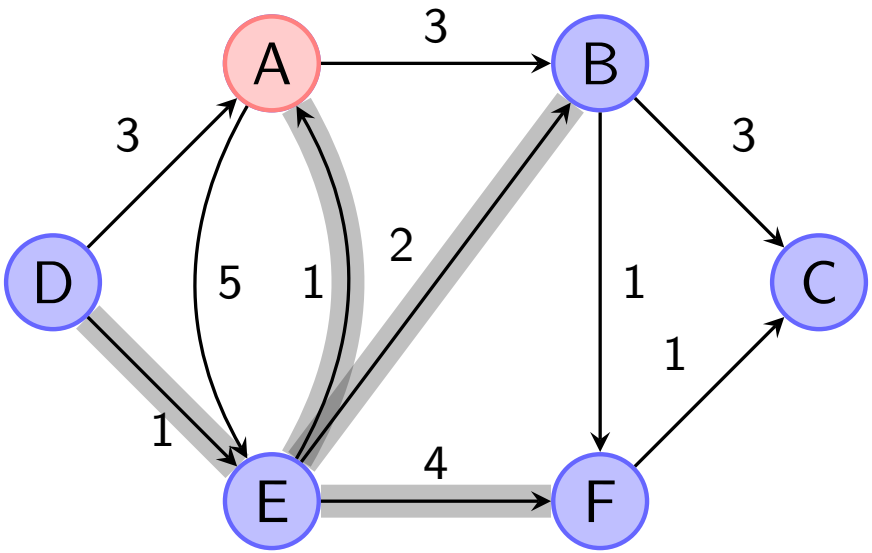


Nœud	Distance	Parent
A	1+1	E
B	1+2	E
C		
D	0	•
E	1	D
F	1+4	E

Frontière = $\{A, B, F\}$

u = E

Algorithme de Dijkstra

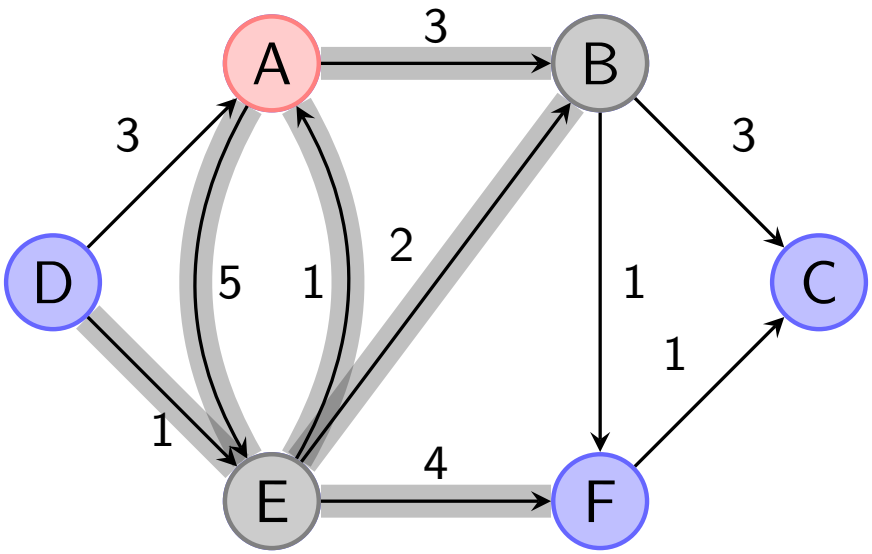


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

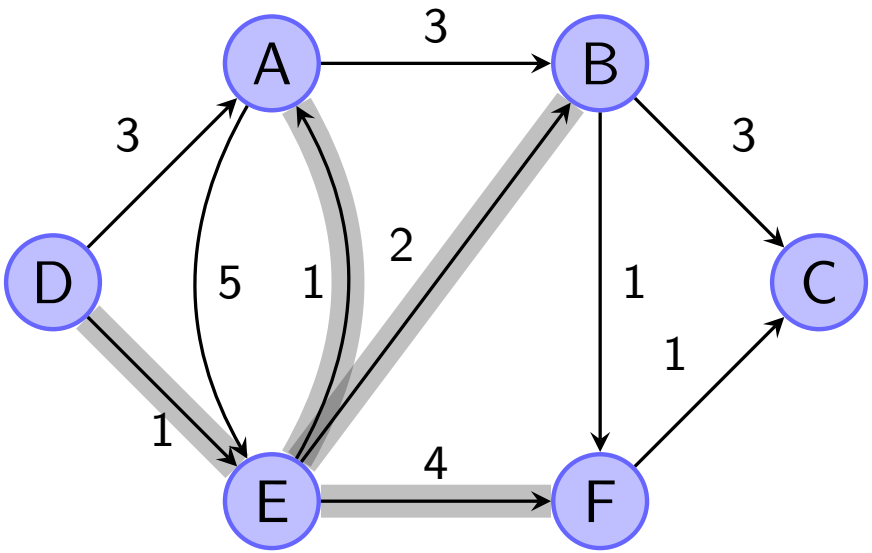


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

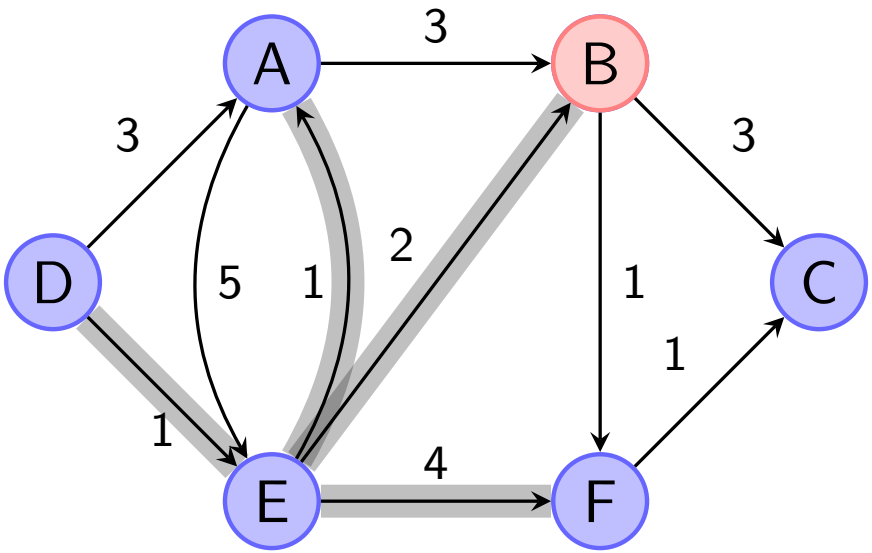


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

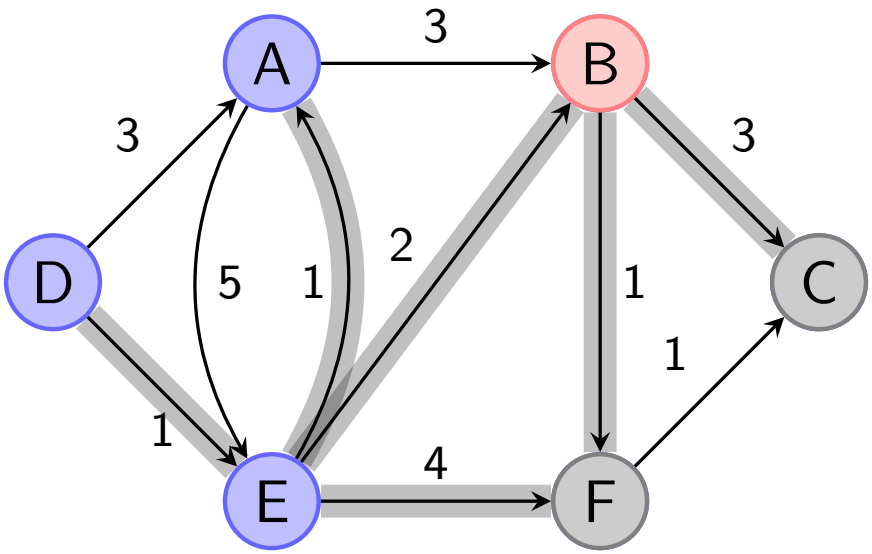


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{F\}$

u = B

Algorithme de Dijkstra

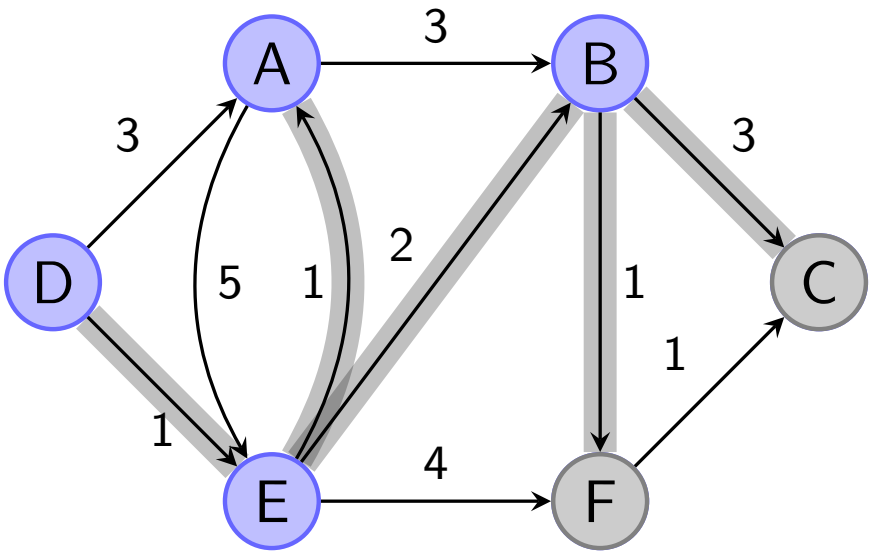


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{C, F\}$

u = B

Algorithme de Dijkstra

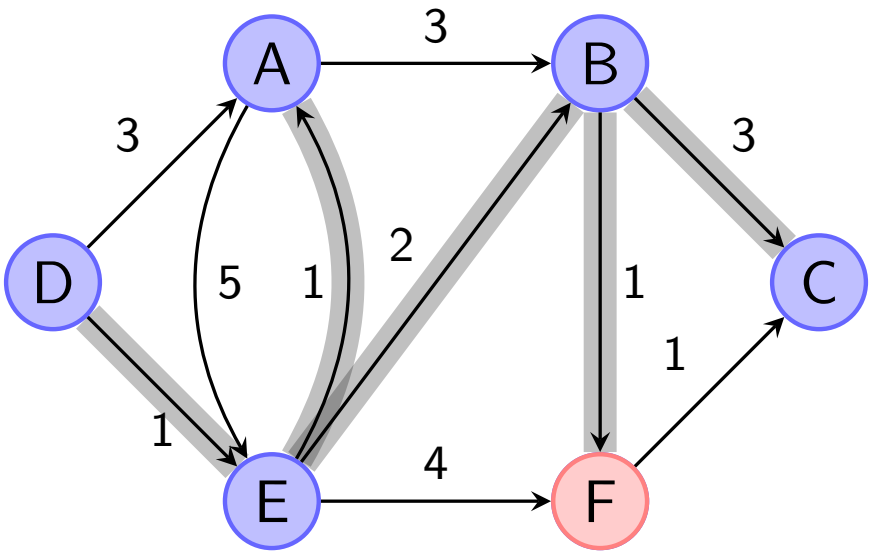


Nœud	Distance	Parent
A	2	E
B	3	E
C	3+3	B
D	0	•
E	1	D
F	3+1	B

Frontière = $\{C, F\}$

u = B

Algorithme de Dijkstra

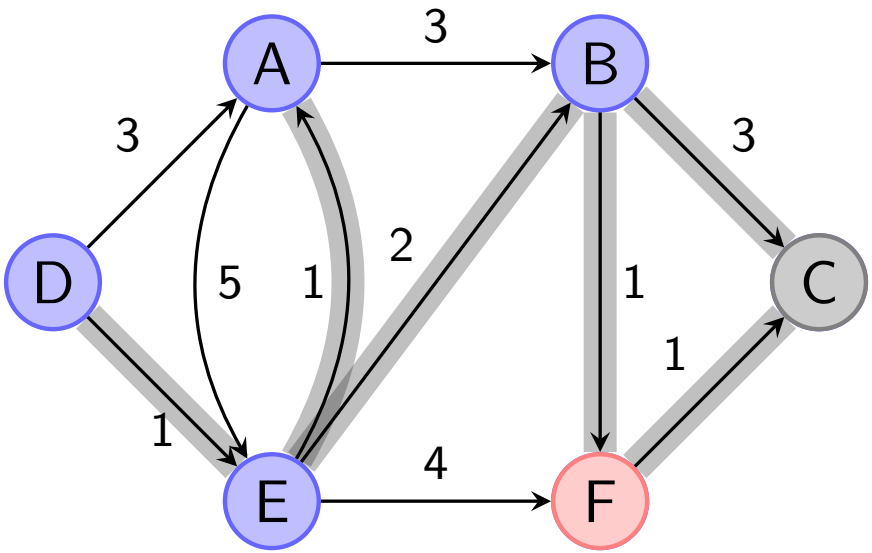


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

u = F

Algorithme de Dijkstra

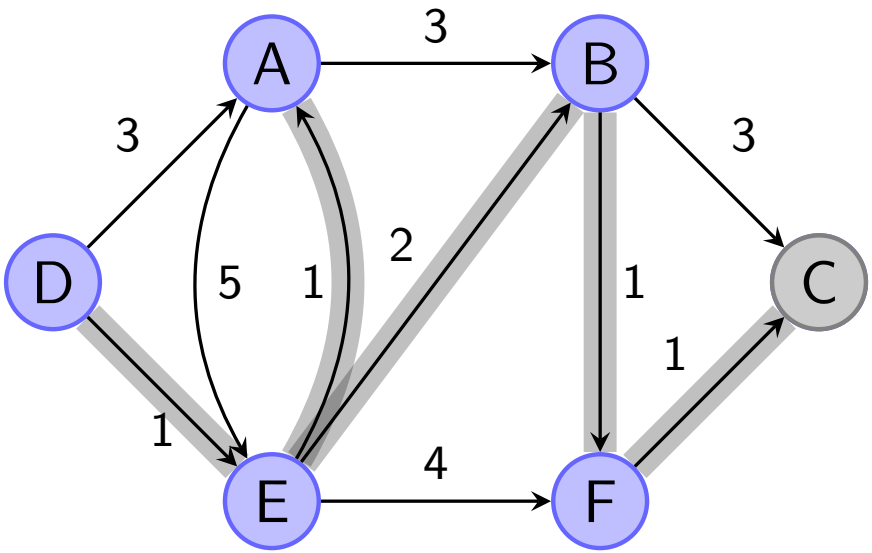


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

u = F

Algorithme de Dijkstra

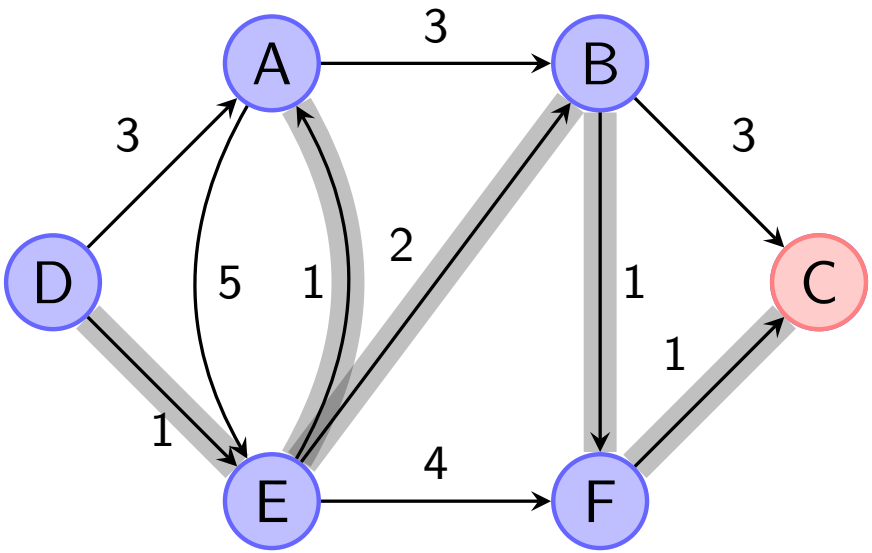


Nœud	Distance	Parent
A	2	E
B	3	E
C	4+1	F
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

u = F

Algorithme de Dijkstra



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

Frontière = $\{\}$

u = C

Arbre couvrant minimal (MST)

Soit un graphe non-orienté, connexe $G = (V, E)$. Un arbre couvrant minimal de G (Minimal Spanning Tree) est un sous-graphe de G contenant tous les sommets de G et tel que la somme des poids de ses arêtes est minimal.

Arbre couvrant minimal (MST)

Soit un graphe non-orienté, connexe $G = (V, E)$. Un arbre couvrant minimal de G (*Minimum Spanning Tree*) est un sous-graphe acyclique de G contenant tous les sommets de G et tel que la somme des poids de ses arêtes est minimal.

Kruskal

L'algorithme de Kruskal permet de trouver un arbre couvrant minimal dans un graphe connexe.

Complexité : $O(|E|log|V|)$ avec une structure Unir-Trouver

L'algorithme de Prim permet également de trouver un MST mais il est un peu plus complexe.

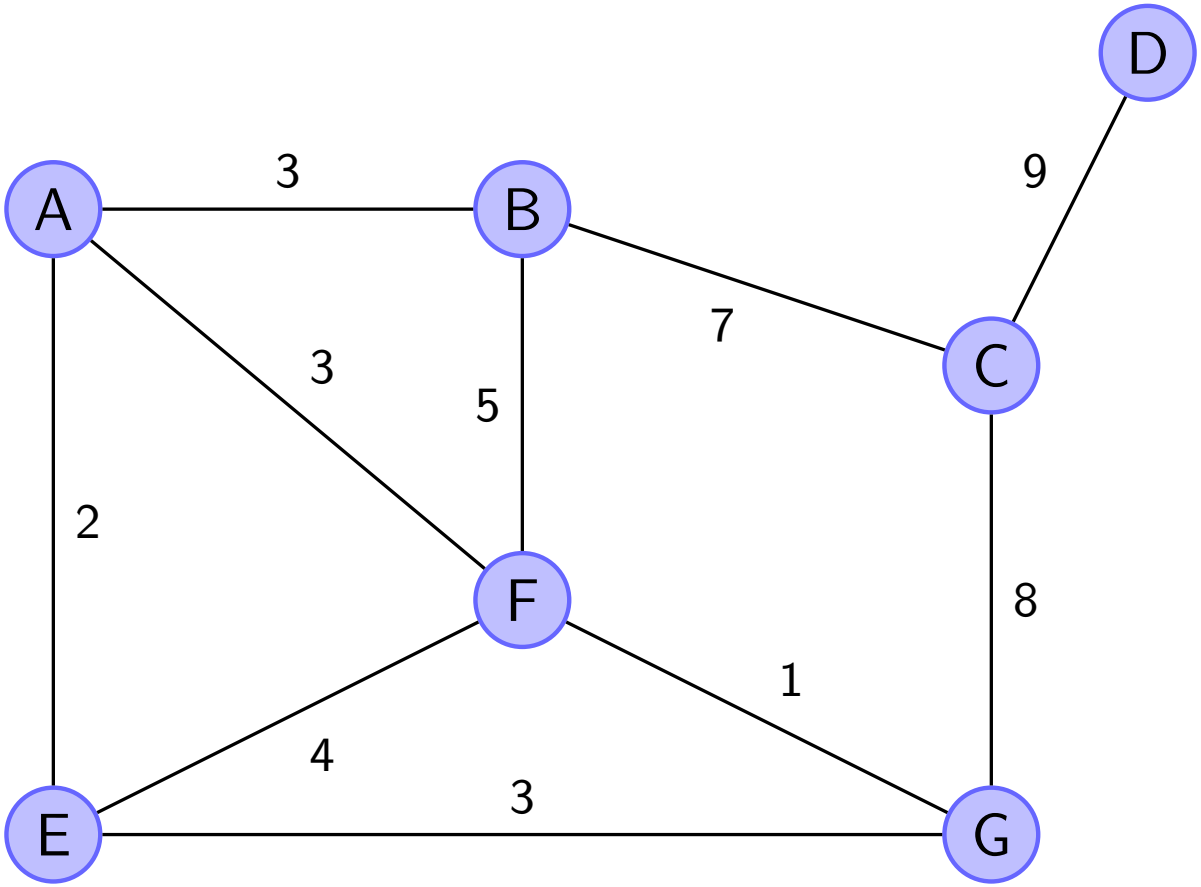
Algorithme de Kruskal

Entrées : Un graphe connexe pondéré $G = (E, V, w)$
Sortie : Un ensemble $T \subseteq E$ décrivant un MST

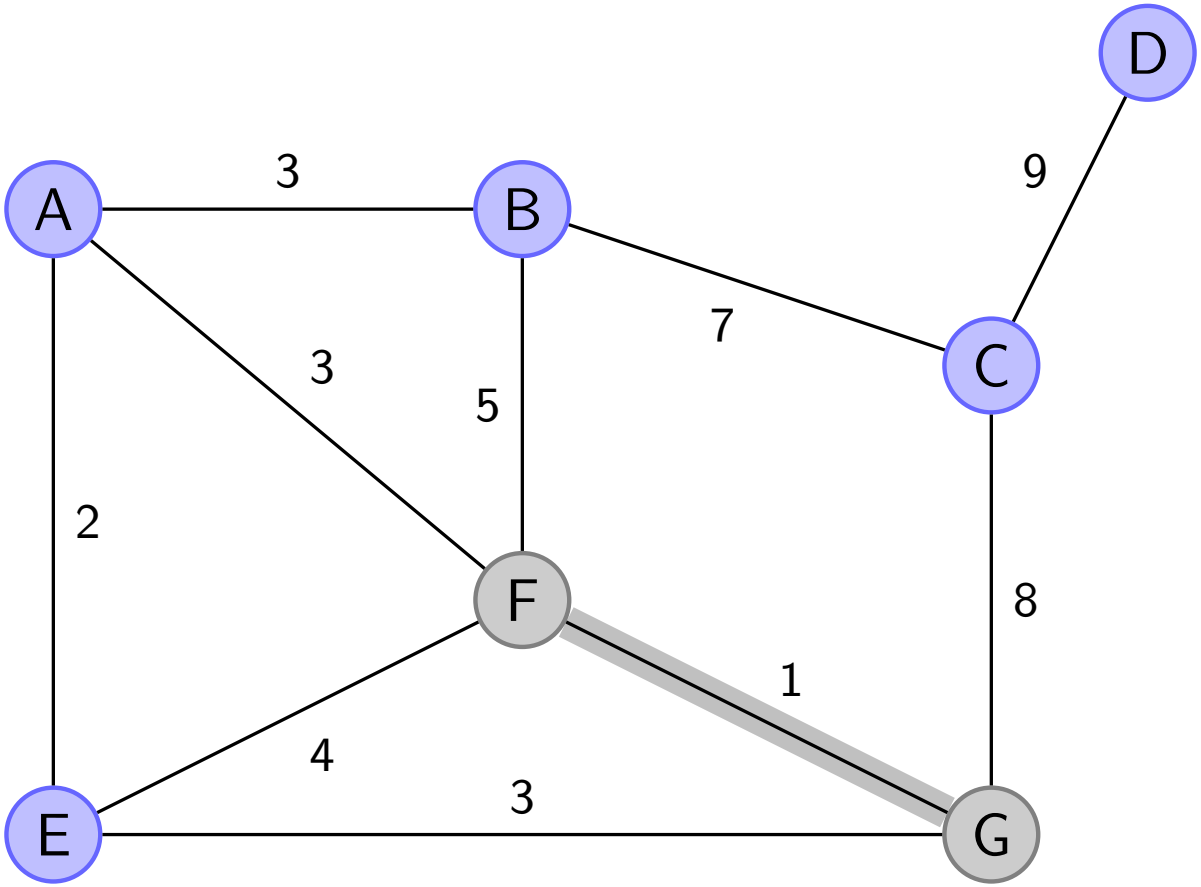
Algorithme :

$T \leftarrow \emptyset$
$P \leftarrow$ **Init**$(|S|)$ ▷ Structure de données Unir-Trouver
Trier les arêtes de E par poids croissant
pour $(u, v) \in E$ dans l'ordre **faire**
    si **Trouver**$(P, u) \neq$ **Trouver**(P, v) **alors**
        Ajouter (u, v) à T
        Unir(P, u, v)
retourner T

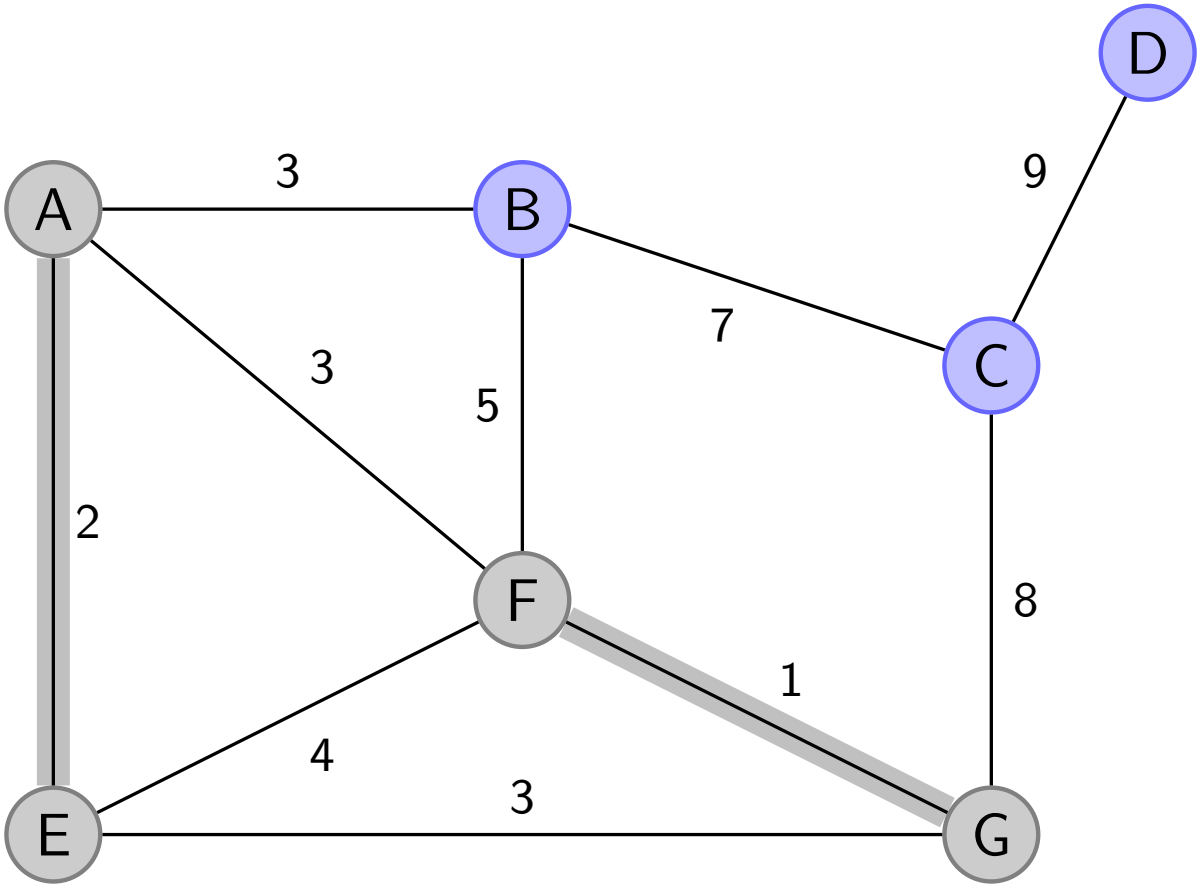
Algorithme de Kruskal



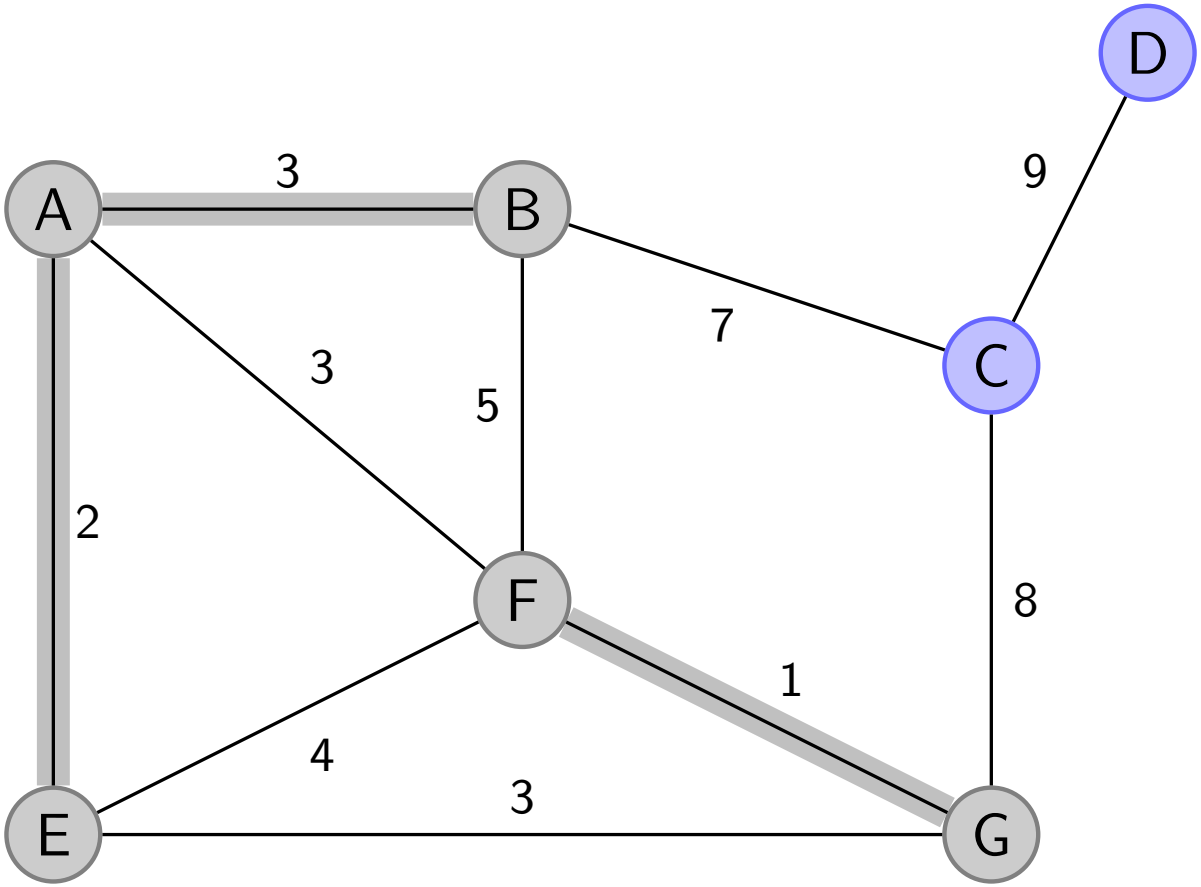
Algorithme de Kruskal



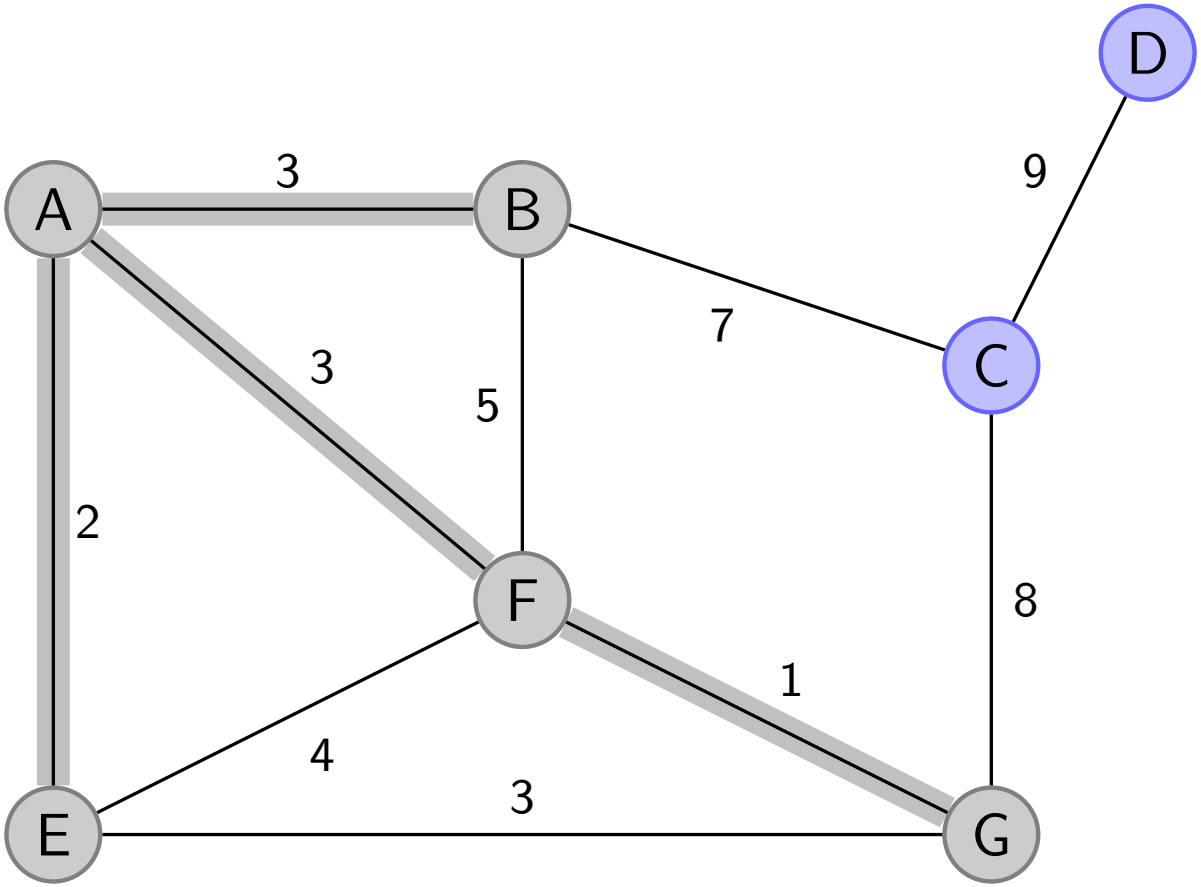
Algorithme de Kruskal



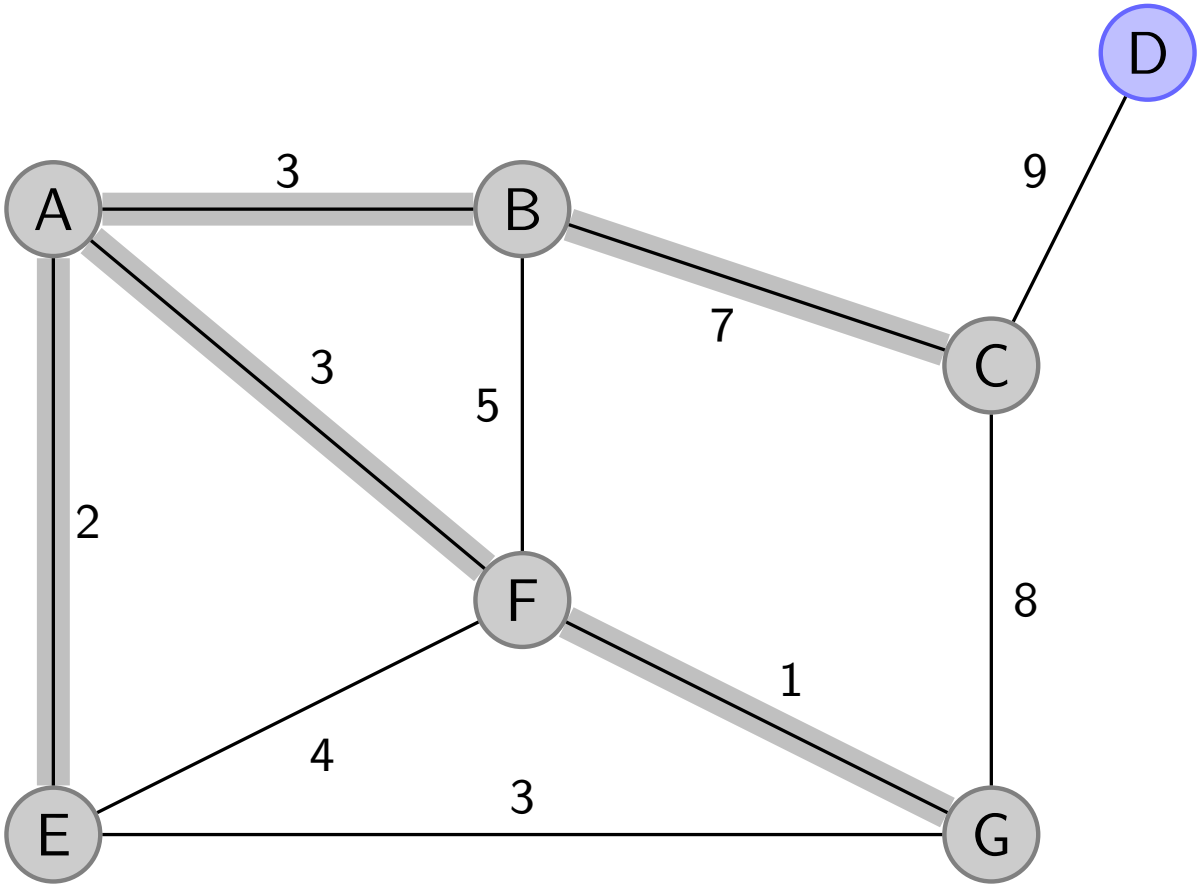
Algorithme de Kruskal



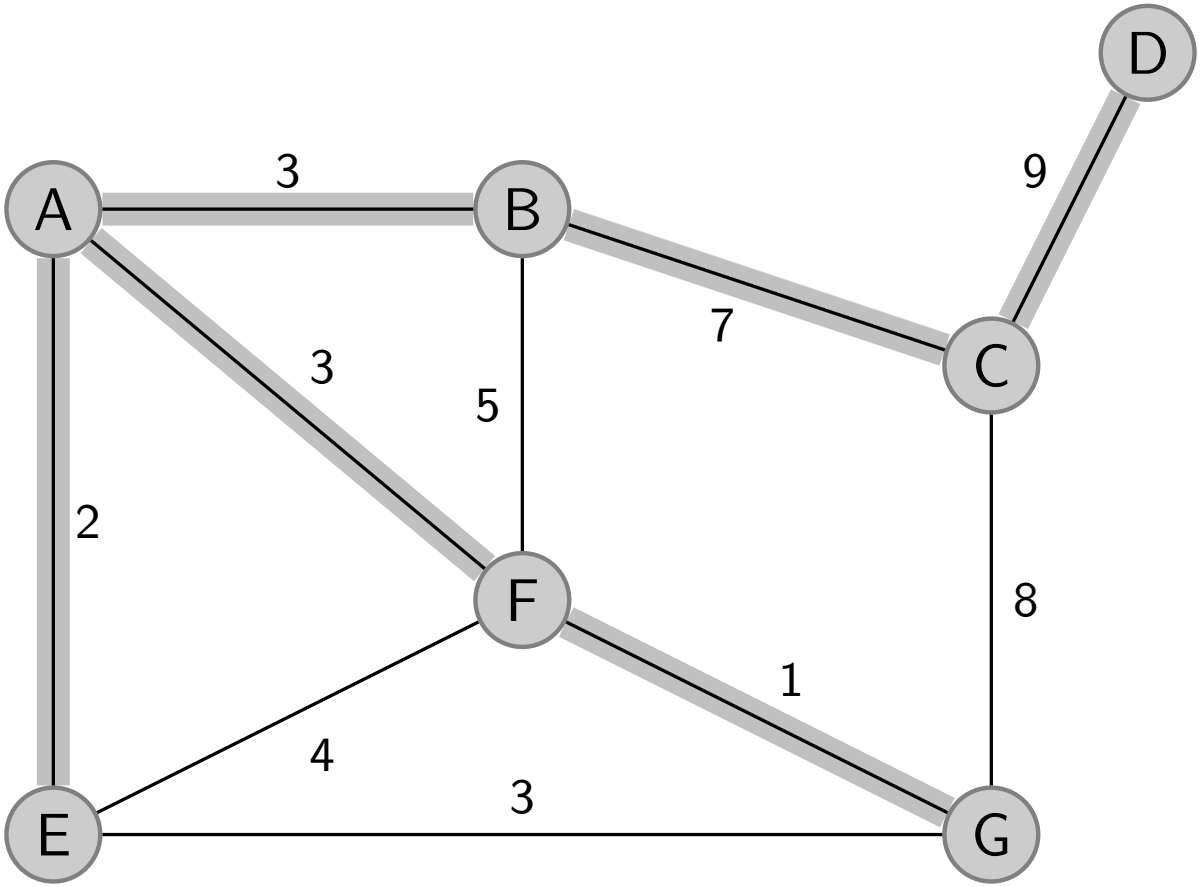
Algorithme de Kruskal



Algorithme de Kruskal



Algorithme de Kruskal



Conclusion de la section 1

Choses à savoir faire pour l'examen :

- Établir la complexité d'un algorithme
- Raisonner sur les propriétés des graphes
- Écrire des algorithmes impliquant BFS/DFS
- Faire tourner Dijkstra à la main sur un exemple
- Faire tourner Kruskal à la main sur un exemple pour trouver un MST