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.

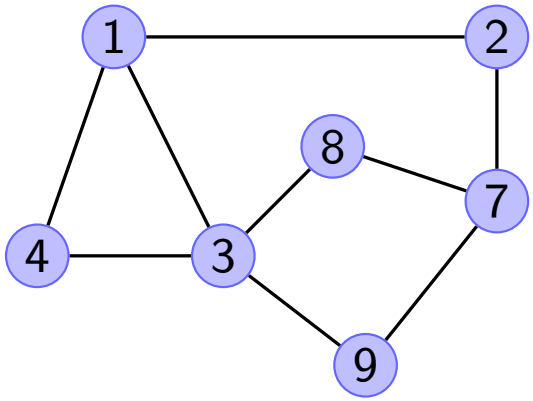
Attention

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.

Définition - Graphe non-orienté

On note $G = (V, E)$ où :

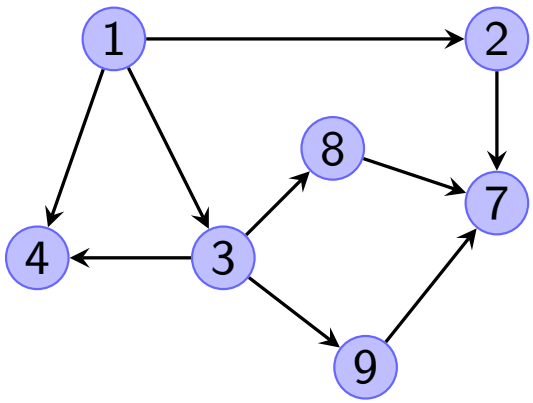
- V est un ensemble de nœuds ;
- E est un ensemble d'arêtes ;
- Une arête $e = (u, v)$ est une paire d'éléments de V.



Définition - Graphe orienté

On note $G = (V, E)$ où :

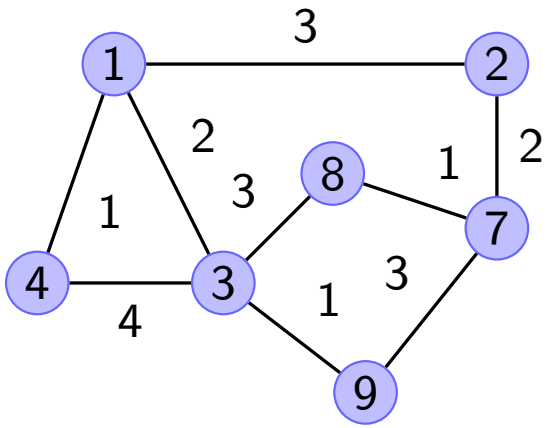
- V est un ensemble de nœuds ;
- E est un ensemble d'arcs ;
- Un arc $e = (u, v)$ est un couple d'éléments de V.



Définition - Graphe pondéré

On note $G = (V, E, w)$ où :

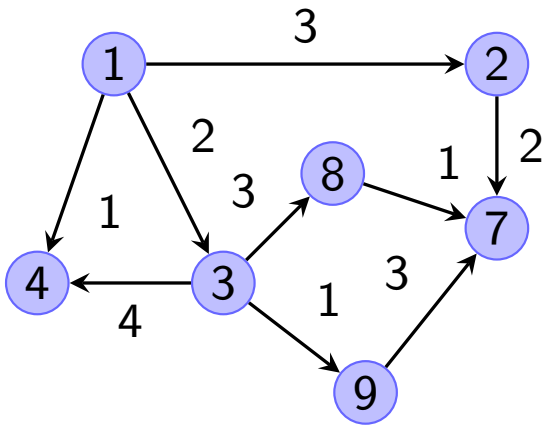
- V est un ensemble de nœuds ;
- E est un ensemble d'arêtes ;
- Une arête $e = (u, v)$ est une paire d'éléments de V.
- $w : E \to \mathbb{R}$ est une fonction de pondération



Définition - Graphe orienté pondéré

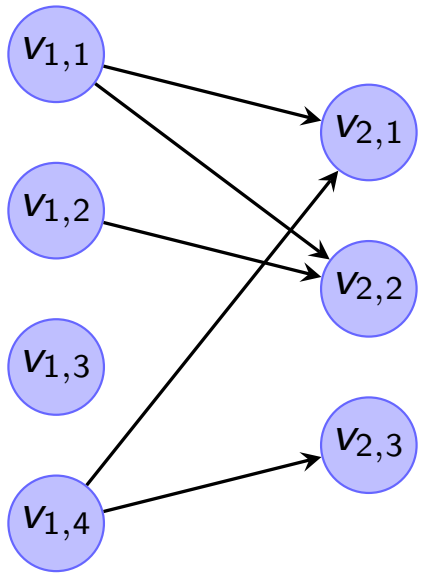
On note $G = (V, E, w)$ où :

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



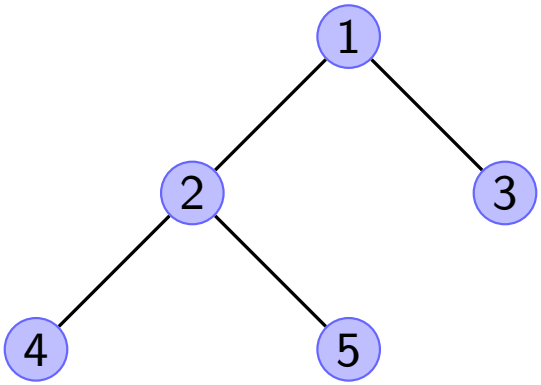
Graphe biparti

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 :

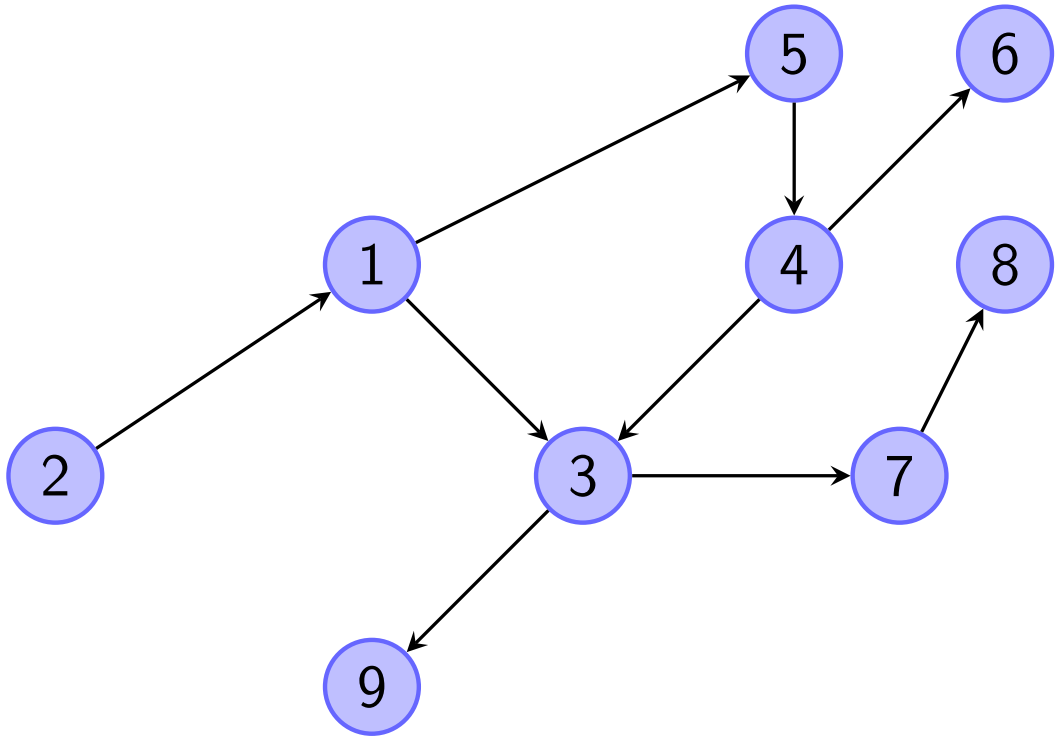
```
def dfs(start_node, graph):

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

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

    visit(start_node)
```

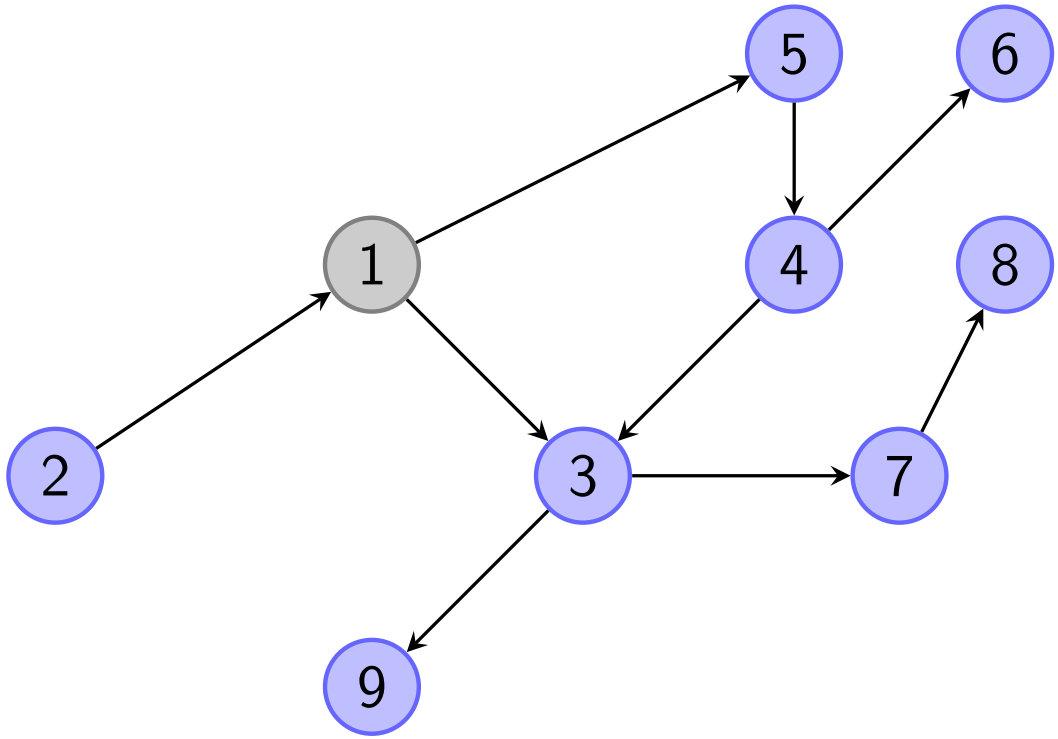
Parcours en profondeur



$\emptyset$

Pile

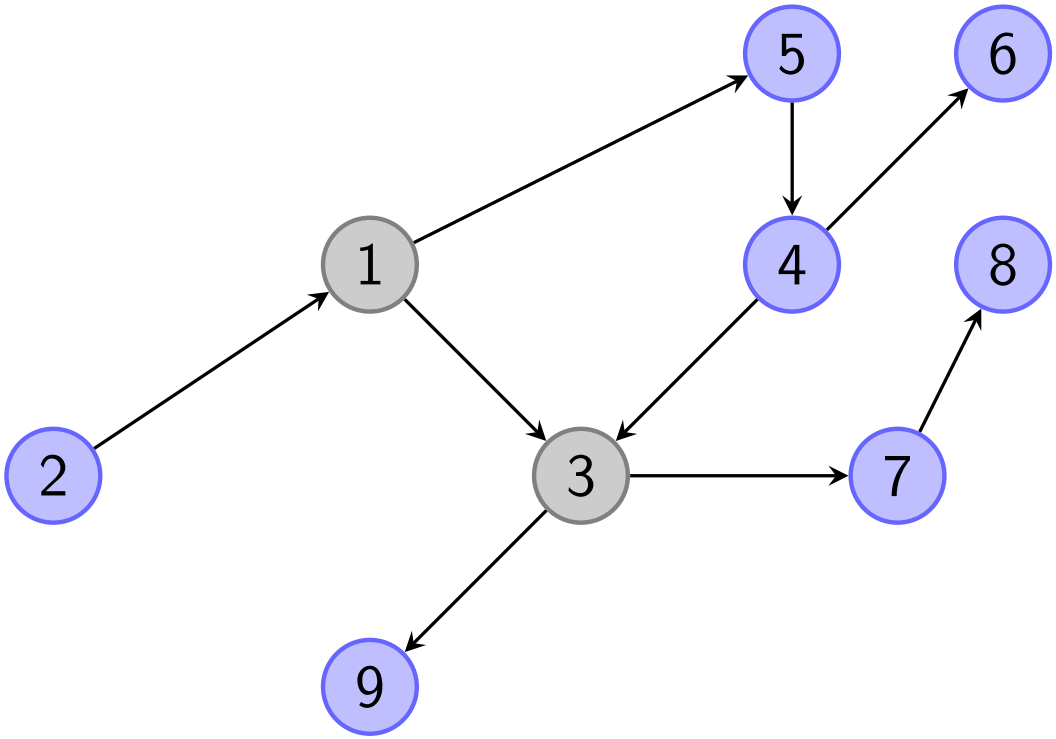
Parcours en profondeur

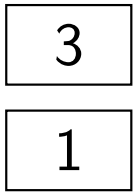


1

Pile

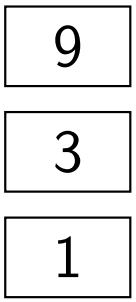
Parcours en profondeur



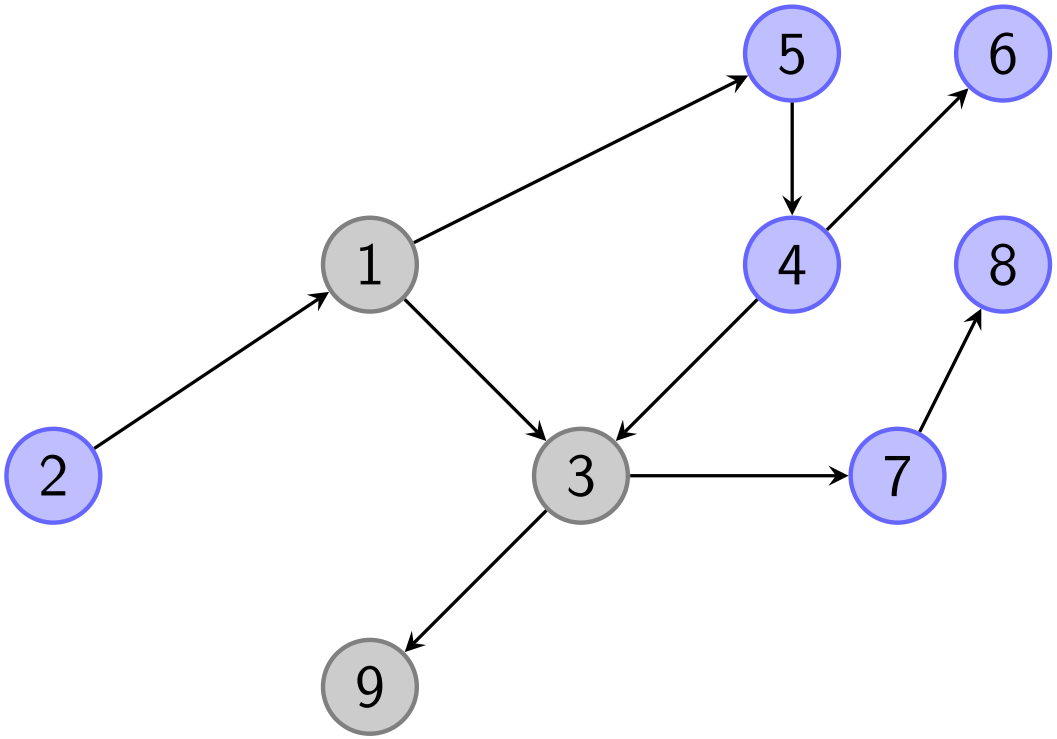


Pile

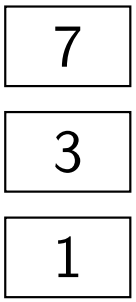
Parcours en profondeur



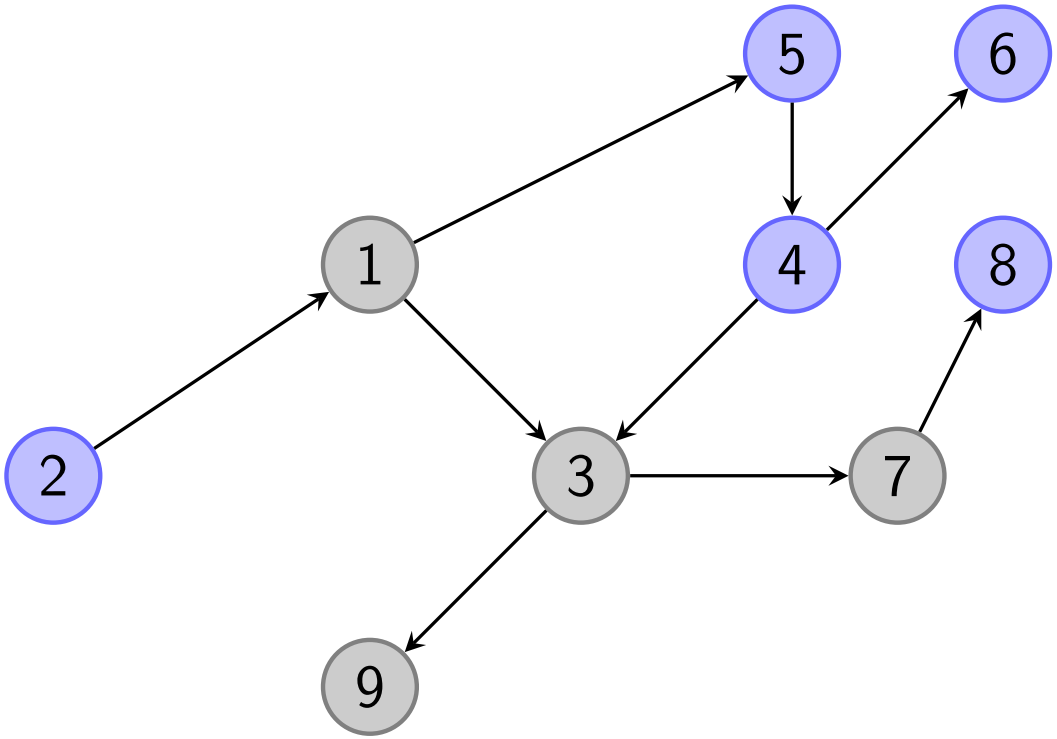
Pile



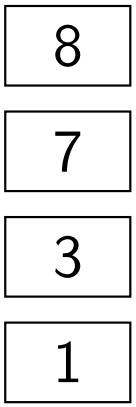
Parcours en profondeur



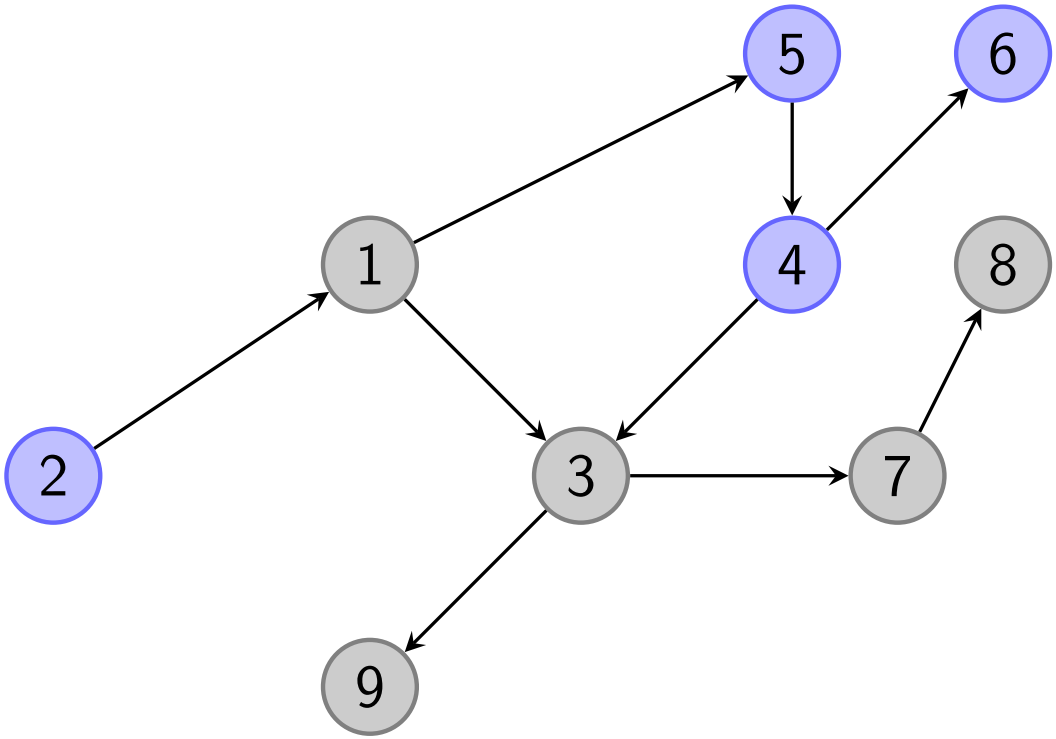
Pile



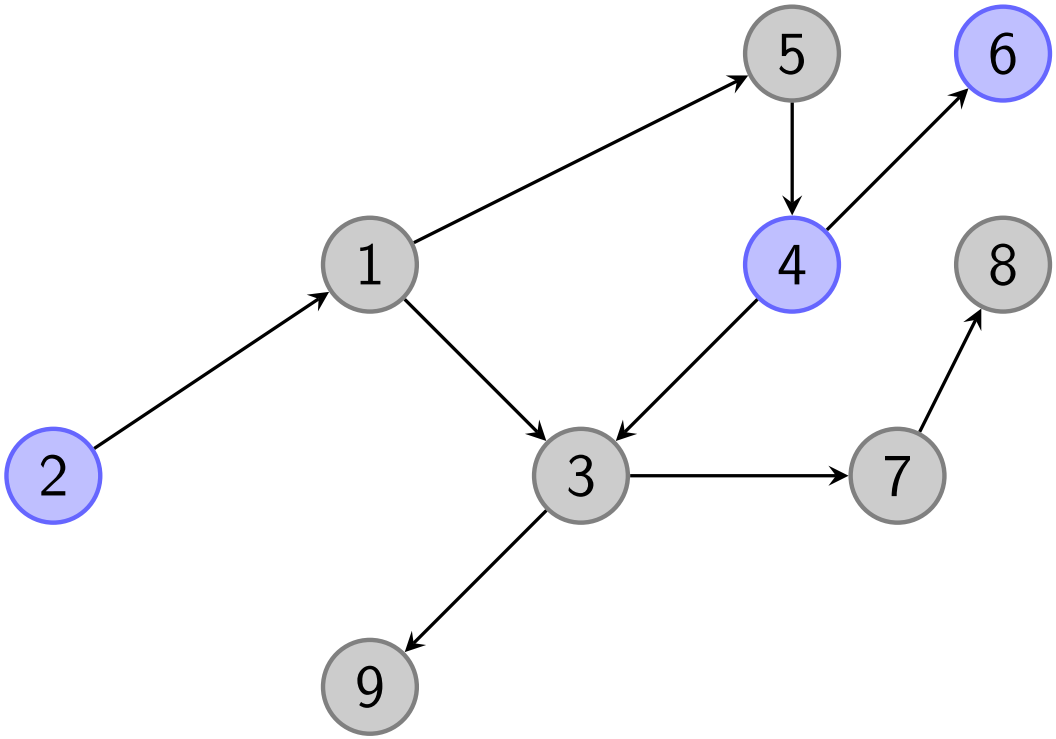
Parcours en profondeur



Pile



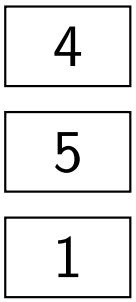
Parcours en profondeur



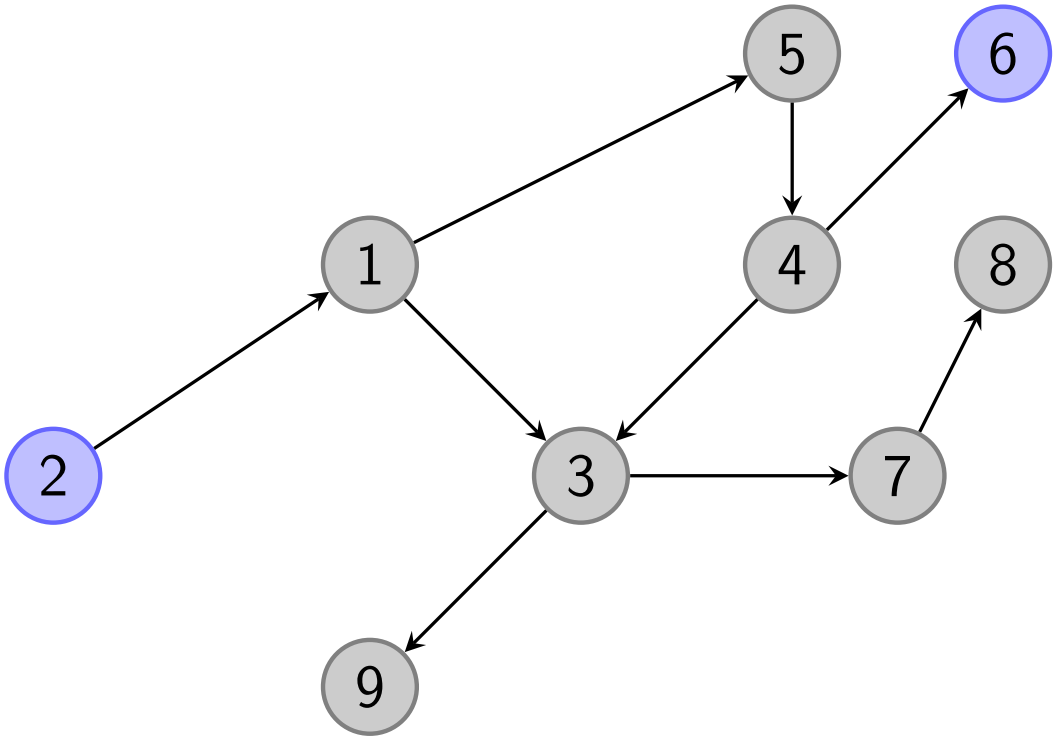
5
1

Pile

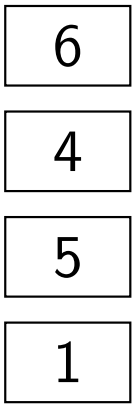
Parcours en profondeur



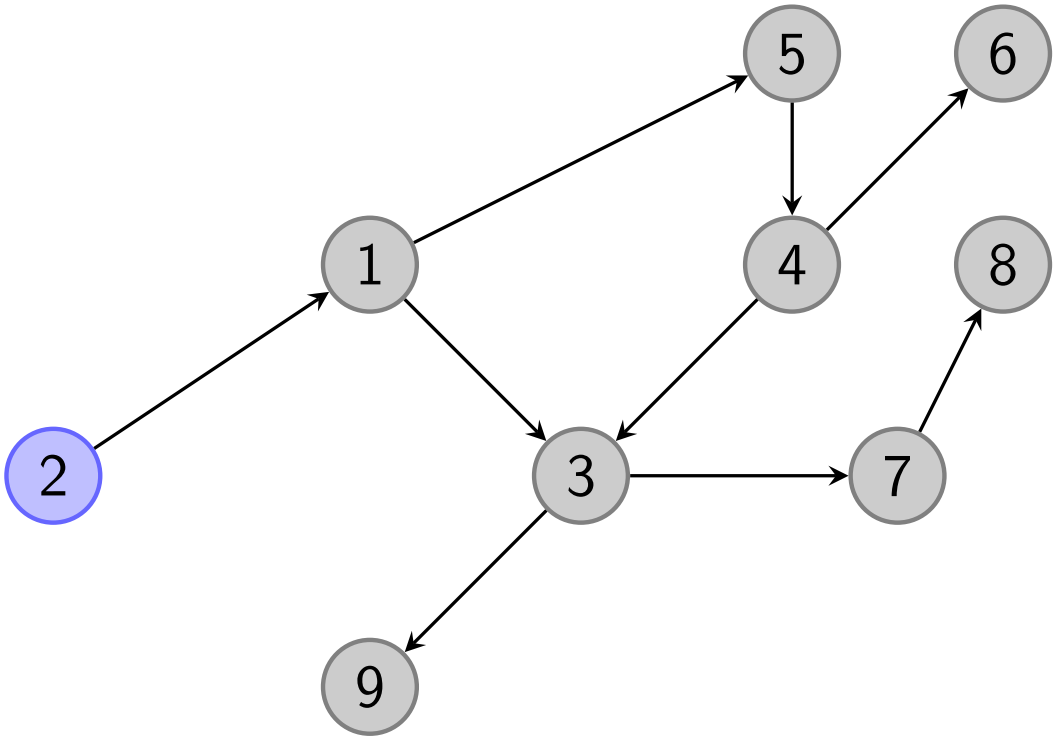
Pile



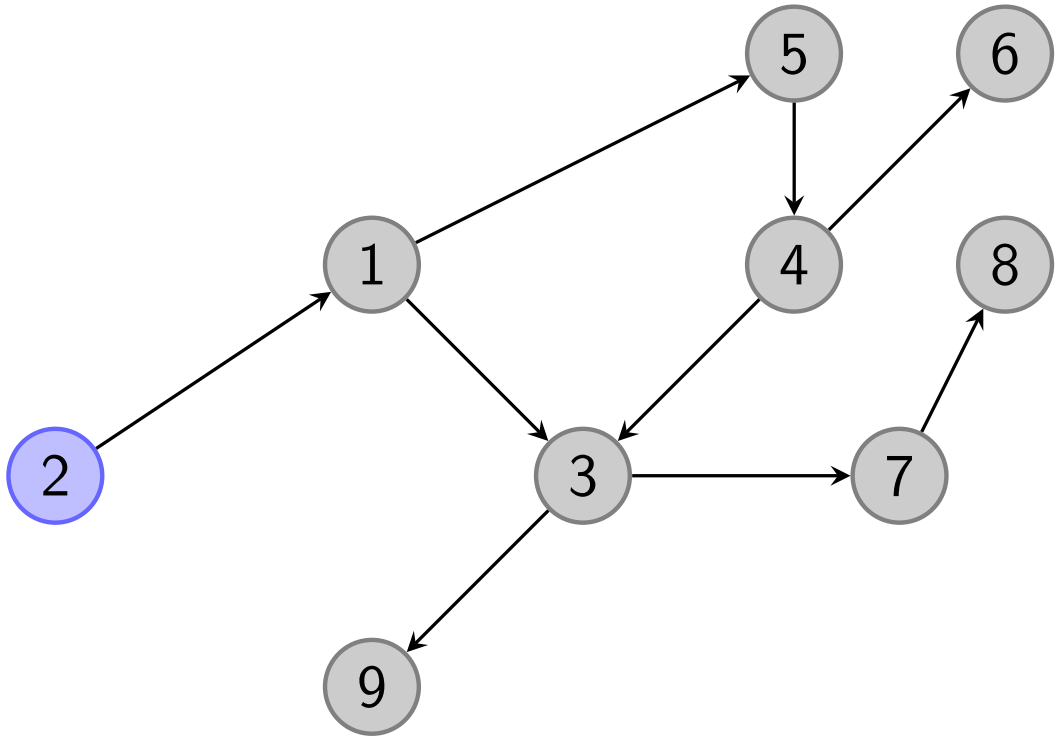
Parcours en profondeur



Pile



Parcours en profondeur



$\emptyset$

Pile

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

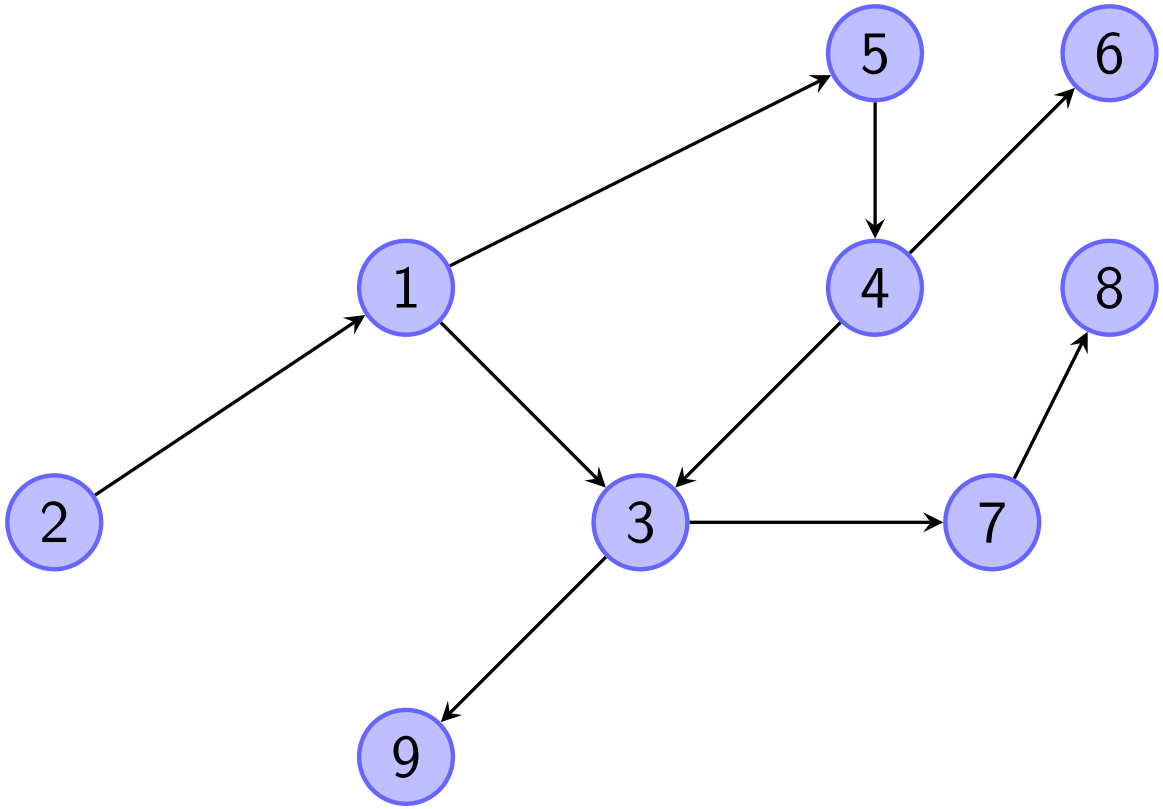
Pour un graphe sous forme de listes d'adjacence :

```
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)
```

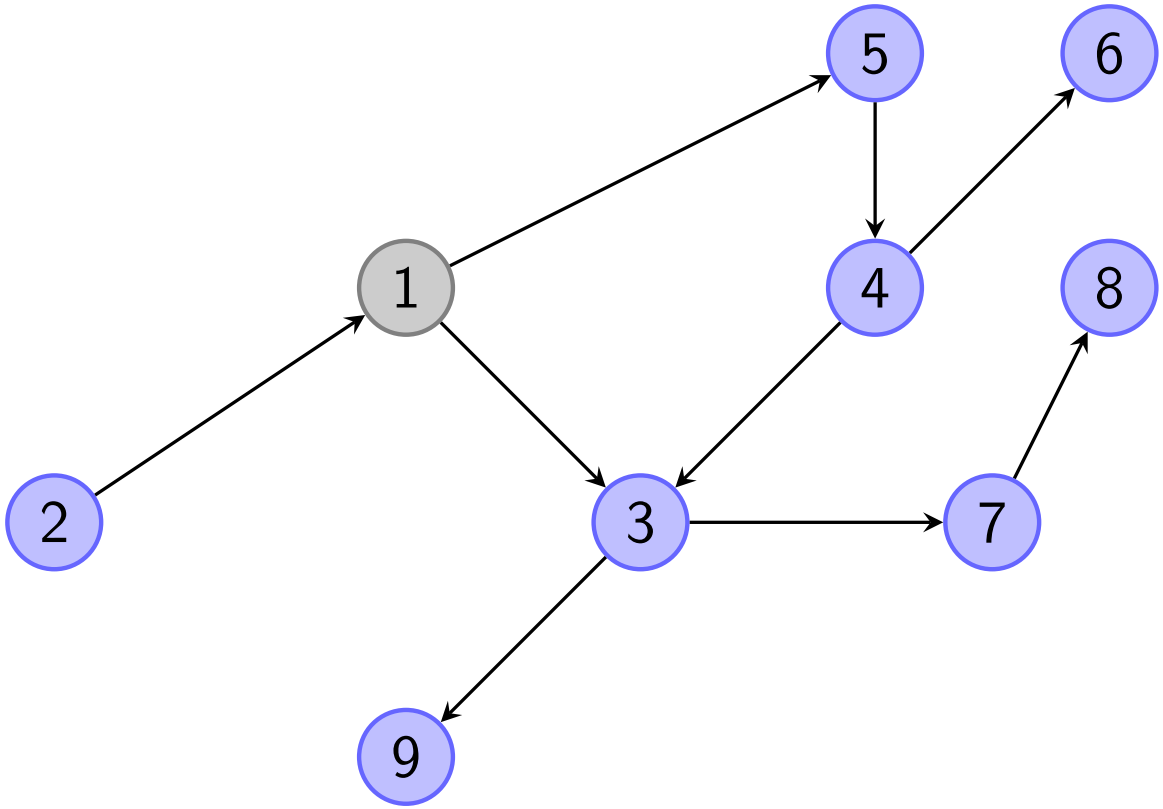
Parcours en largeur



File

$\emptyset$

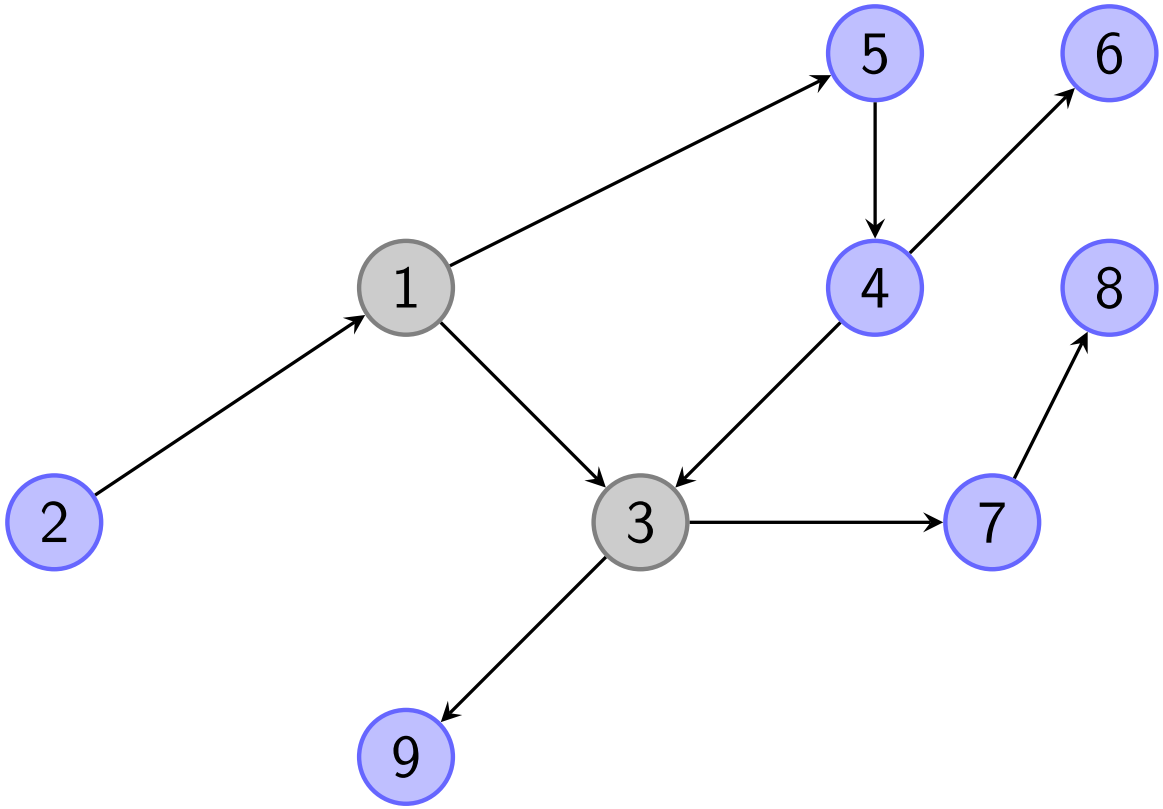
Parcours en largeur



File

1

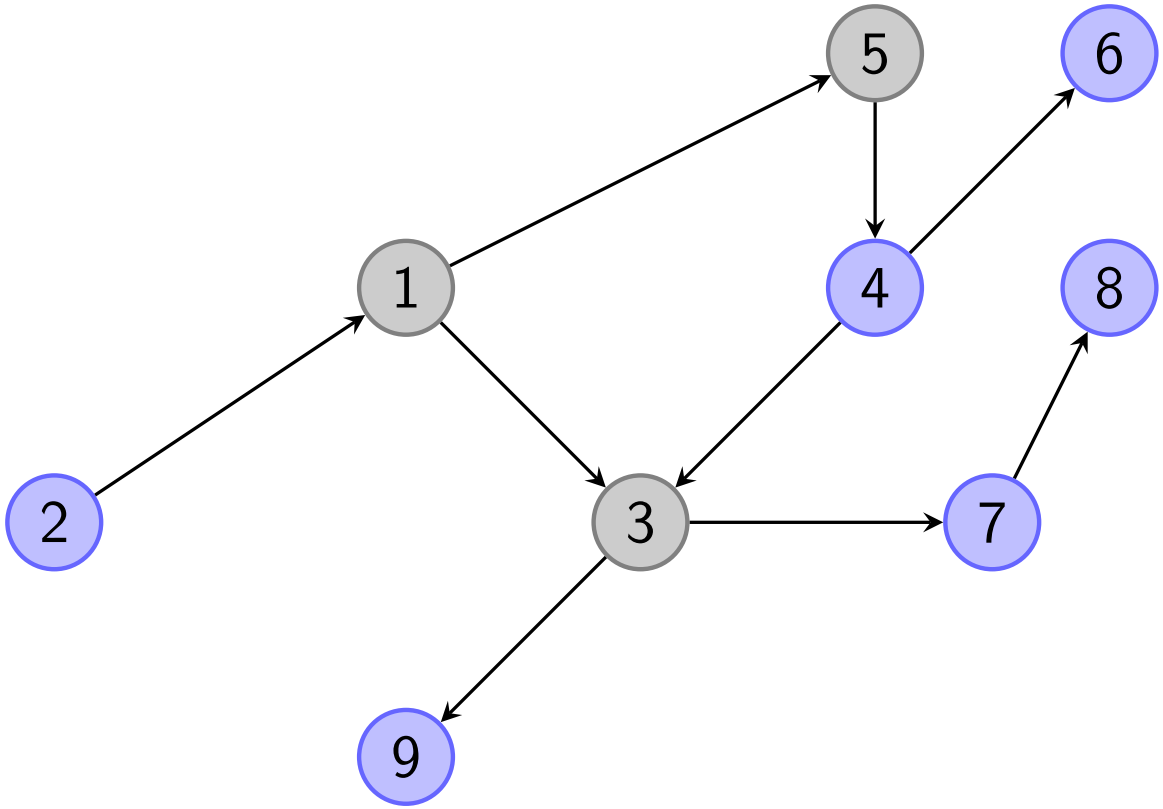
Parcours en largeur



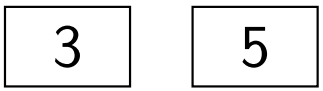
File

3

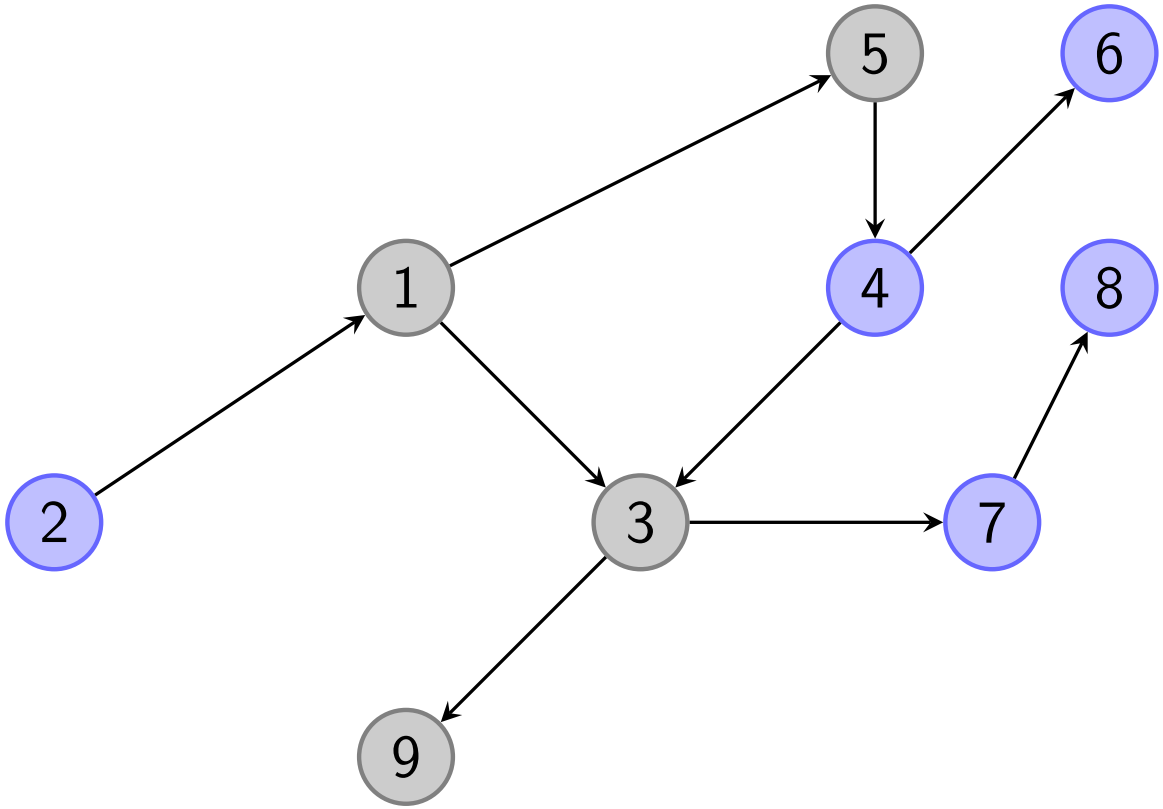
Parcours en largeur



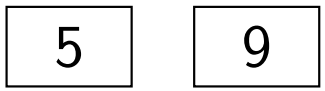
File



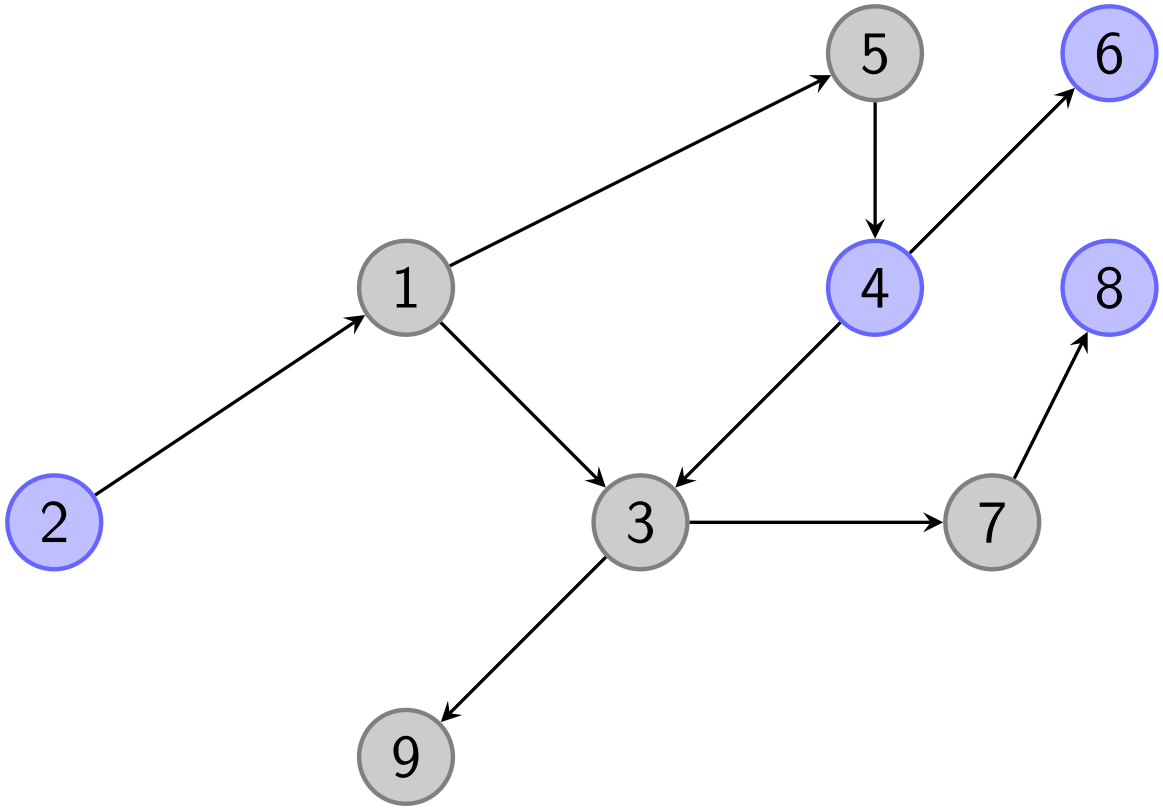
Parcours en largeur



File



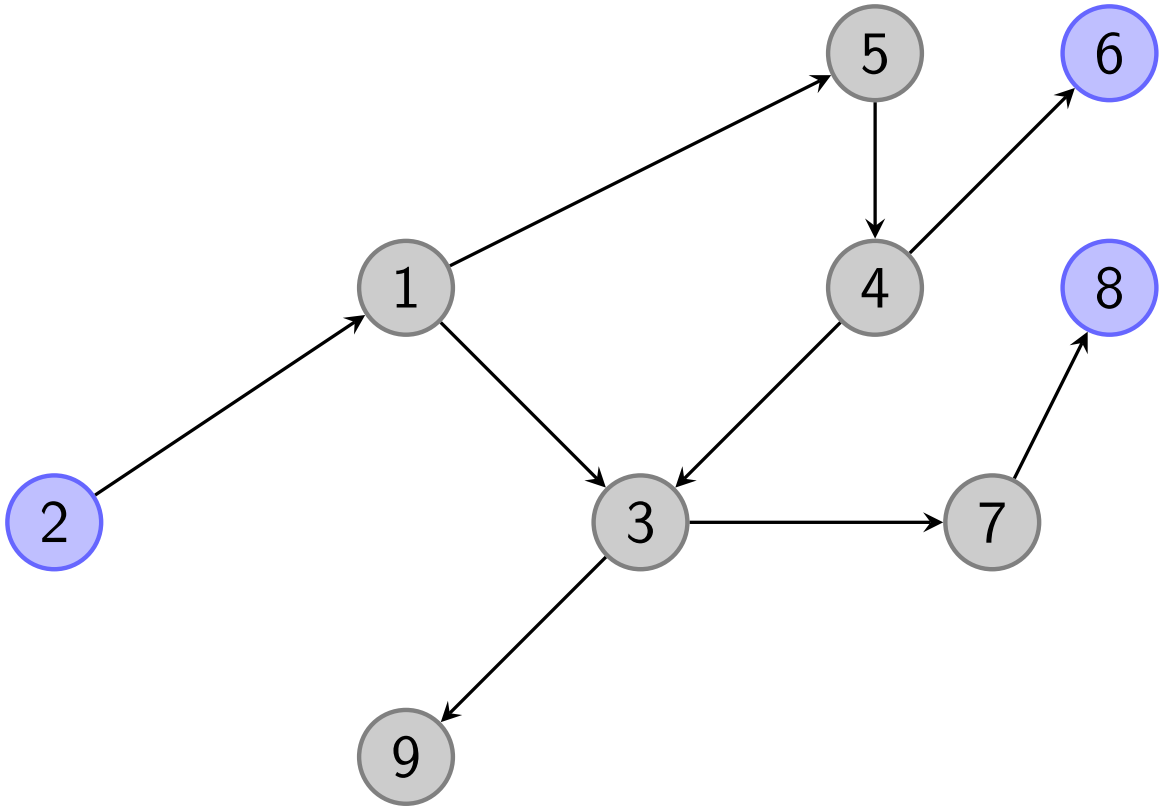
Parcours en largeur



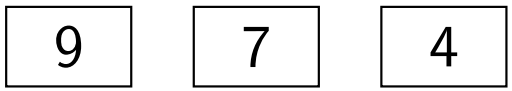
File

5	9	7

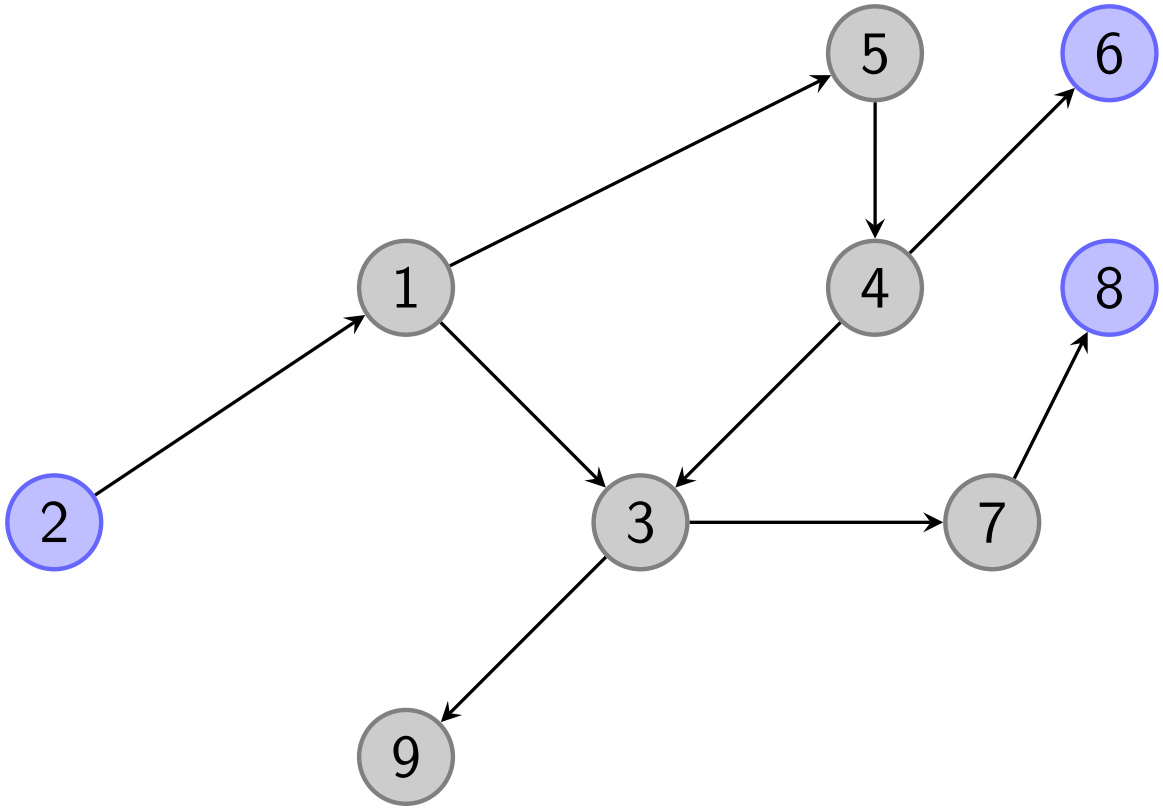
Parcours en largeur



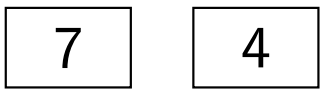
File



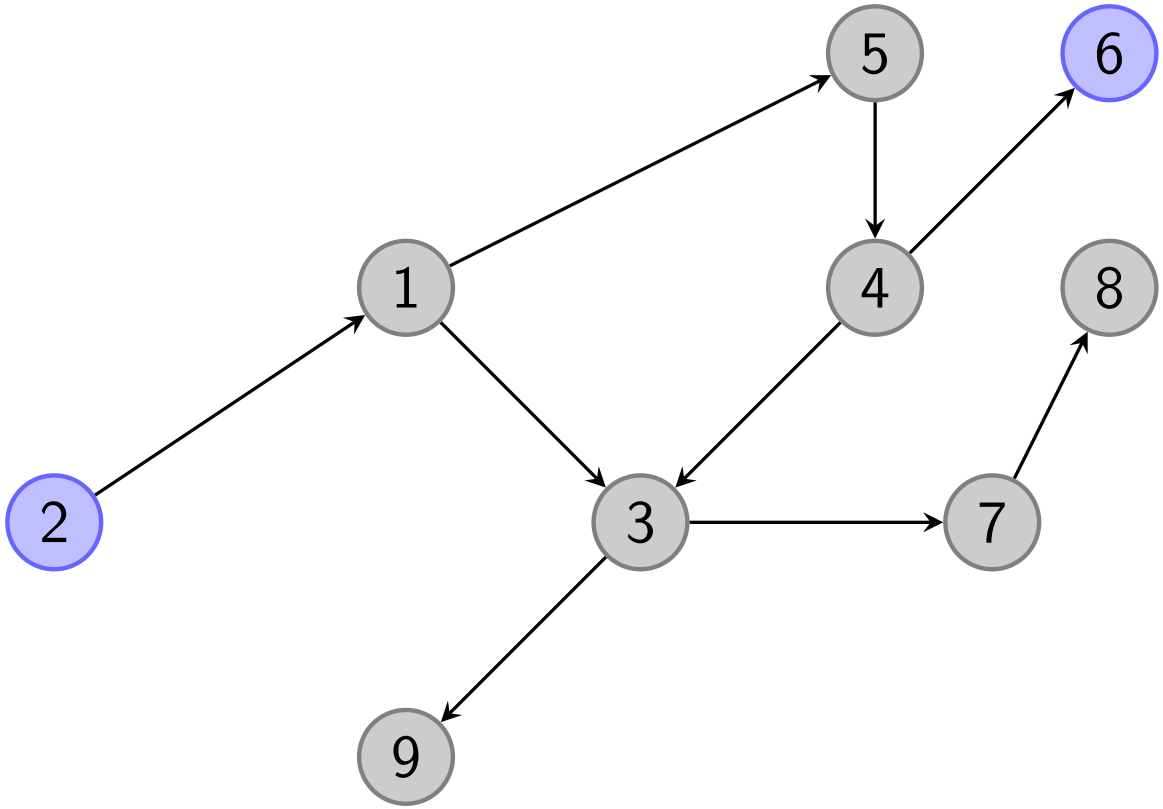
Parcours en largeur



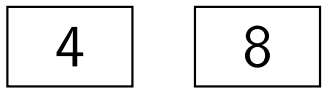
File



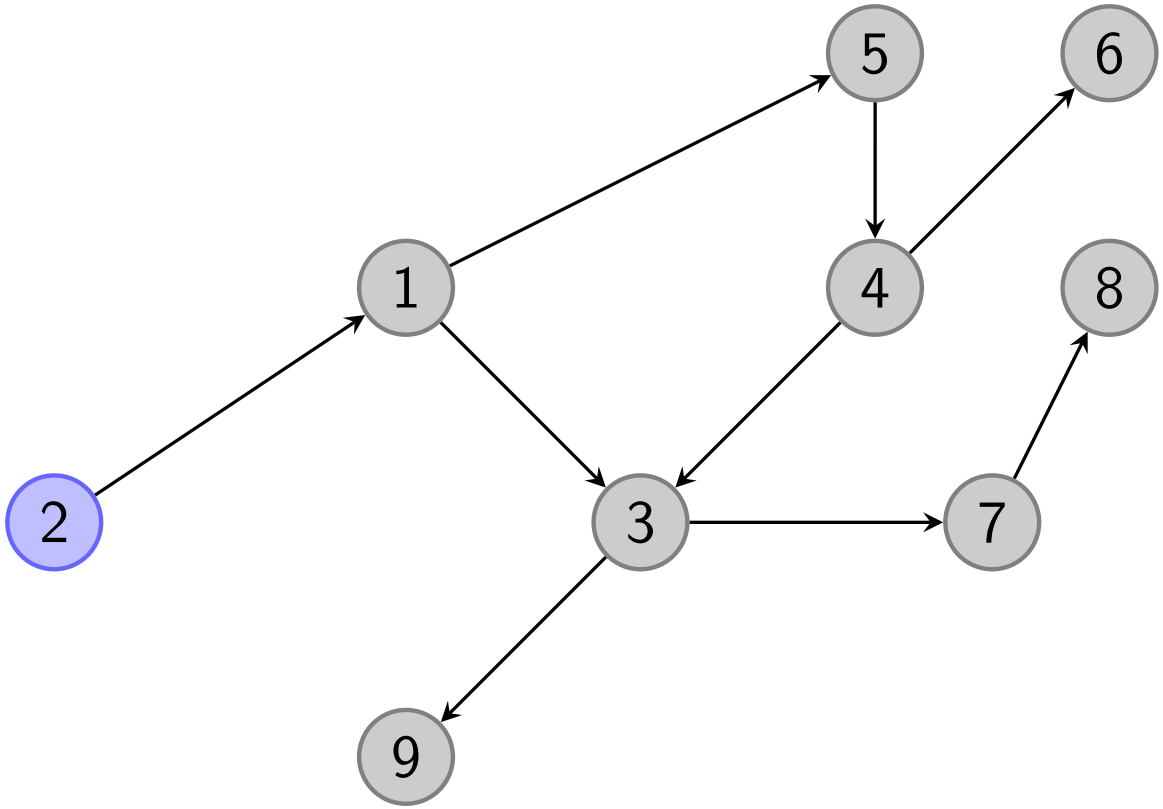
Parcours en largeur



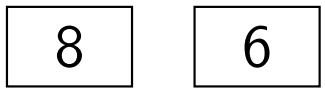
File



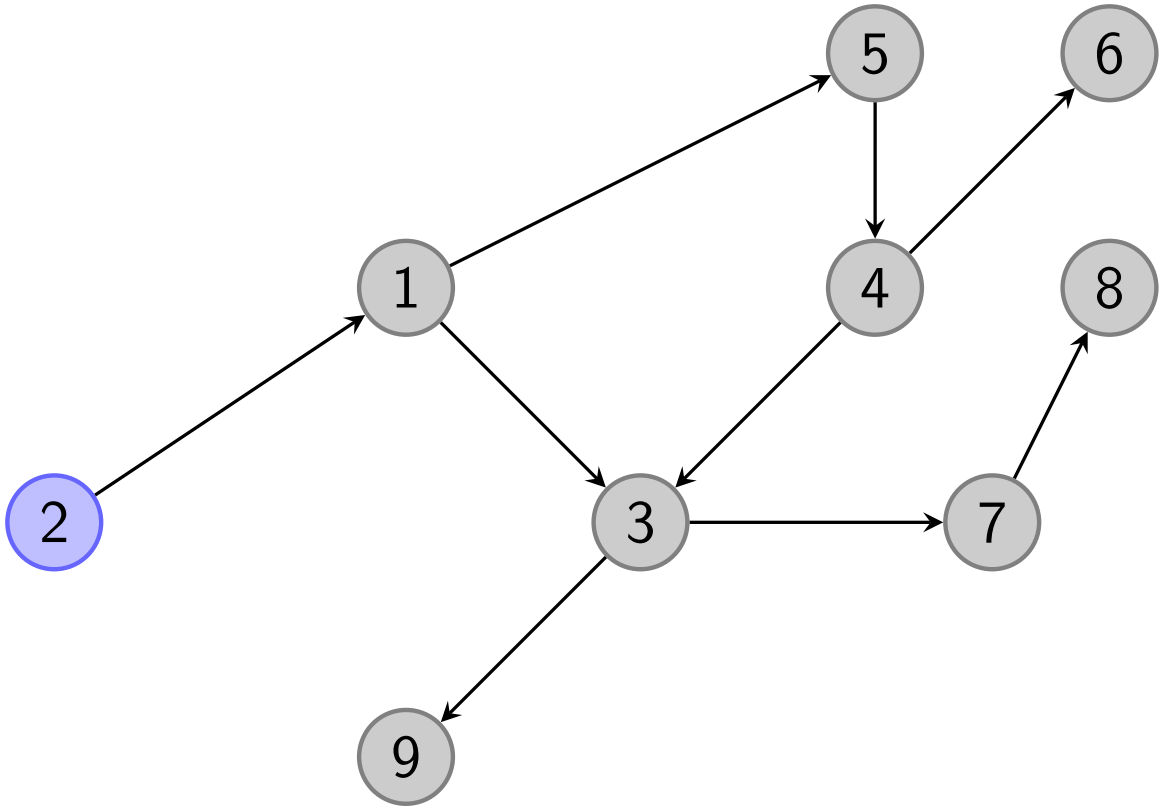
Parcours en largeur



File



Parcours en largeur



File

$\emptyset$

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

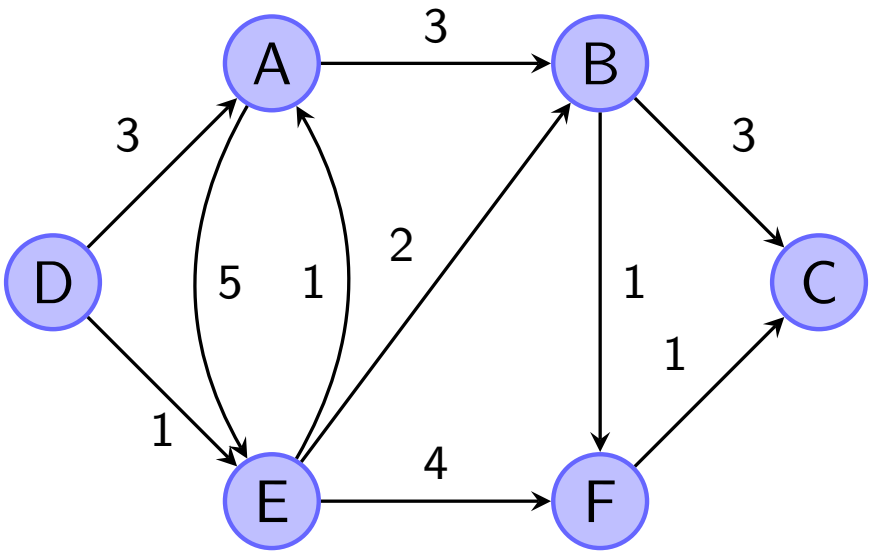
Entrées : Un graphe $G = (V, E, w)$, un sommet de départ s

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

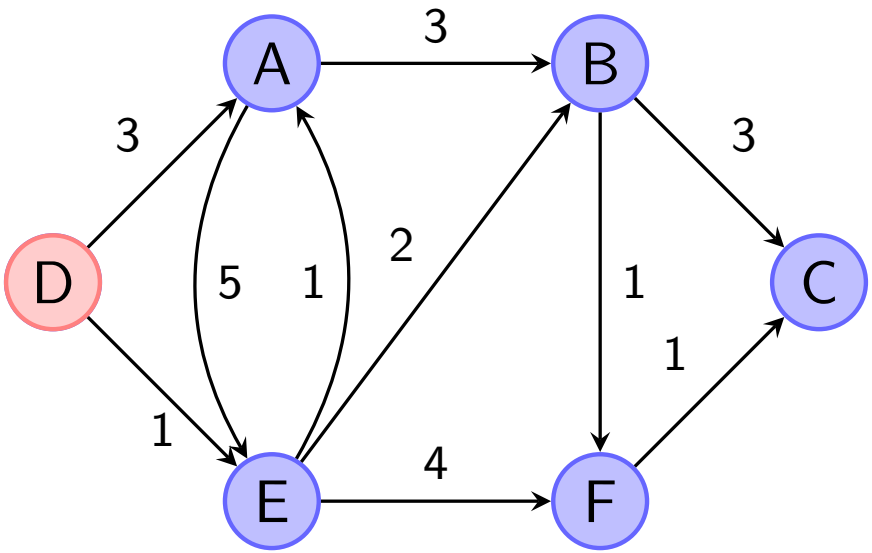


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{D\}$

u =

Algorithme de Dijkstra

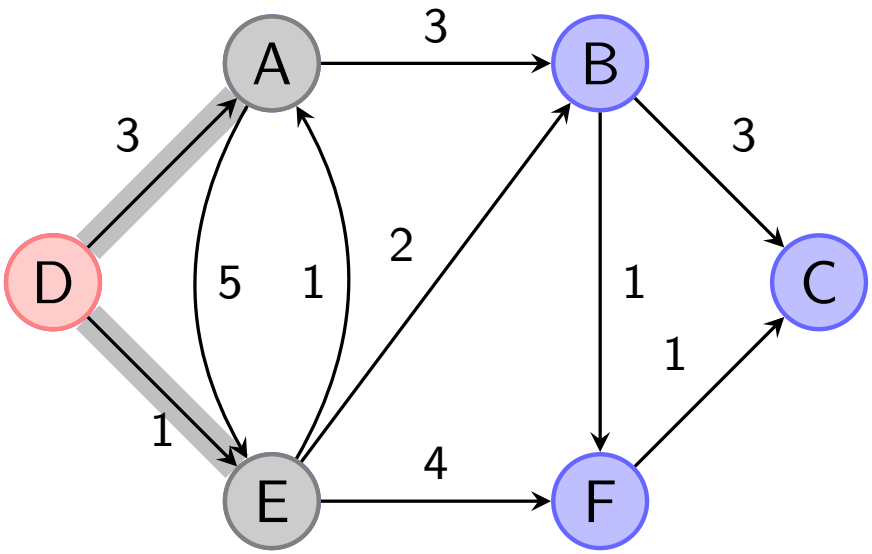


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{\}$

u = D

Algorithme de Dijkstra

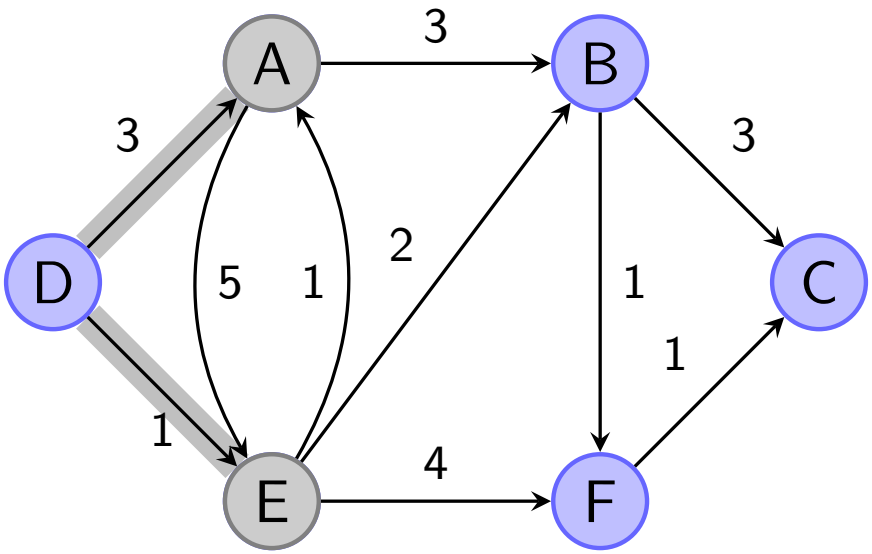


Nœud	Distance	Parent
A		
B		
C		
D	0	•
E		
F		

Frontière = $\{A, E\}$

u = D

Algorithme de Dijkstra

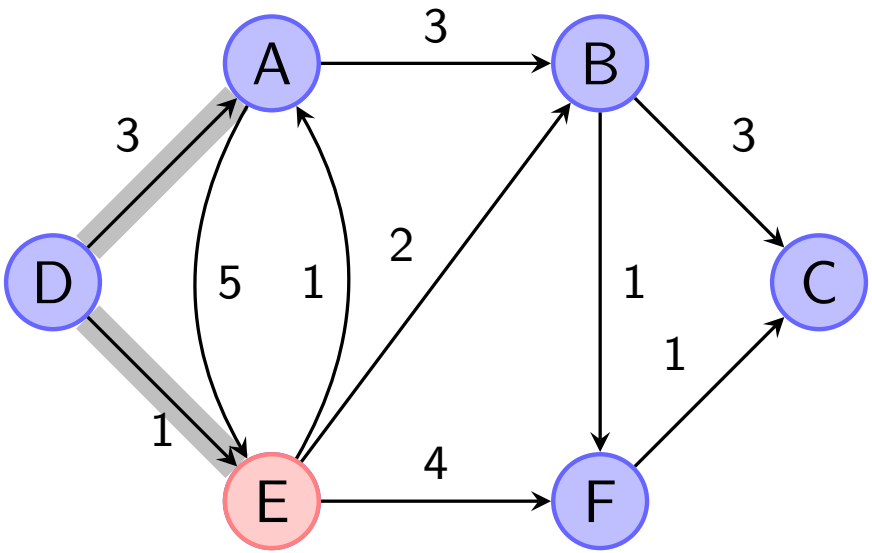


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A, E\}$

u = D

Algorithme de Dijkstra

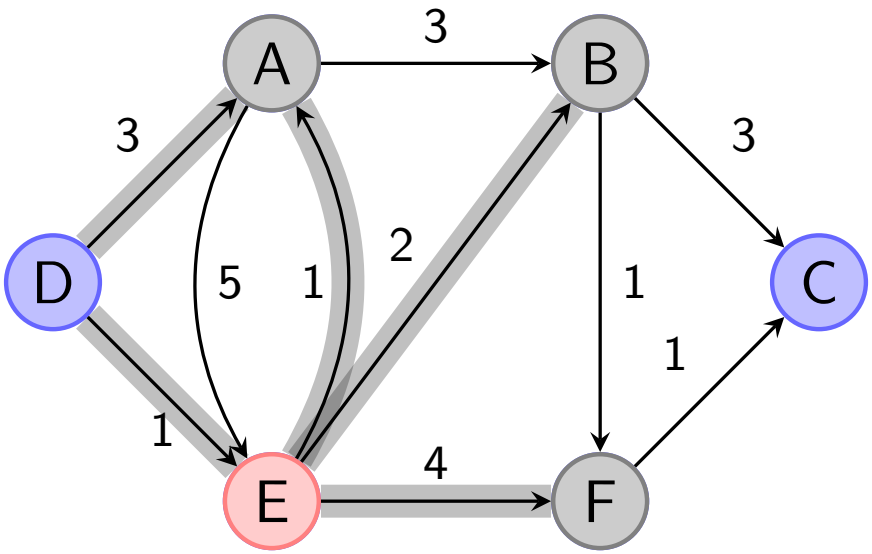


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A\}$

u = E

Algorithme de Dijkstra

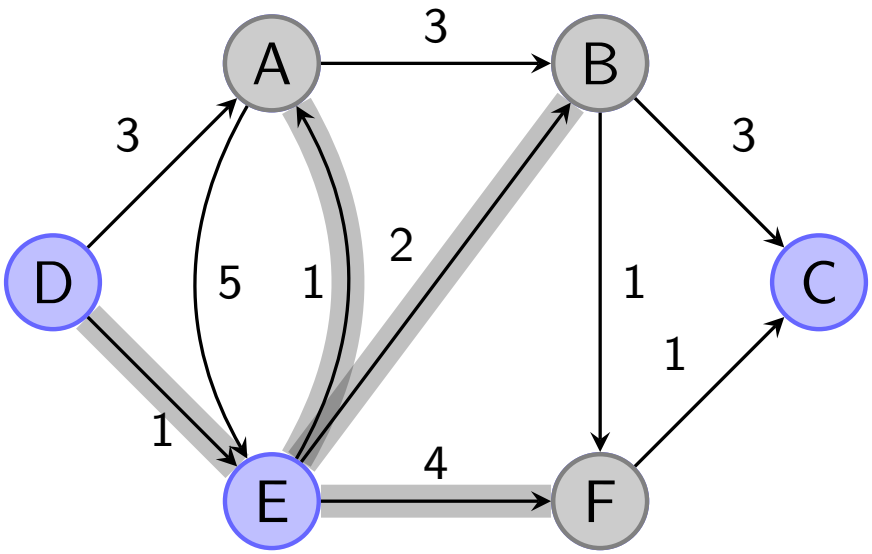


Nœud	Distance	Parent
A	3	D
B		
C		
D	0	•
E	1	D
F		

Frontière = $\{A, B, F\}$

u = E

Algorithme de Dijkstra

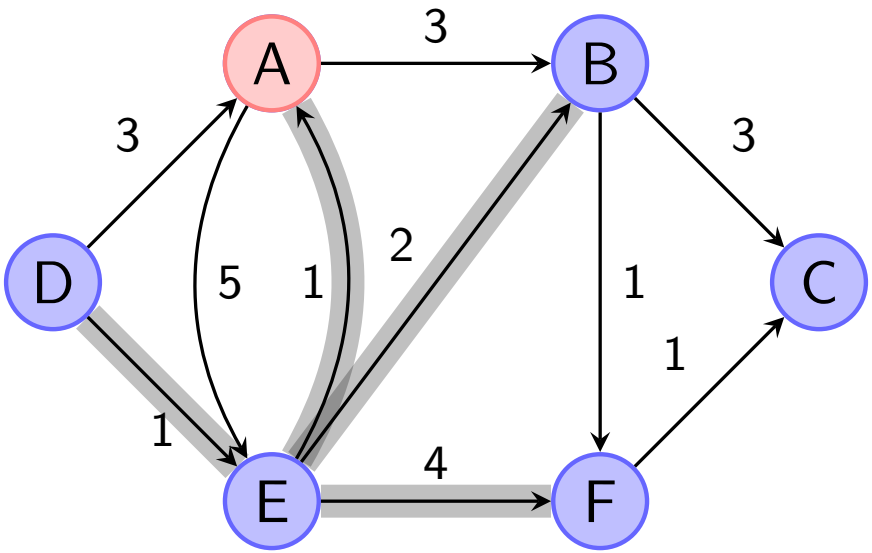


Nœud	Distance	Parent
A	1+1	E
B	1+2	E
C		
D	0	•
E	1	D
F	1+4	E

Frontière = $\{A, B, F\}$

u = E

Algorithme de Dijkstra

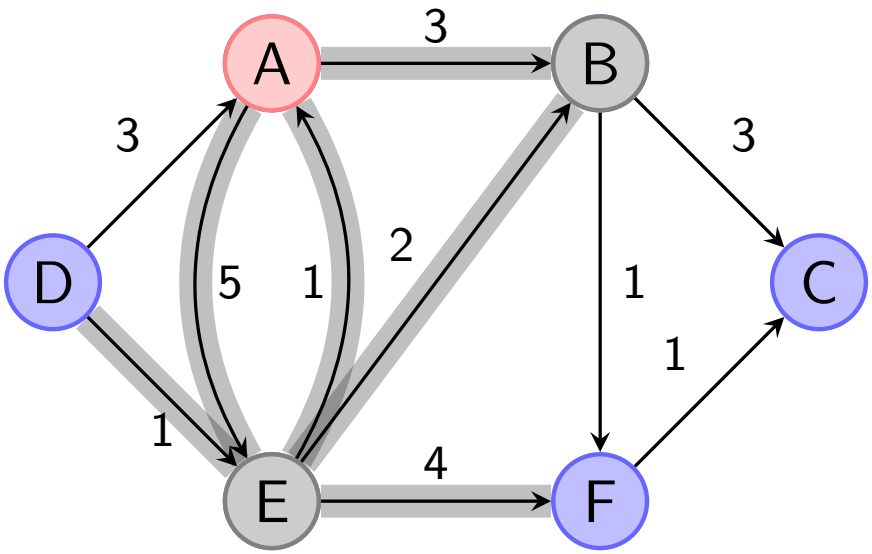


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

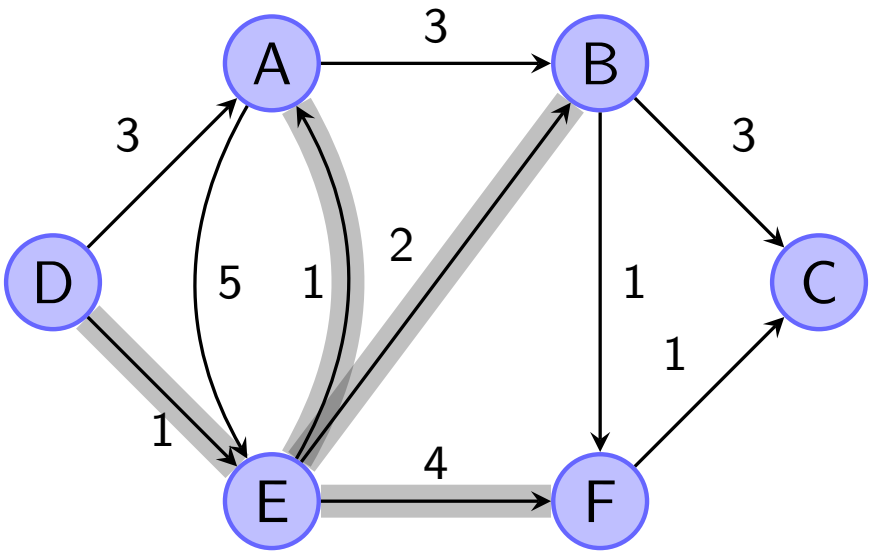


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

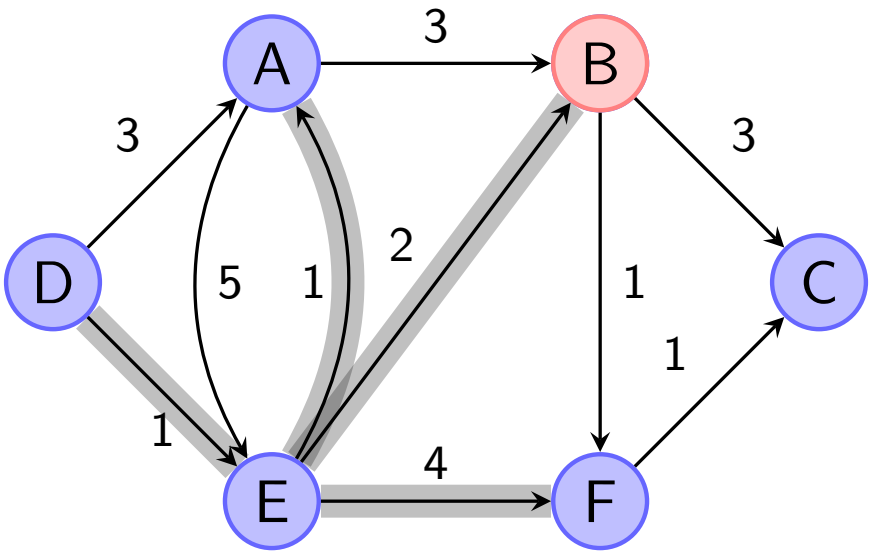


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{B, F\}$

u = A

Algorithme de Dijkstra

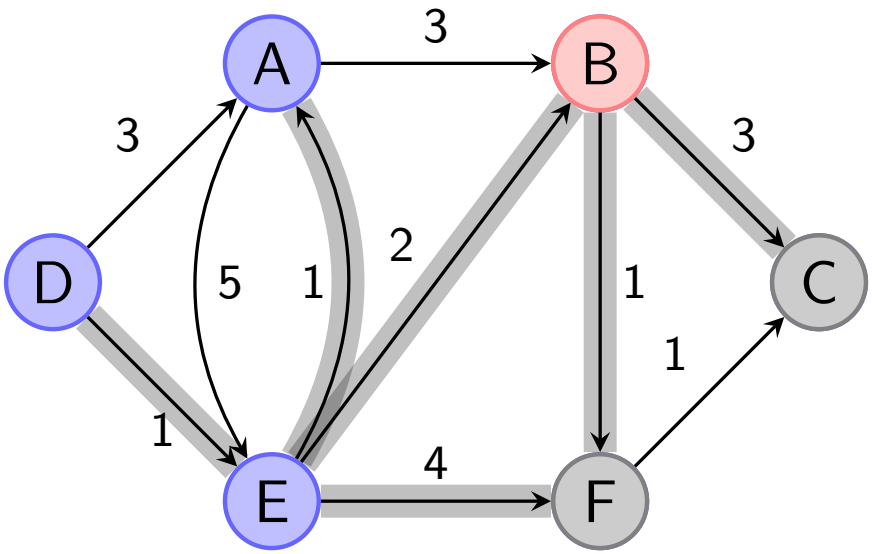


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{F\}$

u = B

Algorithme de Dijkstra

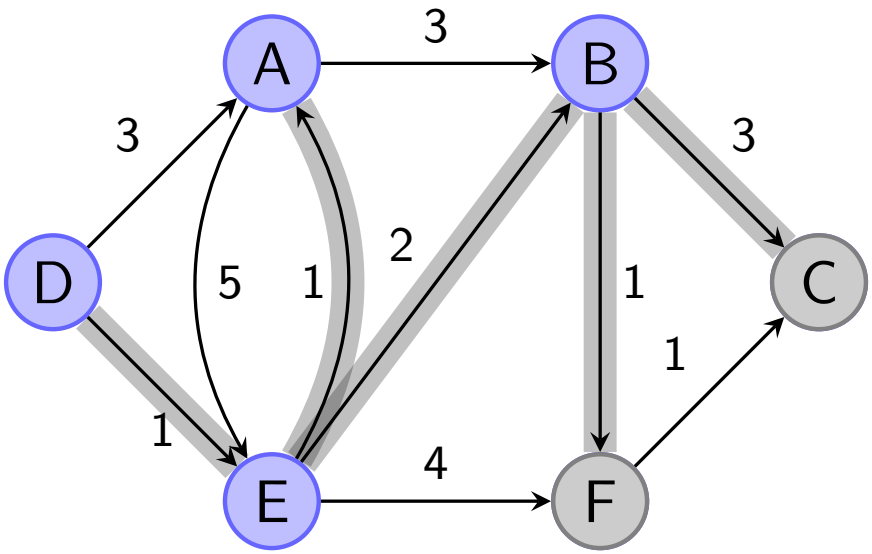


Nœud	Distance	Parent
A	2	E
B	3	E
C		
D	0	•
E	1	D
F	5	E

Frontière = $\{C, F\}$

u = B

Algorithme de Dijkstra

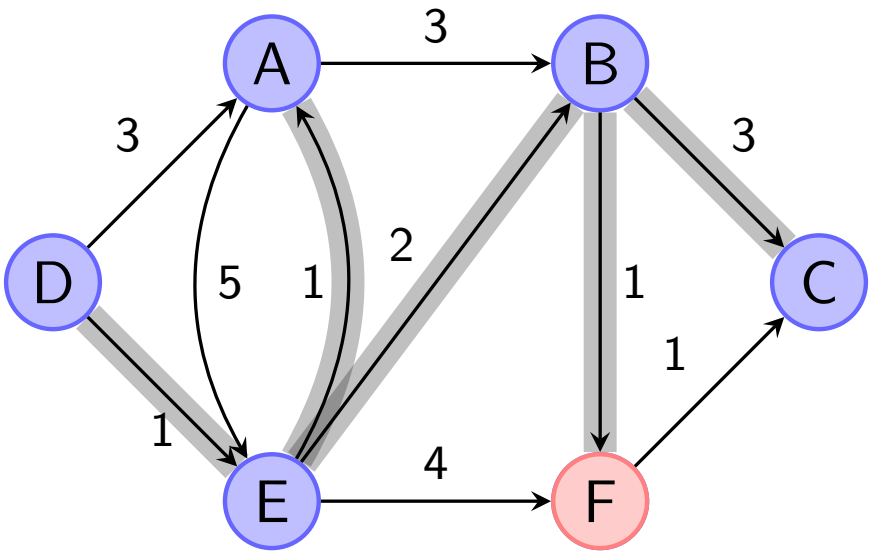


Nœud	Distance	Parent
A	2	E
B	3	E
C	3+3	B
D	0	•
E	1	D
F	3+1	B

Frontière = $\{C, F\}$

u = B

Algorithme de Dijkstra

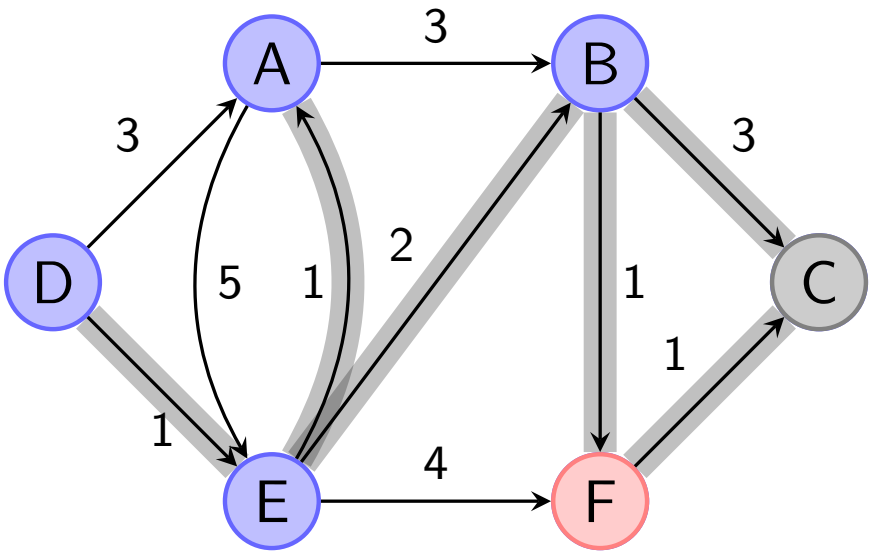


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

u = F

Algorithme de Dijkstra

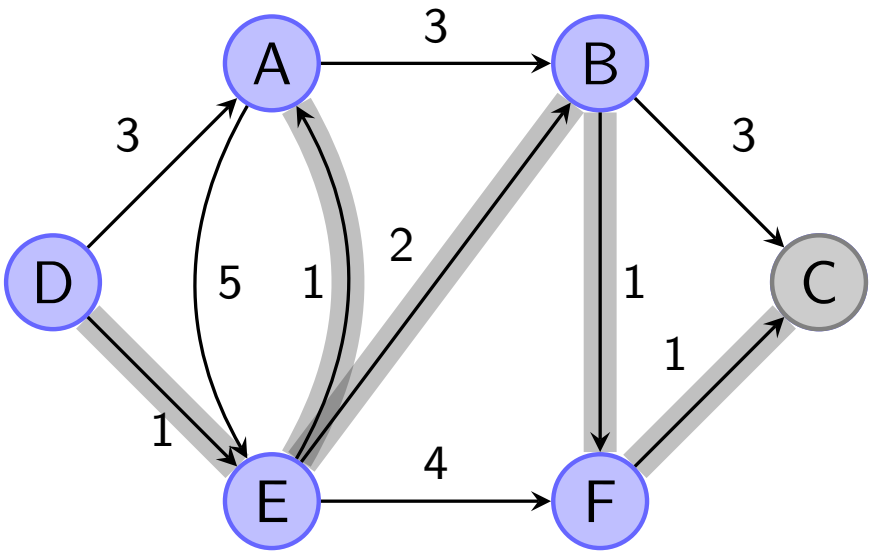


Nœud	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontière = $\{C\}$

u = F

Algorithme de Dijkstra

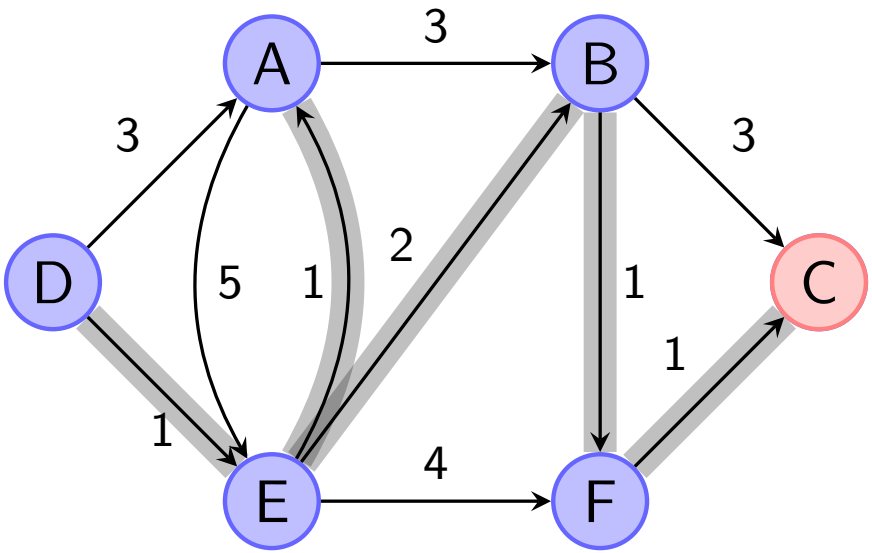


Nœud	Distance	Parent
A	2	E
B	3	E
C	4+1	F
D	0	•
E	1	D
F	4	B

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 (*Minimum Spanning Tree*) est un sous-graphe acyclique connexe 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 est un algorithme glouton qui 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.

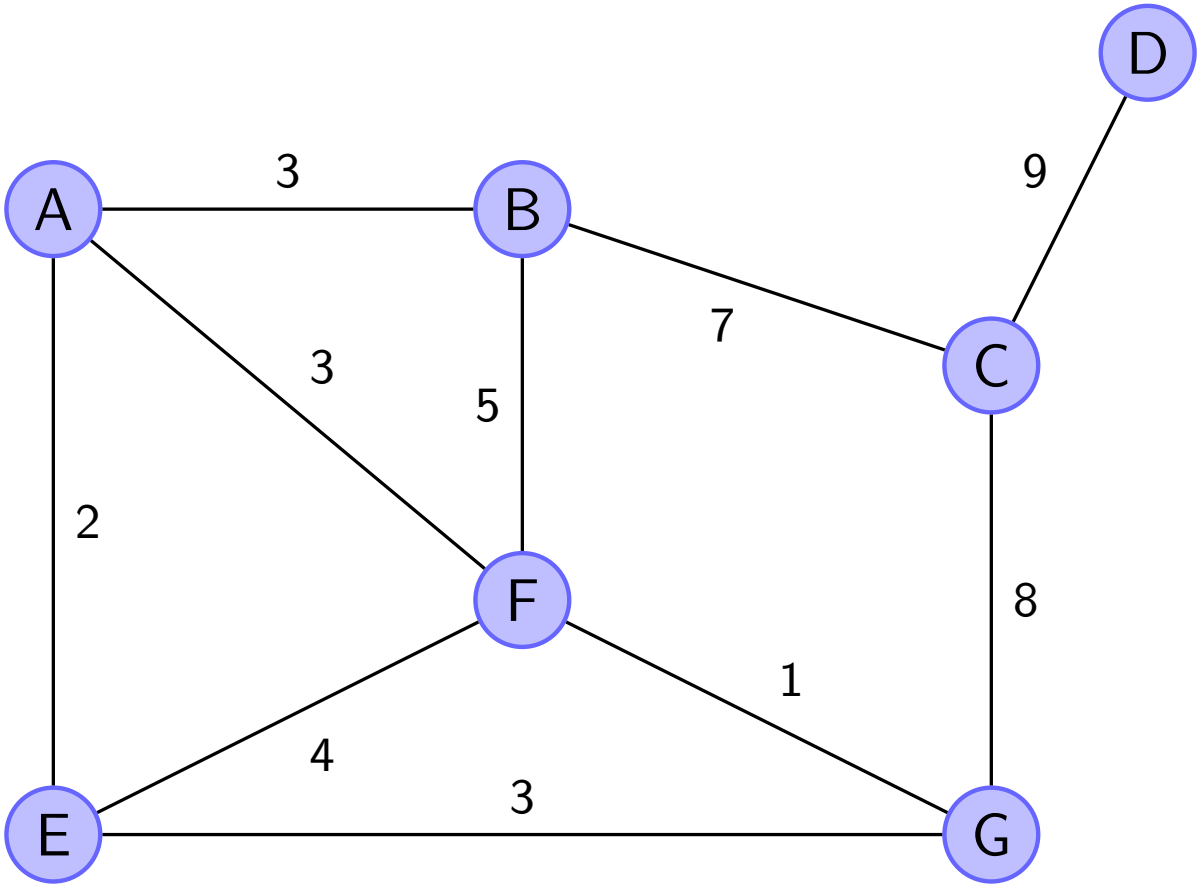
Algorithme de Kruskal

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

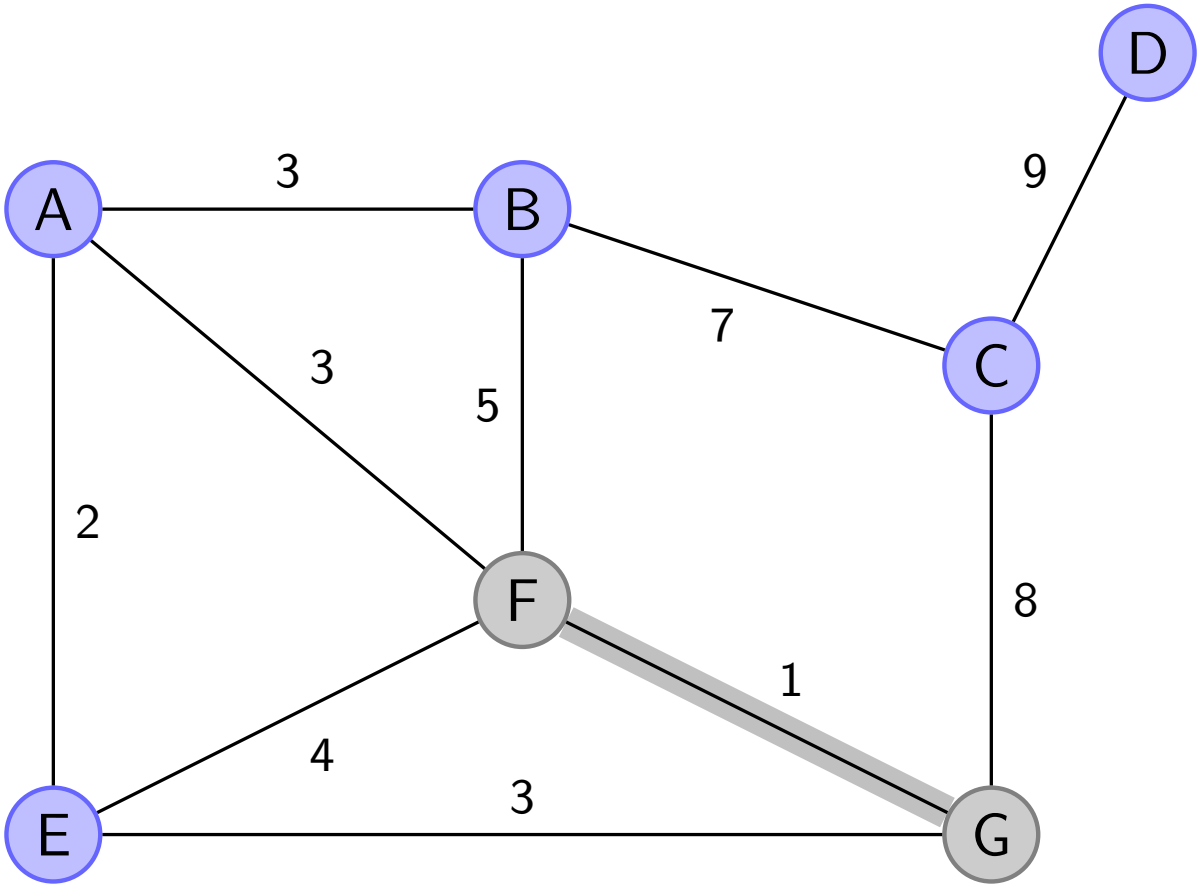
Algorithme :

$T \leftarrow \emptyset$
$P \leftarrow$ **Init**$(|V|)$ ▷ 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

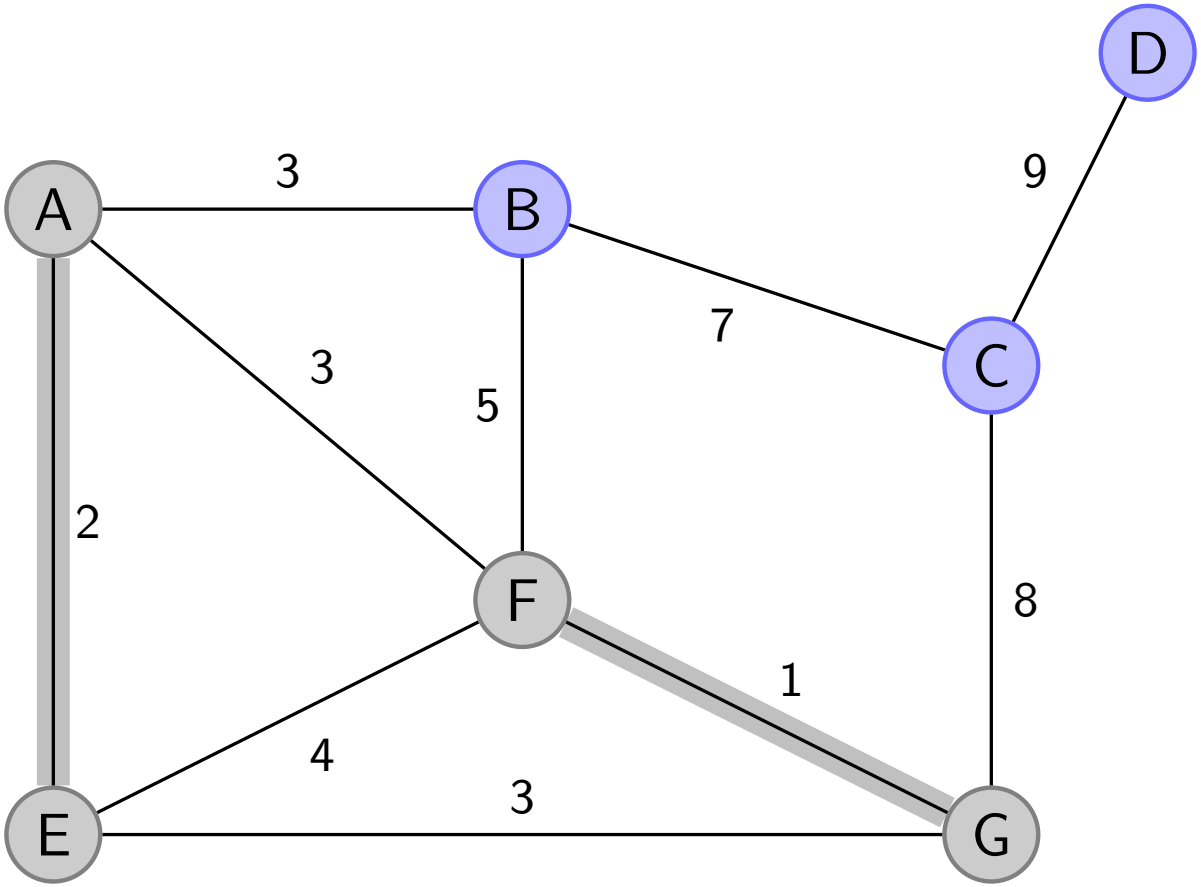
Algorithme de Kruskal



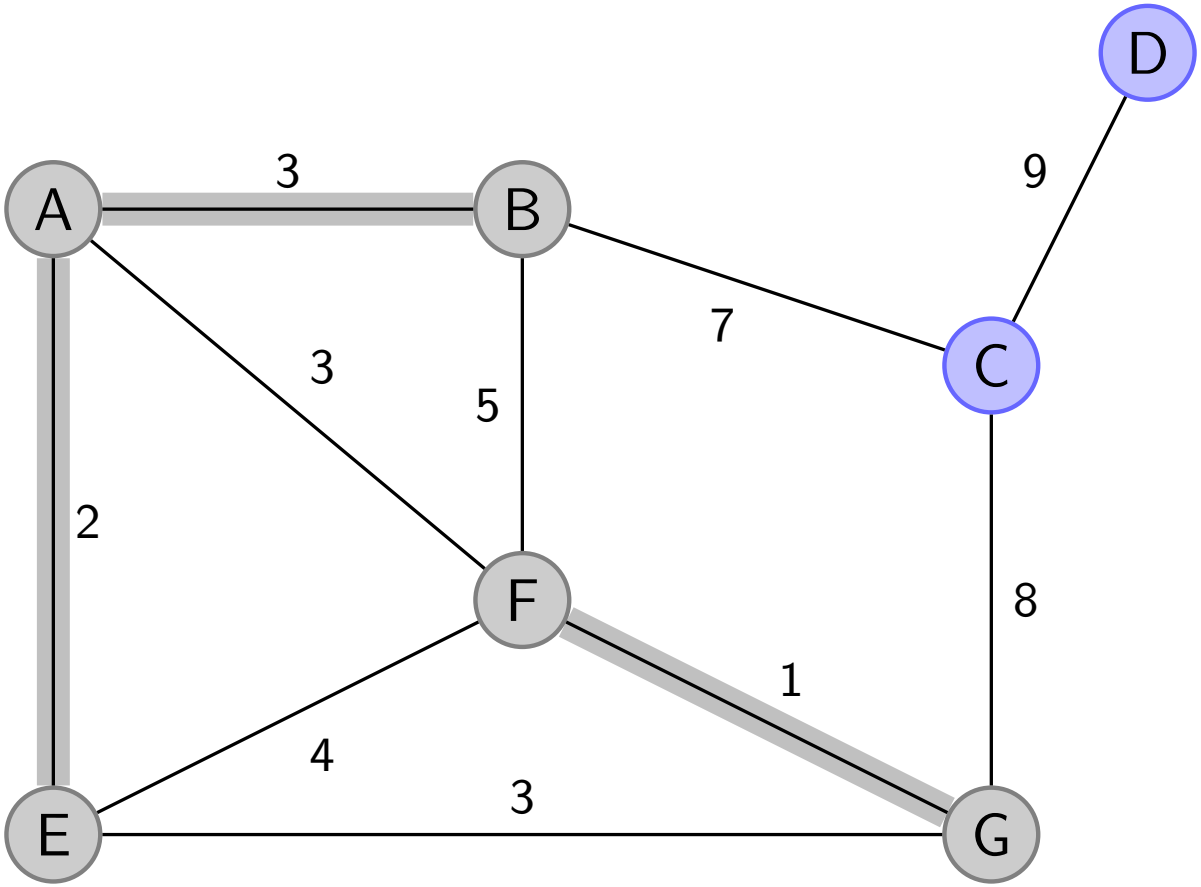
Algorithme de Kruskal



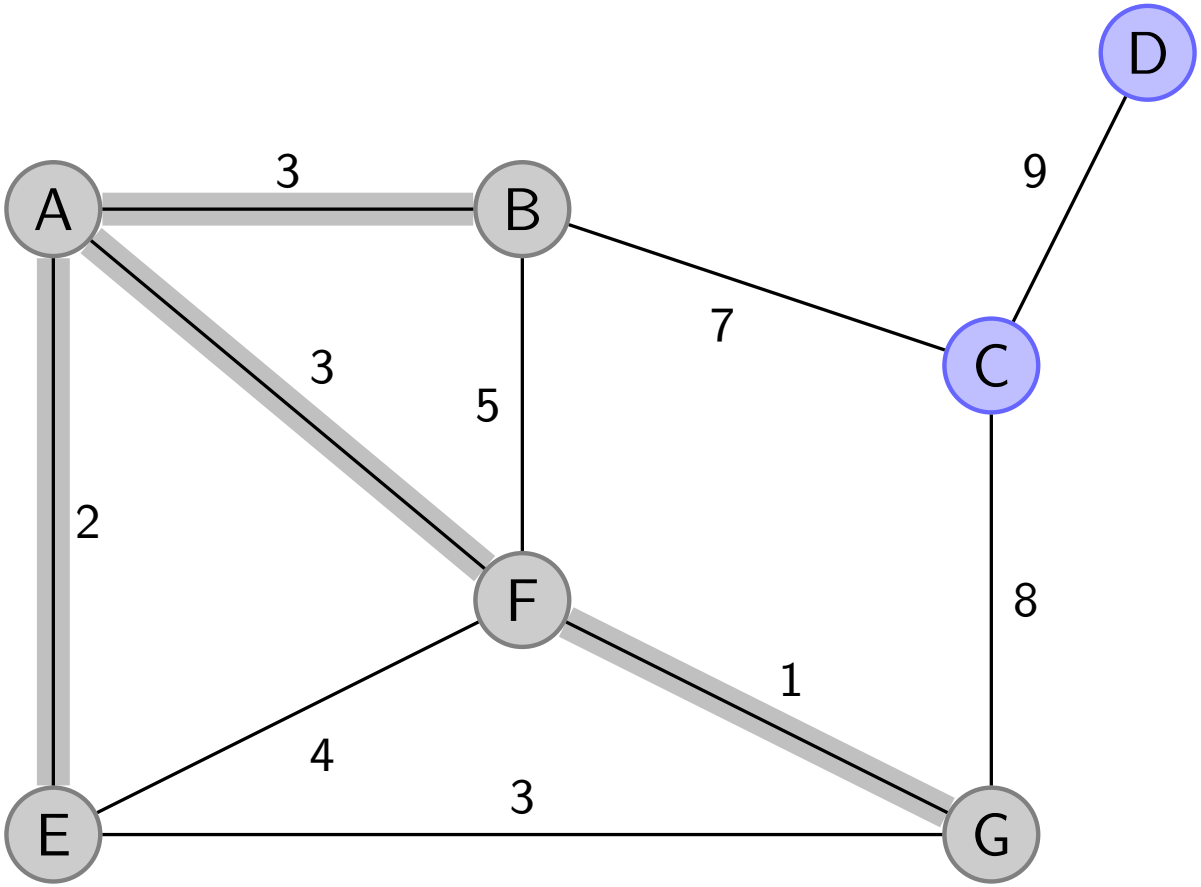
Algorithme de Kruskal



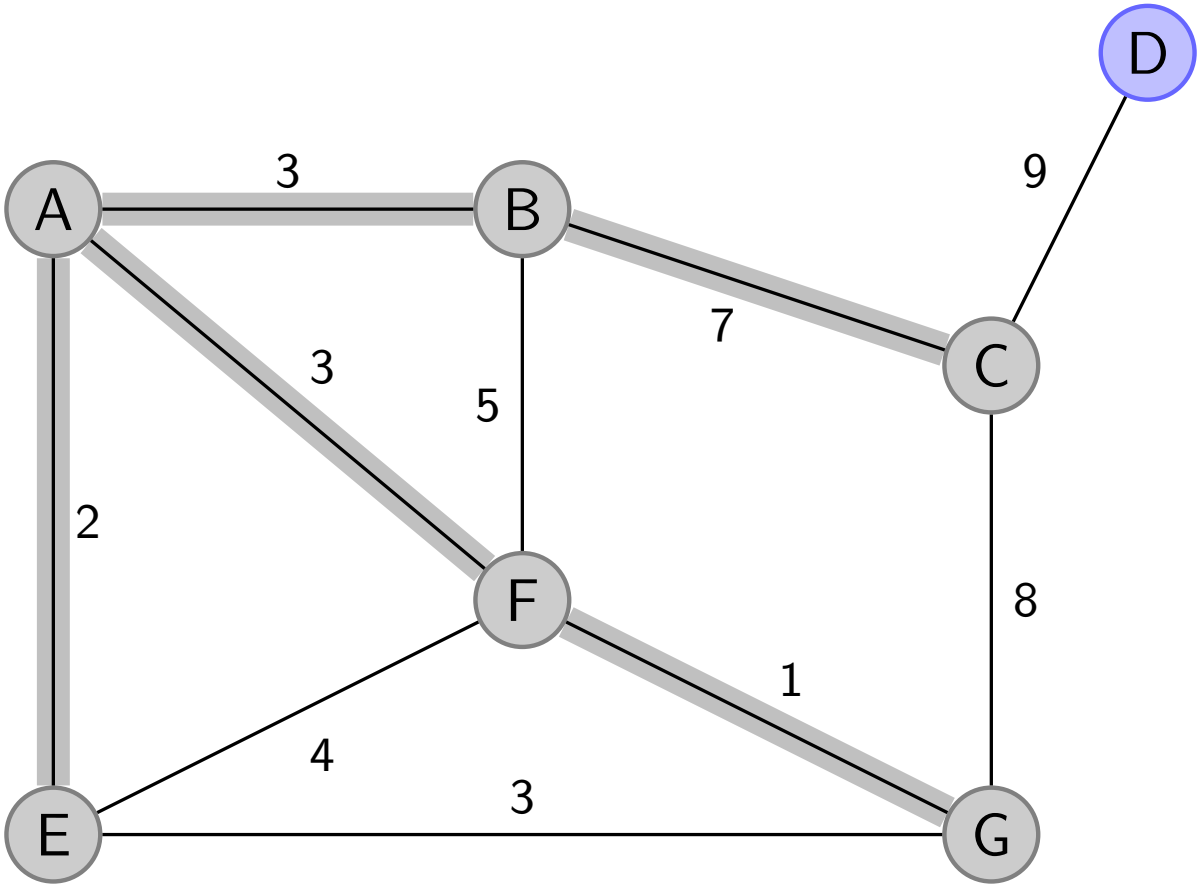
Algorithme de Kruskal



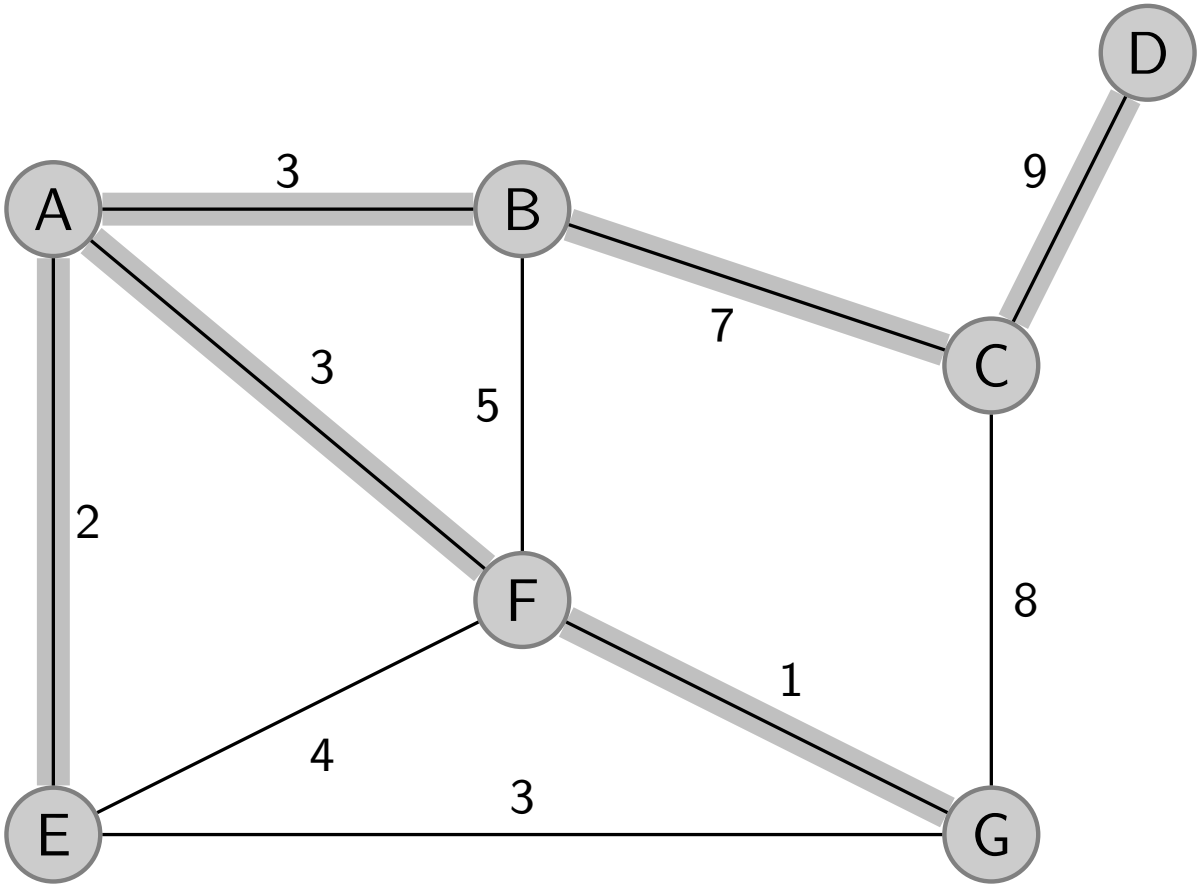
Algorithme de Kruskal



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

Comment s'entraîner :

- Refaire les TDs de pré-requis
- Faire tourner les algorithmes Dijkstra/Kruskal sur des exemples

Types de problèmes

Problèmes de décision

Problèmes auxquels on peut répondre par oui ou non.

- **Problème du plus court chemin** : existe-t-il un chemin de coût $\leq k$ entre deux nœuds s et t dans un graphe pondéré ?
- **Problème du cycle hamiltonien** : existe-t-il un cycle passant par tous les nœuds d'un graphe exactement une fois ?

Problèmes d'optimisation

Problèmes dans lesquels on cherche la solution de coût minimal (ou maximal).

- **Problème du plus court chemin** : quel est le chemin le plus court entre les nœuds s et t dans un graphe pondéré ?
- **Problème du voyageur de commerce** : trouver un cycle hamiltonien de poids minimum.

Formaliser un problème

Pour formaliser un problème, on donne ses entrées (instances), et la question à laquelle on veut répondre.

Exemple

Formaliser le problème du plus court chemin entre deux nœuds s et t dans un graphe pondéré sous forme de problème de décision.

Entrées :

- Un graphe $G = (V, E, w)$ pondéré
- Des nœuds $s, t \in V$
- Un nombre $k \in \mathbb{N}$

Question : existe-t-il un chemin $C \subseteq E$ de s à t tel que :

$$\sum_{c \in C} w(c) \leq k$$

Types de problèmes

Attention !

Dans tout ce qui suit, on s'intéresse uniquement à des **problèmes de décision**.

Dans le cas des **problèmes de décision** uniquement, on peut définir les classes de complexité **P** et **NP**.

Classe P

Un problème est dans la classe P s'il peut être résolu par un algorithme en complexité polynomiale.

Classe NP

Un problème est dans la classe NP si ses solutions peuvent être vérifiées par un algorithme en temps polynomial.

Pour montrer qu'un problème est dans NP, il suffit de donner un algorithme de vérification et de montrer qu'il est polynomial.

On a naturellement $P \subseteq NP$.

Instance et instance positive

Soit un problème $\mathcal{P}$. On note $\mathcal{D}$ les instances (ou entrées) possibles pour $\mathcal{P}$ (e.g l'ensemble des graphes).

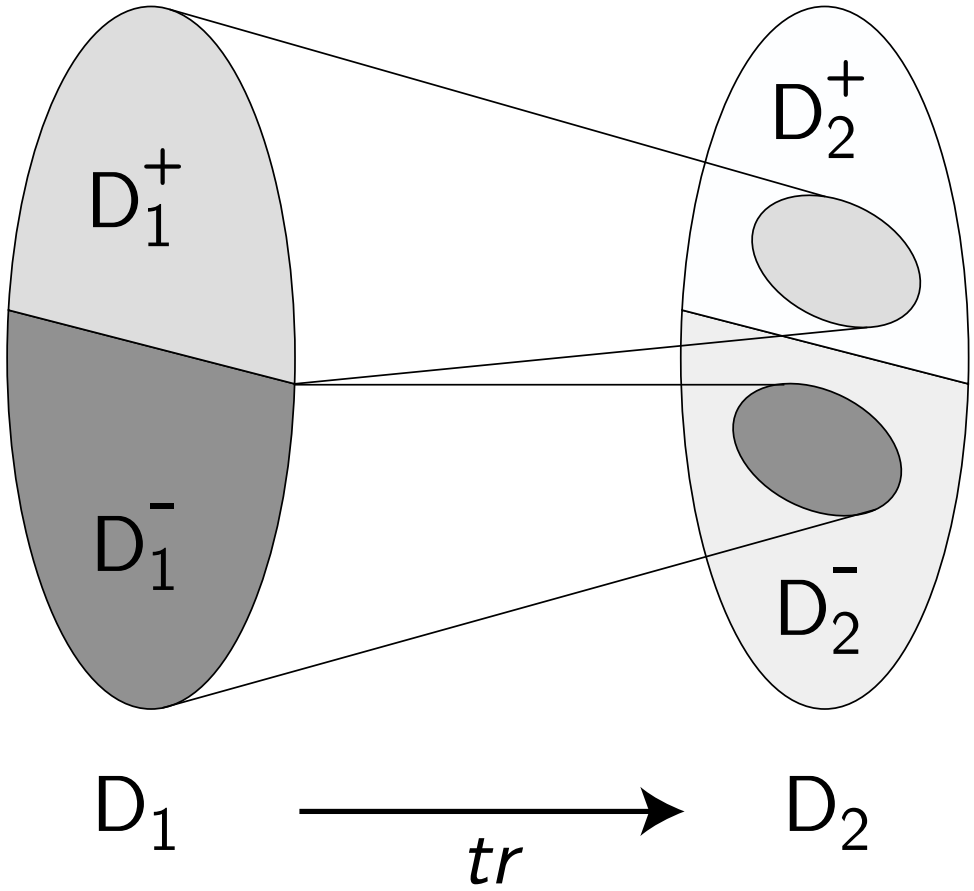
On a $\mathcal{D} = \mathcal{D}^+ \sqcup \mathcal{D}^-$ (union disjointe) où :

- $\mathcal{D}^+$ est l'ensemble des entrées pour lesquelles la réponse au problème est oui (instances positives) ;
- $\mathcal{D}^-$ est l'ensemble des entrées pour lesquelles la réponse au problème est non (instances négatives).

Réduction

On dit qu'un problème $\mathcal{P}_1$ se réduit à un autre problème $\mathcal{P}_2$ s'il existe une fonction tr qui transforme une entrée de $\mathcal{P}_1$ en entrée de $\mathcal{P}_2$ en temps polynomial, telle que $e \in \mathcal{D}_1^+$ ssi $tr(e) \in \mathcal{D}_2^+$.

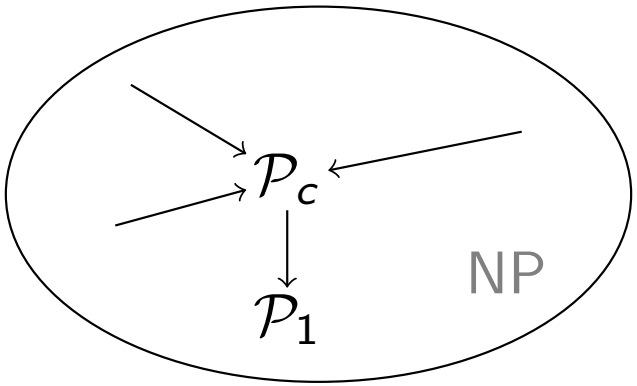
Réduction de problèmes



Problème NP-difficile

Un problème $\mathcal{P}_1$ est dit NP-difficile si tout problème de la classe NP peut s'y réduire.

En pratique, pour montrer que $\mathcal{P}_1$ est NP-difficile, on trouve une réduction d'un problème NP-difficile $\mathcal{P}_c$ à $\mathcal{P}_1$.



Problème NP-complet

Un problème est dit NP-complet s'il est **dans la classe NP** et qu'il est **NP-difficile**.

Exemple de réduction

On se propose de réduire le problème Chemin Hamiltonien (NP-difficile) vers Plus Long Chemin pour montrer que Plus Long Chemin est NP-difficile.

Chemin Hamiltonien

Entrées : un graphe non orienté connexe $G = (V, E)$

Question : existe-t-il un chemin simple (sans cycle) qui visite tous les sommets de G ?

Plus long chemin

Entrées : un graphe non orienté pondéré connexe $G = (V, E, w)$ et un entier $k \in \mathbb{N}$

Question : existe-t-il un chemin simple C dans G tel que la somme des poids des arêtes de C soit au moins k ?

Exemple de réduction

On propose $tr(\langle G = (V, E)\rangle) = \langle G' = (V, E, w), k\rangle$ où $w(e) = 1$ $\forall e \in E$ et $k = |V| - 1$.

Il faut montrer que $\mathcal{I}_{Ham} \in \mathcal{D}^+_{Ham} \iff tr(\mathcal{I}_{Ham}) \in \mathcal{D}^+_{PLC}$

$\boxed{\Rightarrow}$ Soit $\mathcal{I}_{Ham} = \langle G = (V, E)\rangle$ une instance positive du problème du chemin hamiltonien. Vu $\mathcal{I}_{Ham} \in \mathcal{D}^+_{Ham}$, il existe un chemin hamiltonien dans G, qui par définition contient $|V| - 1$ arêtes. Dans $\mathcal{I}_{PLC} = tr(\mathcal{I}_{Ham})$, le chemin est de poids $|V| - 1 = k$. Donc $\mathcal{I}_{PLC} \in \mathcal{D}^+_{PLC}$.

$\boxed{\Leftarrow}$ Réciproquement, si $\mathcal{I}_{PLC}$ est une instance positive du problème plus long chemin, il existe un chemin simple de poids au moins $k = |V| - 1$. Comme chaque arête est de poids 1, il doit visiter au moins $k + 1 = |V|$ sommets. Le chemin étant simple (pas de répétition de sommets), on a un chemin hamiltonien. Donc $\mathcal{I}_{Ham} \in \mathcal{D}^+_{Ham}$.

On a réduit Ham (NP-difficile) à PLC donc PLC est NP-difficile.

Méthode pour montrer la NP-complétude

Pour montrer qu'un problème $\mathcal{P}_1$ est **NP-complet** :

1. On montre que le problème est **dans NP** en exhibant un algorithme de vérification de complexité polynomiale ;
2. On montre que le problème est **NP-difficile** en réduisant un autre problème $\mathcal{P}_c$ NP-difficile à lui ;
   - Trouver une fonction tr pour transformer une entrée de $\mathcal{P}_c$ en entrée de $\mathcal{P}$
   - Vérifier que tr est polynomiale
   - Prouver l'équivalence $e \in \mathcal{D}_c^+ \iff tr(e) \in \mathcal{D}_1^+$
3. On conclut en disant que NP & NP-difficile $\iff$ NP-complet

Problèmes NP-complets classiques

SAT

Entrée : une formule logique sous FNC e.g $(x_1 \vee x2) \wedge (\neg x_1)$

Question : la formule est-elle satisfiable ?

Stable

Entrées : un graphe non-orienté connexe $G = (V, E)$; $k \in \mathbb{N}$

Question : existe-t-il un stable S de G, càd un ensemble de sommets qui ne sont pas reliés entre eux par une arête, tel que $|S| \geq k$?

Vertex-Cover

Entrées : un graphe non-orienté connexe $G = (V, E)$; $k \in \mathbb{N}$

Question : existe-t-il un ensemble $V' \subseteq V$ de taille $|V'| \leq k$ tel que toute arête $(u, v) \in E$ a au moins une de ses extrémités dans V' ($u \in V'$ ou $v \in V'$) ?

Set-Cover

Entrées : un ensemble d'éléments U ; une famille $S \subset \mathcal{P}(U)$ de sous-ensembles de U ; un nombre $k \in \mathbb{N}$

Question : existe-t-il une sous-famille $S' \subseteq S$ telle que :

- S' est une couverture de U, càd $U = \cup_{S_i \in S'} S_i$
- $card(S') \leq k$

Clique

Entrées : un graphe $G = (V, E)$; $k \in \mathbb{N}$

Question : existe-t-il un ensemble $S \subseteq V$ tel que :

- $\forall u, v \in S^2$, $(u, v) \in E$ (le sous-graphe induit est complet)
- $|S| \geq k$

Conclusion de la section 2

Choses à savoir faire pour l'examen :

- Formaliser/modéliser un problème (donner entrées et question)
- Donner la nature d'un problème (décision ou optimisation)
- Connaître les problèmes classiques et leur classe
- Montrer qu'un problème est NP-complet (dans NP + NP-difficile)

Comment s'entraîner :

- Refaire le TD 1 (tous les exercices), le TD 2 (toutes les questions sauf la 1.3 et la 2.2), et le TD 3 (exercice 1, questions 1 à 4).

Backtracking

Idée : explorer un *arbre de recherche* de solutions partielles, en construisant la solution étape par étape.

- À chaque étape : on choisit une option parmi plusieurs (branchements)
- Si la solution partielle ne peut plus mener à une solution valide, on revient en arrière (backtrack)
- Sinon, on continue jusqu'à :
  - obtenir une solution complète valide
  - avoir épuisé toutes les options dans la branche : on backtrack et on passe à la branche suivante

Exemple de problème

Problème Subset-Sum

Entrées :

- Une liste de nombres $A = [a_1, a_2, ..., a_n]$ positifs
- Un nombre $T \geq 0$

Question : existe-t-il un sous-ensemble de A dont la somme des éléments vaut T ?

On construit un arbre de solutions :

À la profondeur i, on a deux branchements :

- soit on prend a_i
- soit on ne prend pas a_i

Exemple de problème

Problème Subset-Sum

Entrées :

- Une liste de nombres $A = [a_1, a_2, ..., a_n]$ positifs
- Un nombre $T \geq 0$

Question : existe-t-il un sous-ensemble de A dont la somme des éléments vaut T ?

À chaque étape :

- si la somme courante $S > T$, ça ne sert à rien de continuer dans cette branche
- si au bout de la branche, on a encore $S < T$, on revient en arrière

Algorithme du backtracking

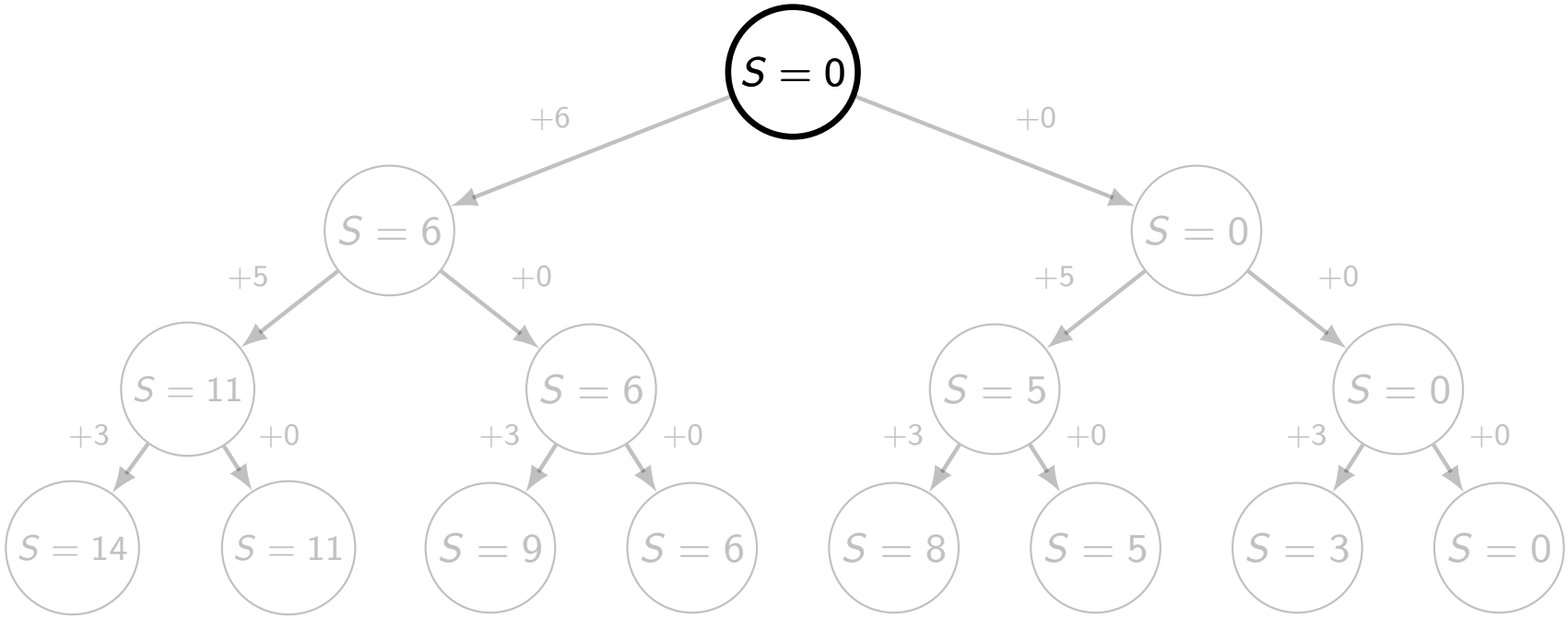
```
fonction SubsetSum(A, T)
    fonction BT(i, S)
        si S = T alors                          ▷ on a trouvé une solution !
            retourner True
        si i = n alors                          ▷ on a épuisé tous les nombres de A
            retourner False
        si S > T alors                          ▷ on a dépassé T, inutile de continuer
            retourner False

        si BT(i+1, S + A[i]) alors              ▷ on prend l'élément a_i
            retourner True
        si BT(i+1, S) alors                     ▷ on ne prend pas l'élément a_i
            retourner True
        retourner False                         ▷ pas de solution trouvée dans le sous arbre

    retourner BT(0, 0)
```

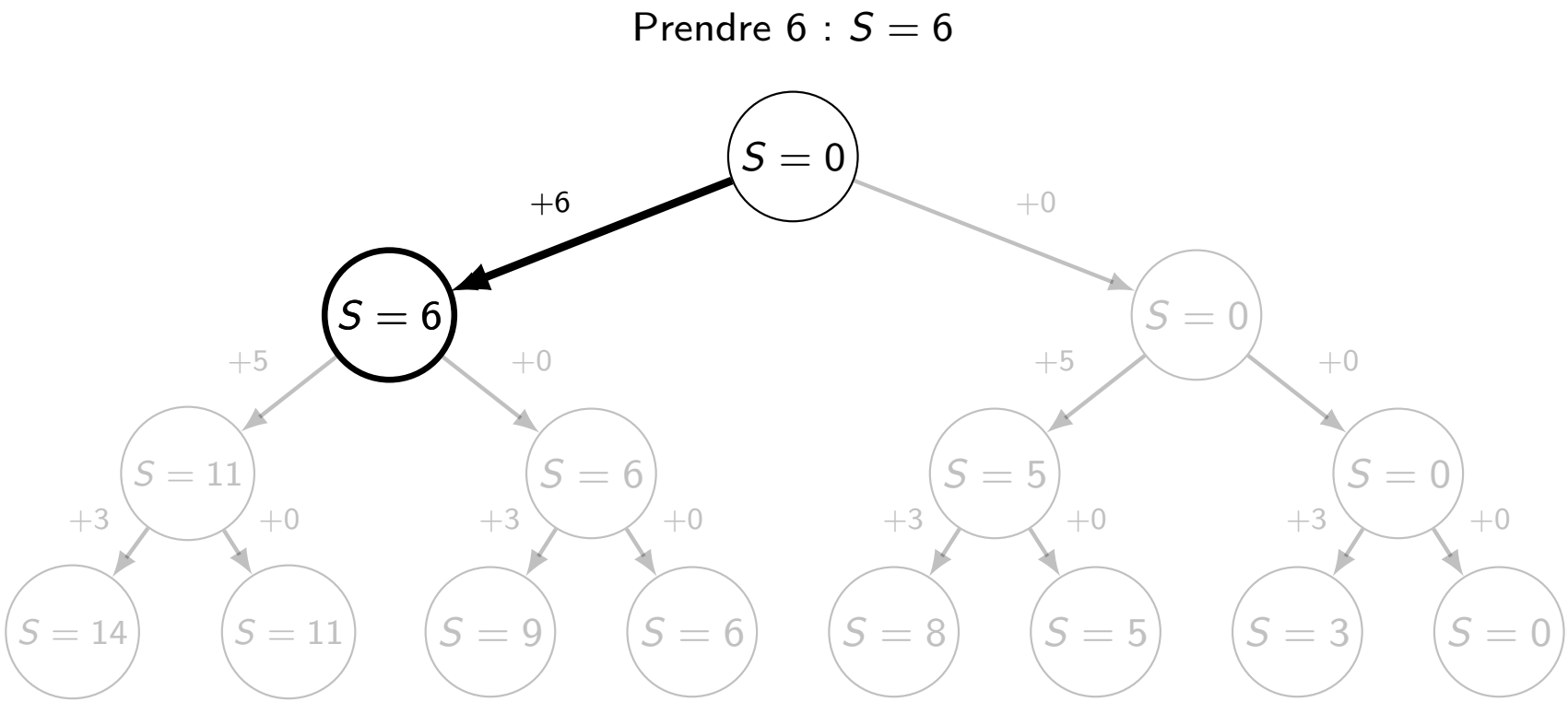
Backtracking : exécution (arbre de décision)

Départ avec $i = 0$ et $S = 0$



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

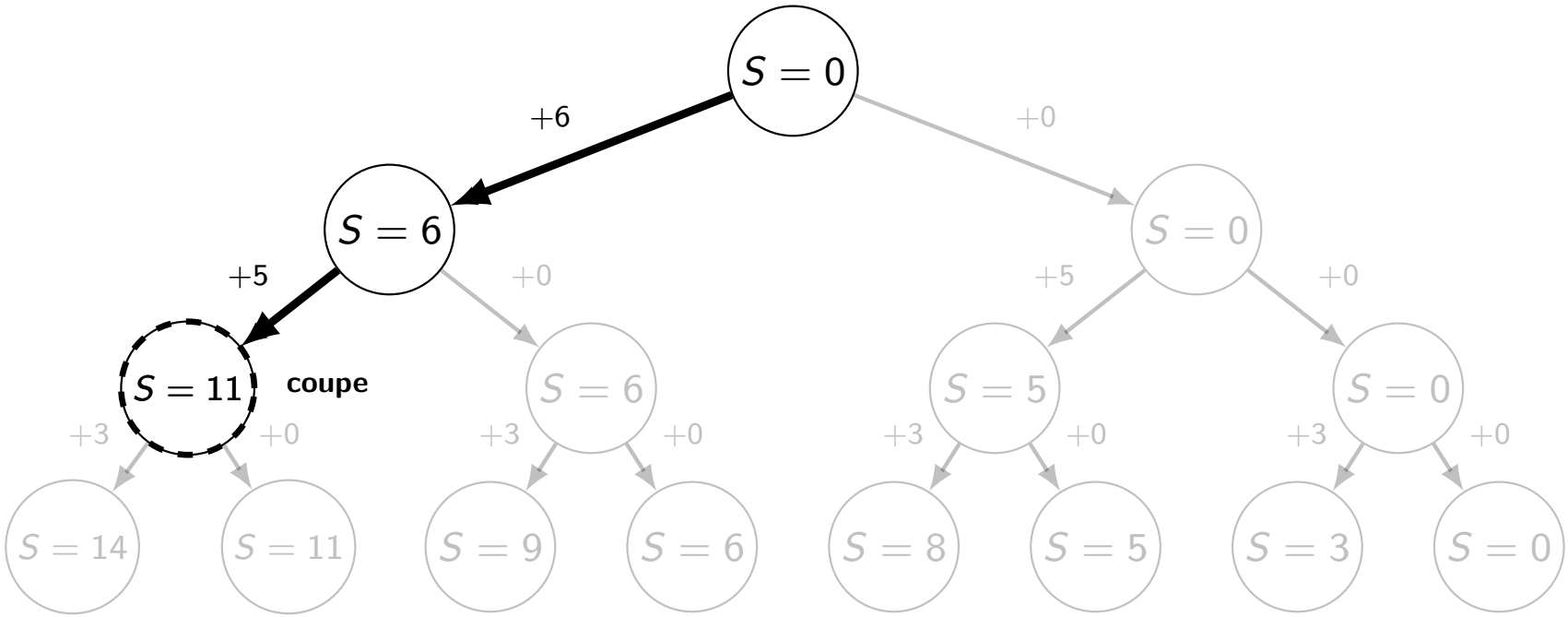
Backtracking : exécution (arbre de décision)



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

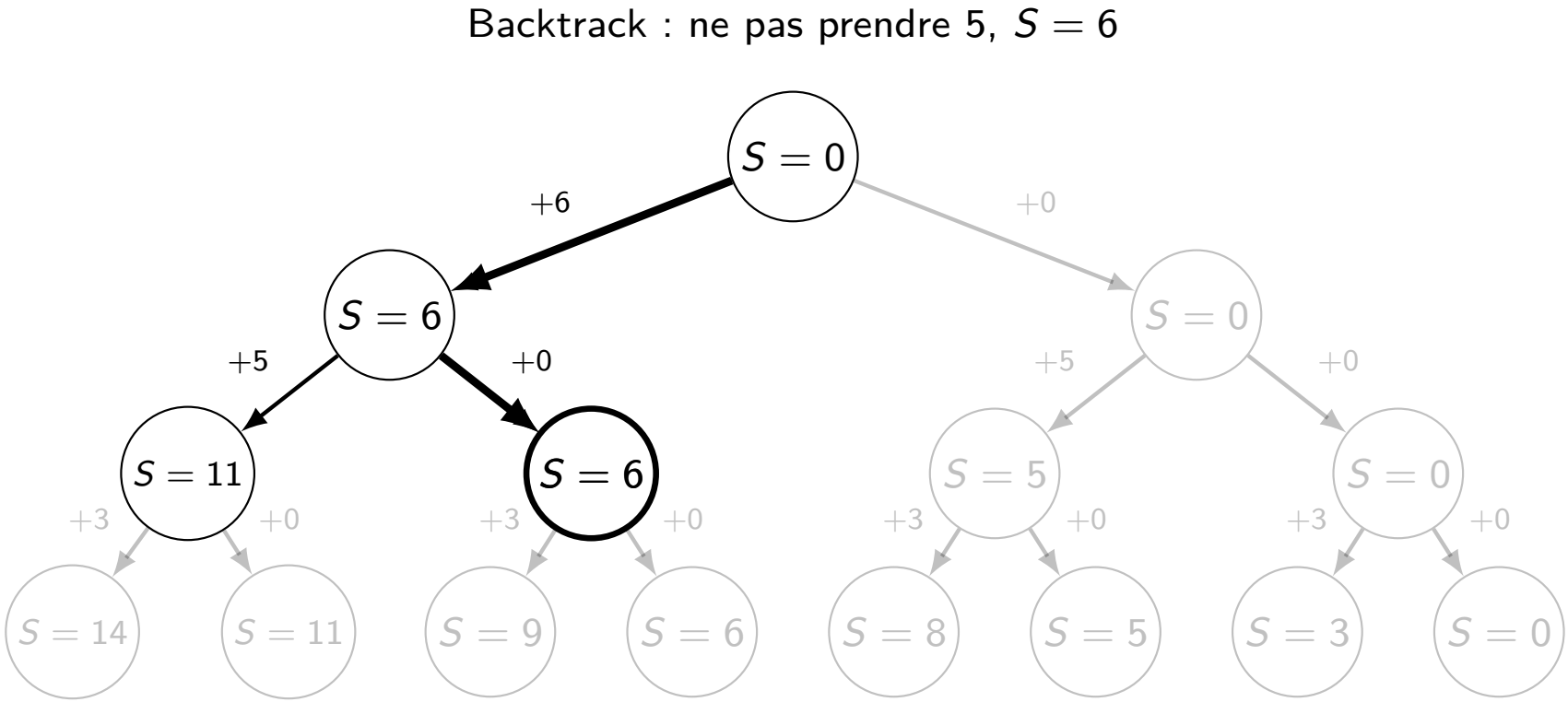
Backtracking : exécution (arbre de décision)

Prendre 5 : $S = 11 > 8 \Rightarrow$ on coupe



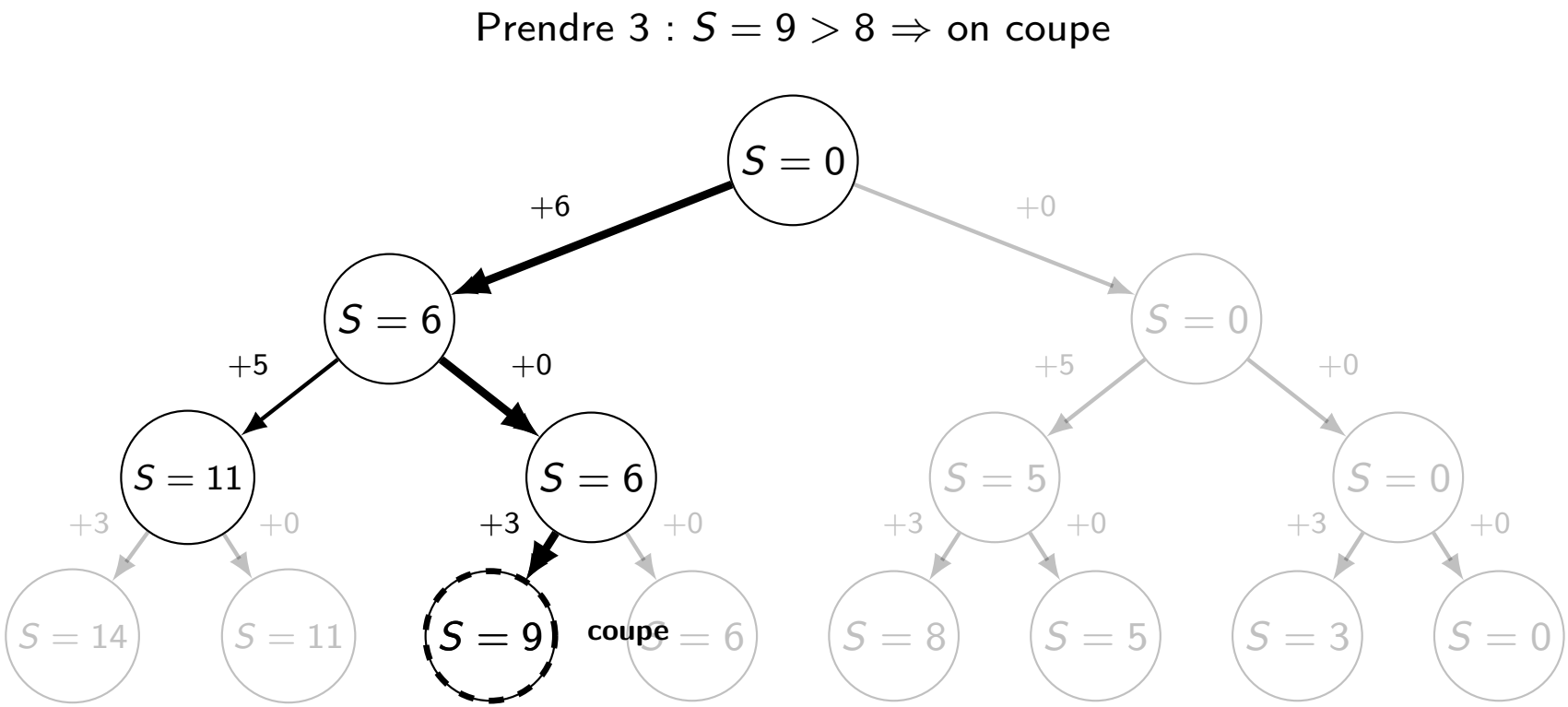
$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

Backtracking : exécution (arbre de décision)



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

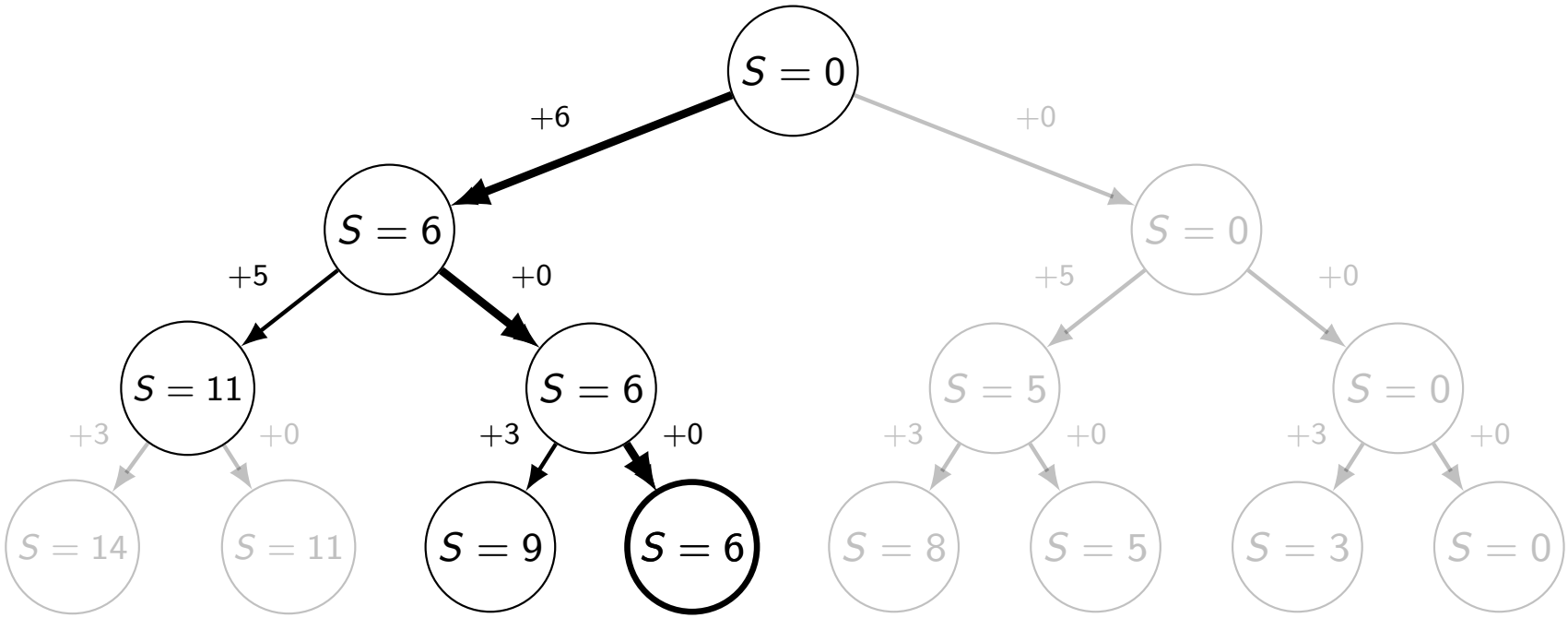
Backtracking : exécution (arbre de décision)



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

Backtracking : exécution (arbre de décision)

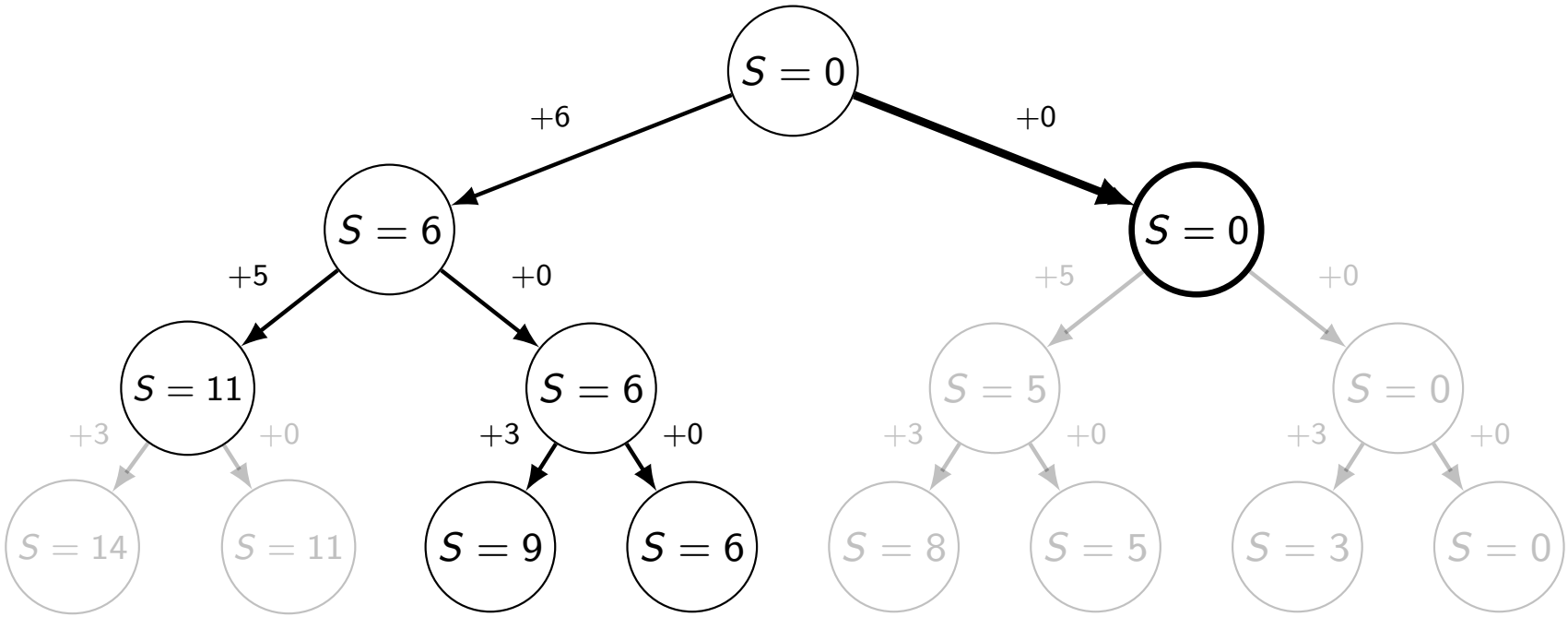
Backtrack : ne pas prendre 3. la branche échoue



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

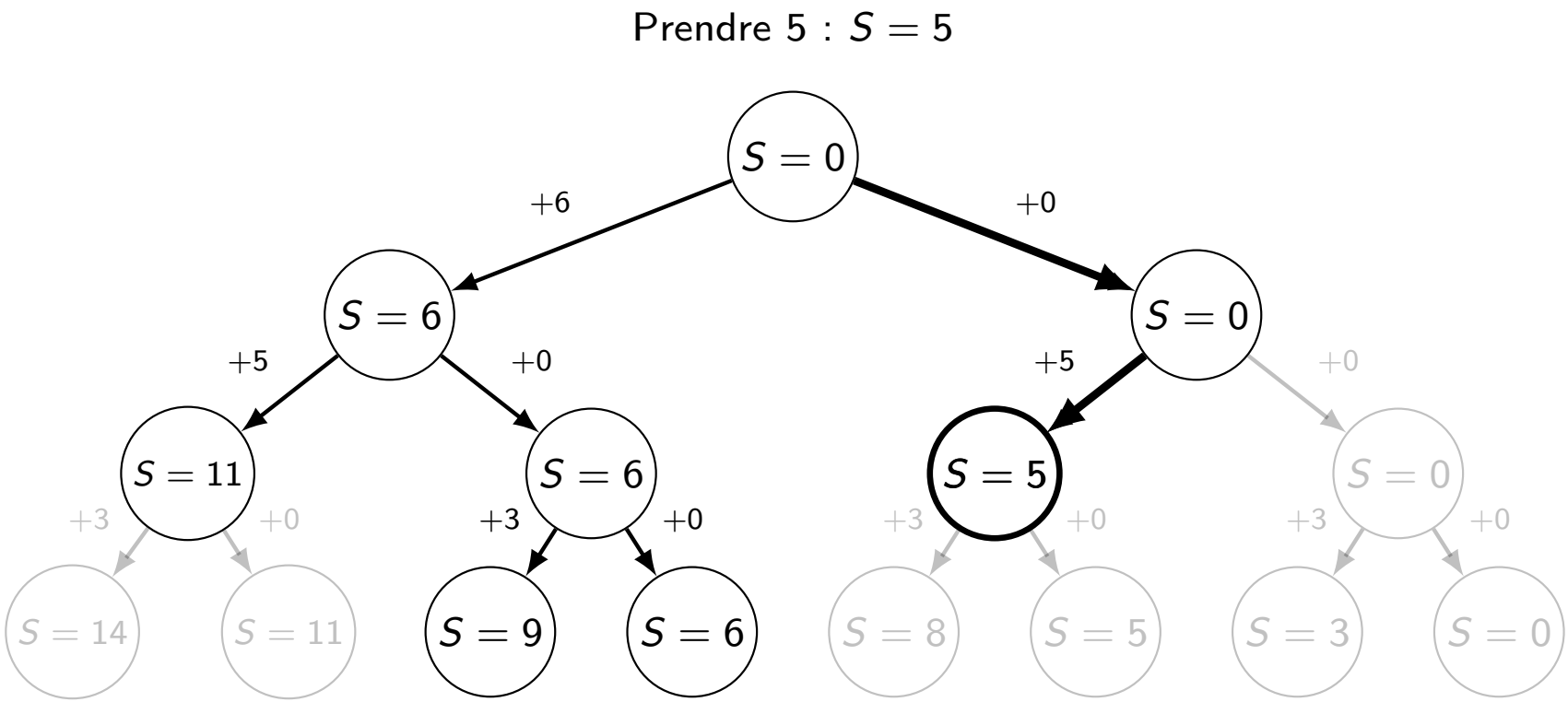
Backtracking : exécution (arbre de décision)

Retour racine : ne pas prendre 6



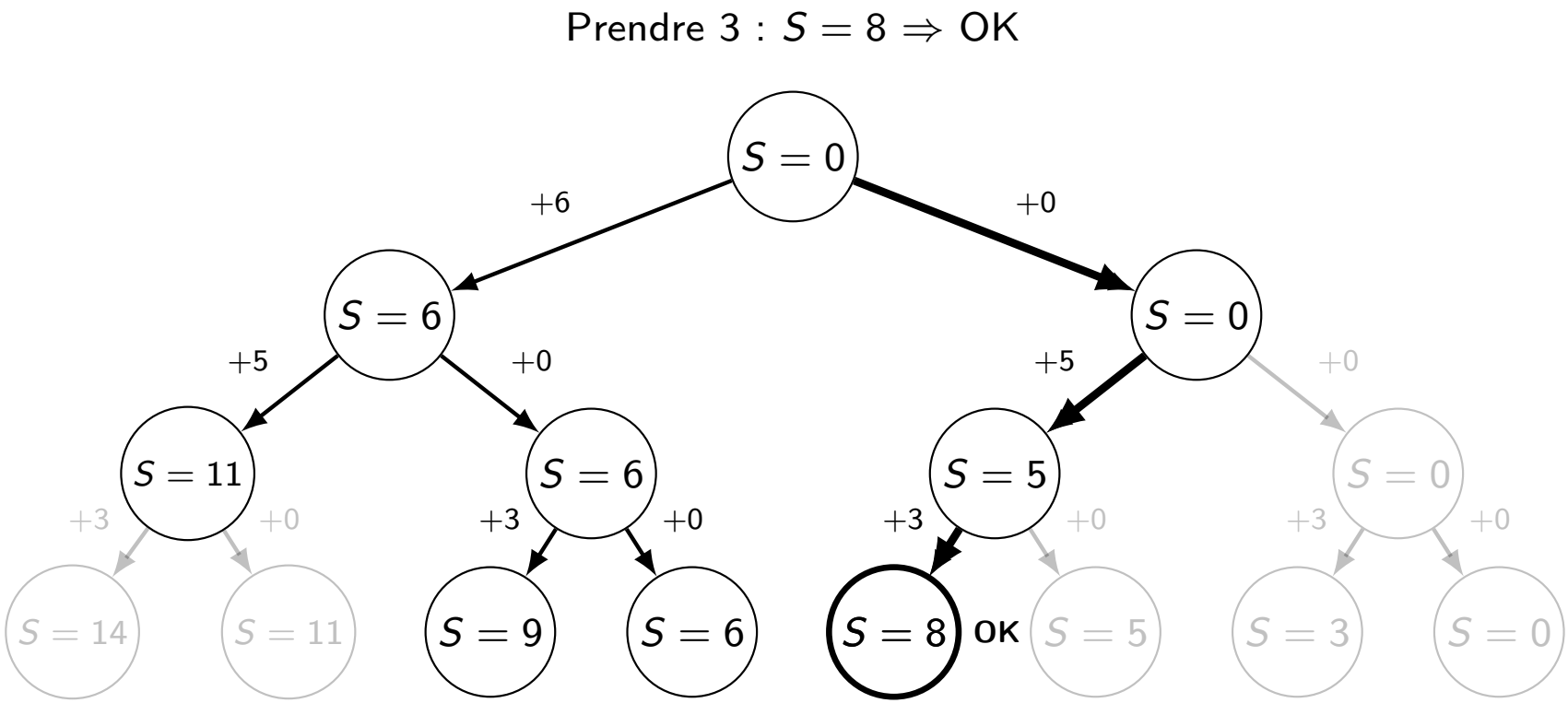
$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

Backtracking : exécution (arbre de décision)



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

Backtracking : exécution (arbre de décision)



$A = [6, 5, 3]$, cible $T = 8$. Pruning : si $S > T$, on coupe la branche

Complexité du backtracking

Le nombre de nœuds de l'arbre est en ici $O(2^n)$: n choix (niveaux) avec 2 options à chaque fois (prendre / ne pas prendre).

Les opérations effectuées sur chaque nœud (vérification des cas de base, sommes) sont ici en $O(1)$.

D'où une complexité totale en $O(2^n)$ (exponentielle).

Dans le cas général, si les opérations effectuées sur les nœuds ne sont pas en $O(1)$, la complexité peut être encore moins bonne.

Pour des **problèmes d'optimisation**, on va chercher à parcourir le moins de branches possibles pour rapidement trouver la solution optimale.

Branch & Bound

On fait un backtracking mais avant d'explorer une branche, on calcule une borne optimiste sur le résultat qu'on peut obtenir (à l'aide d'une heuristique) pour savoir si, en fonction de la meilleure solution actuelle, ça vaut le coup d'explorer la branche.

Il faut toujours que la borne estimée soit optimiste (e.g surestimation pour un problème de maximisation) pour que Branch & Bound ne coupe pas une branche contenant la meilleure solution.

✨ La programmation linéaire en nombres entiers ✨

Optimisation (maximisation/minimisation) d'une grandeur à coefficients entiers.

Exemple de problème

Maximiser $Z = 100x_1 + 150x_2$ sous les contraintes :

- $8000x_1 + 4000x_2 \leq 40000$
- $15x_1 + 30x_2 \leq 200$
- $x_1, x_2 \in \mathbb{N}$

Ce qu'il faut retenir : on peut utiliser le branch and bound pour trouver la solution optimale dans ce genre de problèmes.

La dernière question de l'exercice 2 du TD 3 traite de ce sujet.

Conclusion de la section 3

Choses à savoir faire pour l'examen :

- Expliquer dans quels cas de figure on utilise le backtracking/branch and bound
- Écrire un algorithme de backtracking
- Justifier une complexité exponentielle (dénombrement)

Comment s'entraîner :

- Refaire le TD 3 (exercice 2, en entier)

Points non traités

Je fais le choix de ne pas parler des approximations du Problème du Voyageur de Commerce parce qu'elles ont été traitées au TD Pratique 3.

Algorithme glouton

Algorithme qui fait des choix localement optimaux pour construire une solution.

Exemple : Kruskal

L'algorithme de Kruskal est un algorithme glouton : à chaque étape, il considère l'arête de **poids minimal** parmi celles restantes et ne créant pas de cycle.

En l'occurrence, Kruskal est un algorithme optimal mais ce n'est pas toujours le cas !

Dans le cas général, on va donc chercher à borner le coût de la solution renvoyée par un algorithme pour garantir sa qualité.

✨ α-approximation ✨

Soit un problème d'optimisation, dont f est la fonction de coût.

On dit qu'un algorithme est une α-approximation ($\alpha \geq 1$) pour ce problème si, pour toute entrée e, en notant S la solution renvoyée par l'algorithme pour e et S^* une solution optimale :

$$max\left(\frac{f(S)}{f(S^*)}, \frac{f(S^*)}{f(S)}\right) \leq \alpha$$

Minimisation : $f(S) \leq \alpha f(S^*)$

Maximisation : $f(S) \geq \frac{1}{\alpha} f(S^*)$

En pratique, on a souvent des solutions plus proches de l'optimale que de la borne.

Conclusion de la section 4

Choses à savoir faire pour l'examen :

- Écrire un algorithme glouton
- Montrer qu'un algorithme est une α-approximation d'un problème

Comment s'entraîner :

- Refaire le TD 4 (exercice 1)

Principe

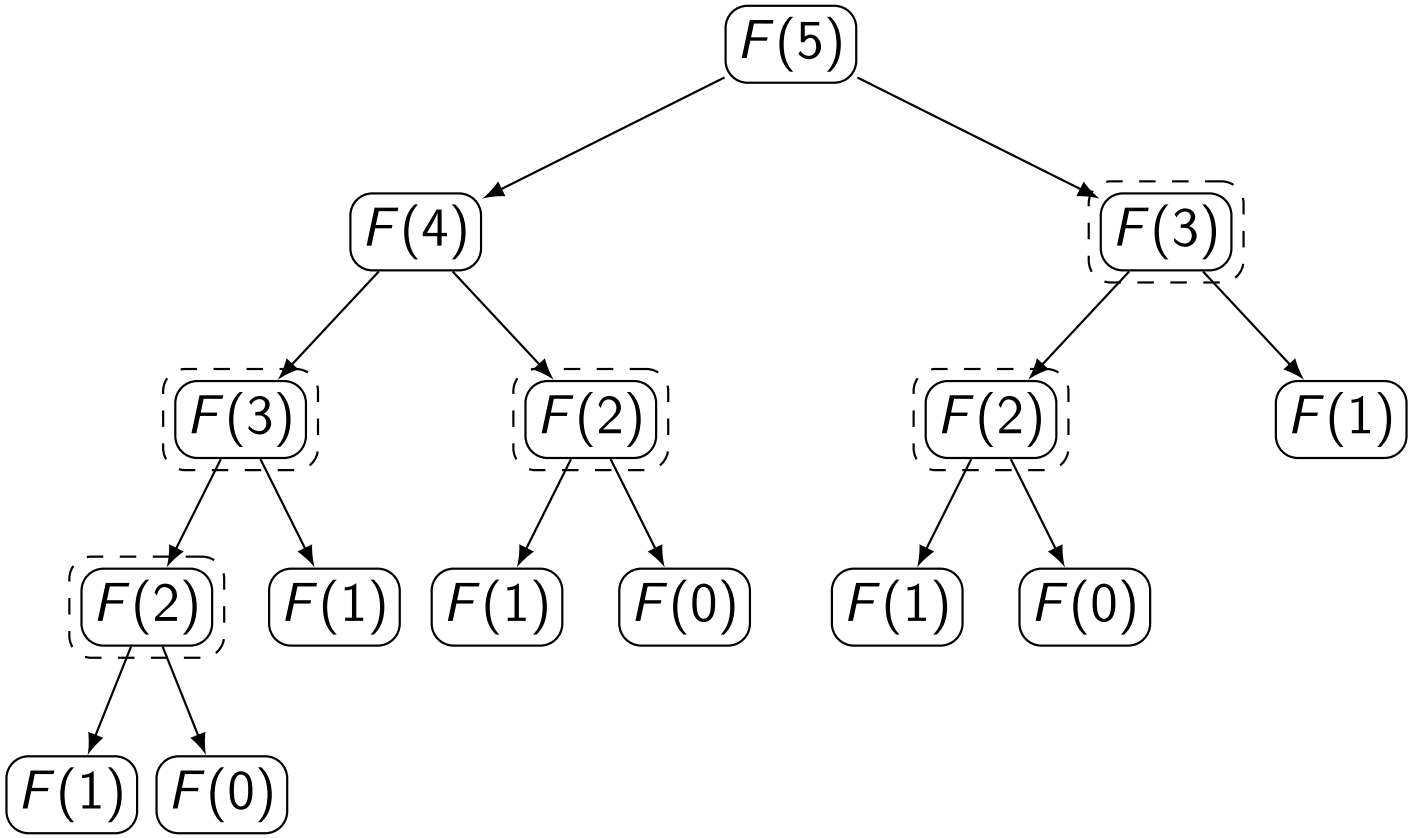
Stocker le résultat de calculs intermédiaires qui interviennent plusieurs fois dans le résultat pour éviter de les calculer plusieurs fois.

Dès qu'on a une formule de récurrence d'ordre 2 ou plus, on doit penser à la programmation dynamique.

Exemple : la suite de Fibonacci

Suite de Fibonacci

$$F(0) = F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$



Suite de Fibonacci

- Approche naïve (récursive)

```
def fib(n):
  if n==1 or n==2:
    return 1
  return fib(n-1)+fib(n-2)
```

$\Rightarrow$ complexité exponentielle $O(\phi^n)$ (ϕ le nombre d'or)

- Programmation dynamique (récursive avec mémoïsation)

```
table = {0:0, 1:1}
def fib(n):
  if not n in table:
    table[n] = fib(n-1) + fib(n-2)
  return table[n]
```

$\Rightarrow$ complexité linéaire $O(n)$

Suite de Fibonacci

- Programmation dynamique (itérative avec tabulation)

```
table = {0:0, 1:1}
def fib(n):
    for i in range(2, n+1):
        table[i] = table[i-1] + table[i-2]
    return table[n]
```

$\Rightarrow$ complexité linéaire $O(n)$

Dans les exemples qui suivent et dans tout le cours, on utilise la programmation dynamique itérative (*bottom up*).

Un exemple d'algorithme très utile utilisant la programmation dynamique :

Algorithme de Bellman-Ford

Pour des graphes **pondérés** (même à poids négatifs), l'algorithme de Bellman-Ford nous donne le poids du plus court chemin entre un nœud de départ s et tous les autres nœuds du graphe.

L'algorithme a une moins bonne complexité temporelle que Dijkstra mais est plus général.

Principe de l'algorithme

On stocke $OPT(i, v)$ la longueur du plus court chemin entre le nœud départ et v, avec au plus i arêtes.

À chaque étape, on regarde s'il est plus court de garder le chemin à $i-1$ arêtes, ou d'utiliser un autre chemin déjà existant vers un voisin de v et d'y ajouter l'arête qui le relie à v. $(\star)$

Complexité : $O(|V| \times |E|)$ avec une liste d'adjacence, et $O(|V|^3)$ avec une matrice d'adjacence

Le choix $(\star)$ est traduit par la formule :

$$OPT(i, v) = \min\left(OPT(i-1, v), \min_{(u,v)\in E}(OPT(i-1, u) + w((u, v)))\right)$$

Bellman-Ford : algorithme

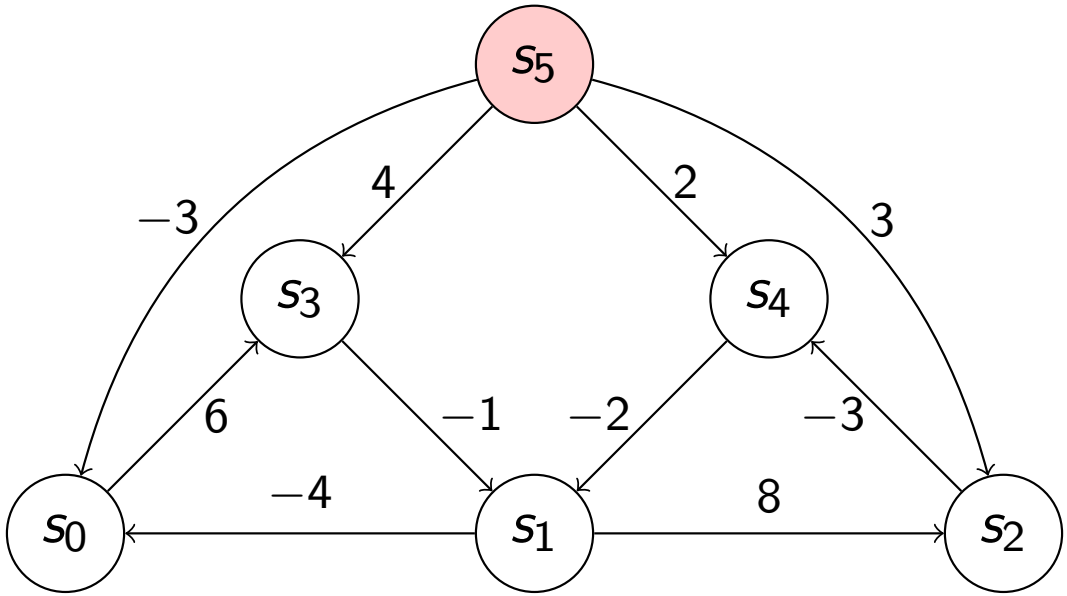
fonction BELLMAN-FORD($G = (V, E, w), s \in V$)
    $d[0, v] \leftarrow +\infty$ pour tout sommet $u \in V$
    $d[0, s] \leftarrow 0$

    pour i allant de 1 à $|V| - 1$ **faire**
        pour $v \in V$ **faire**
            pour $(u, v) \in E$ **faire**
                si $d[i, v] > d[i - 1, u] + w(u, v)$ **alors**
                    $d[i, v] = d[i - 1, u] + w(u, v)$
    retourner d

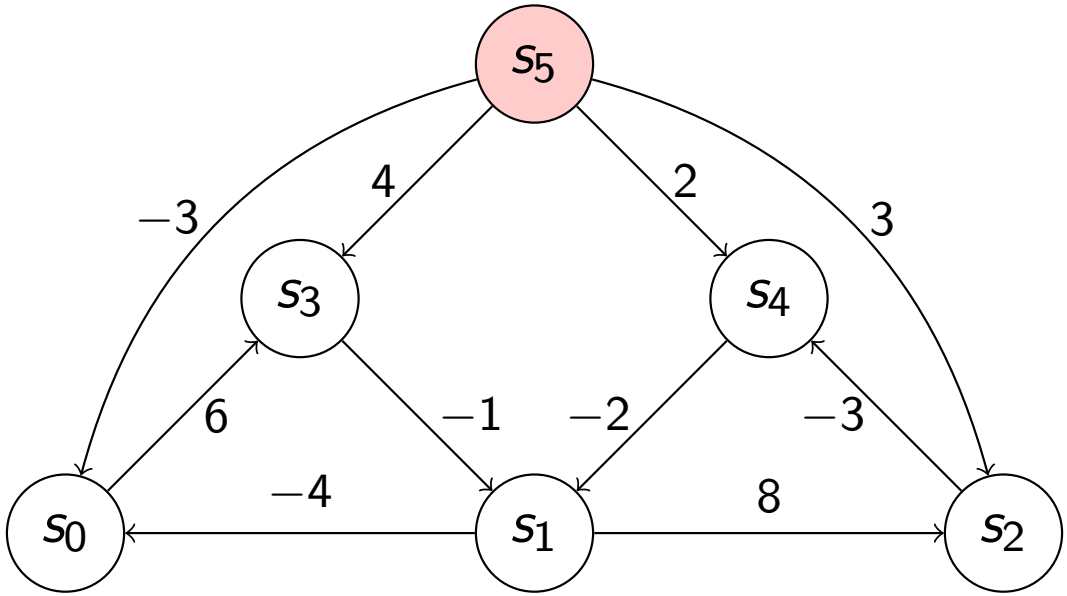
Justification de la première boucle

Un chemin optimal ne peut contenir que $|V| - 1$ arêtes au plus, sinon on aurait une répétition de sommets dans le chemin.

Bellman-Ford : exemple

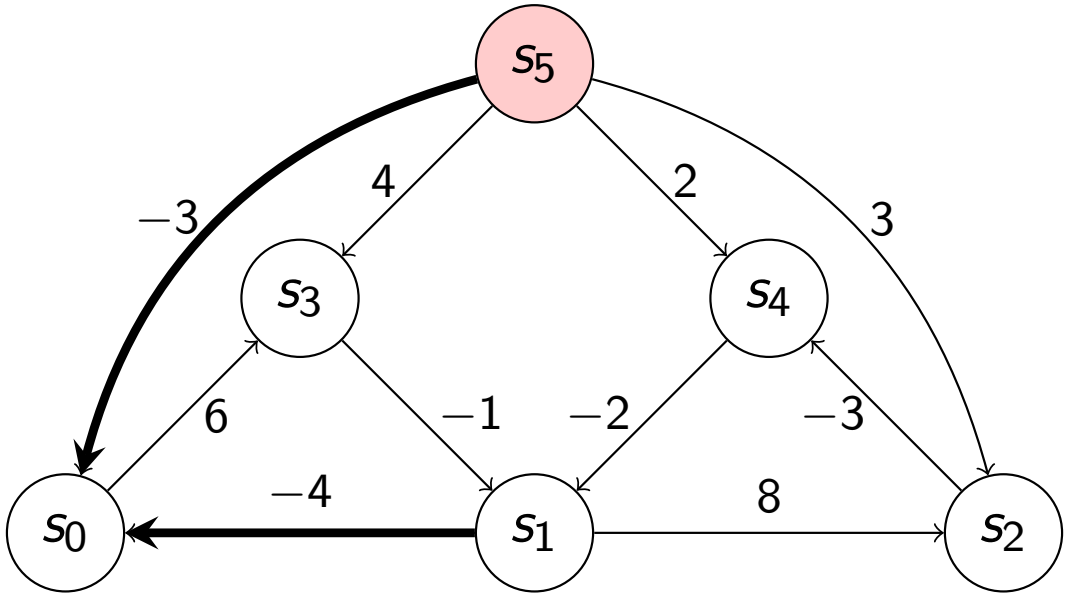


Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0

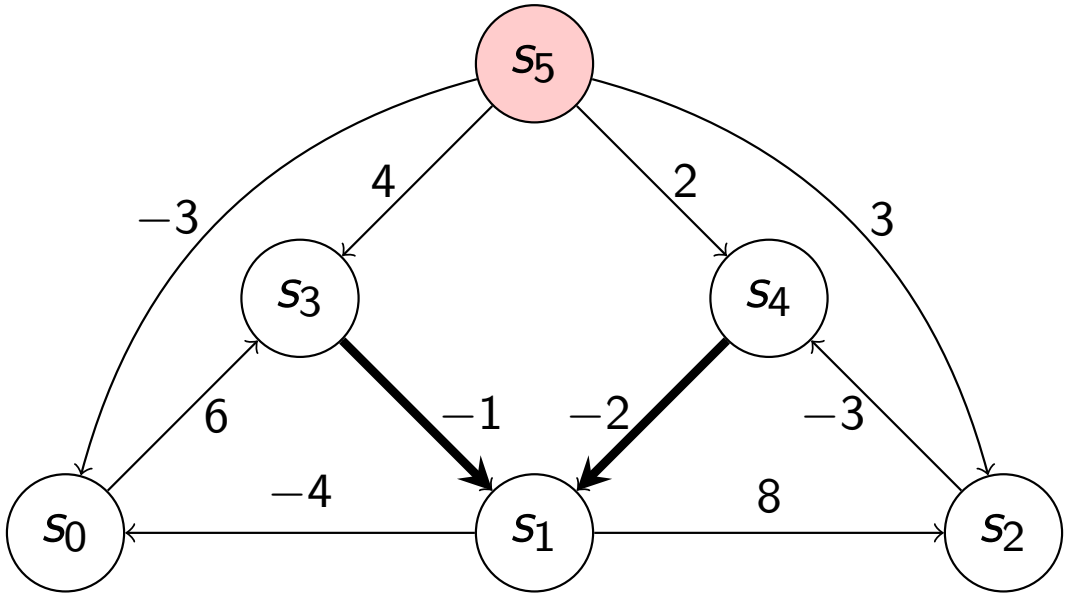
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3					

$s_0 = min(\infty, min(\infty - 4, 0 - 3))$

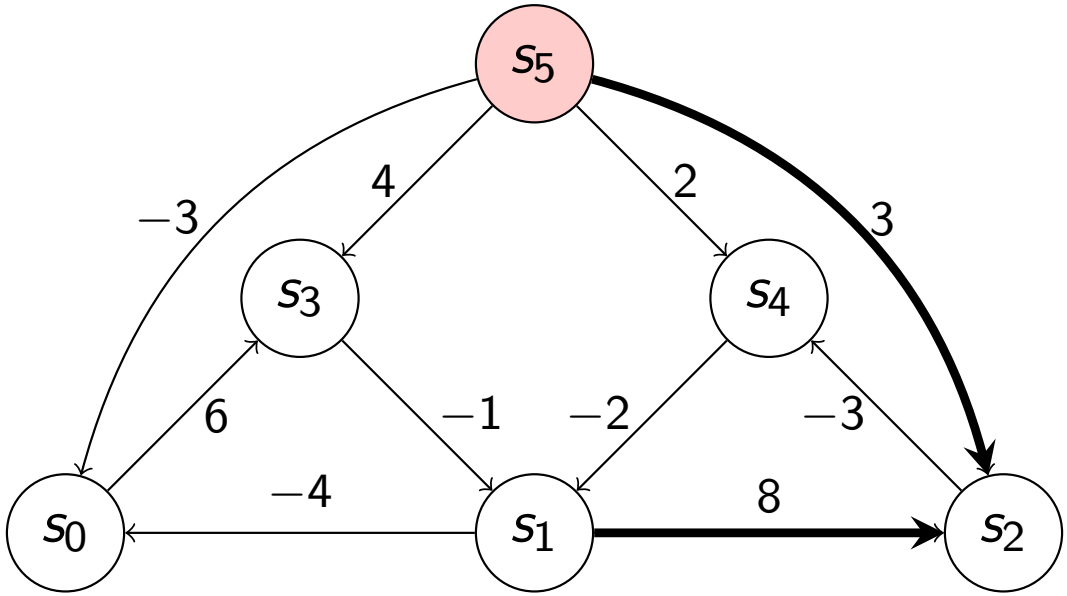
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞				

$s_1 = min(\infty, min(\infty - 1, \infty - 2))$

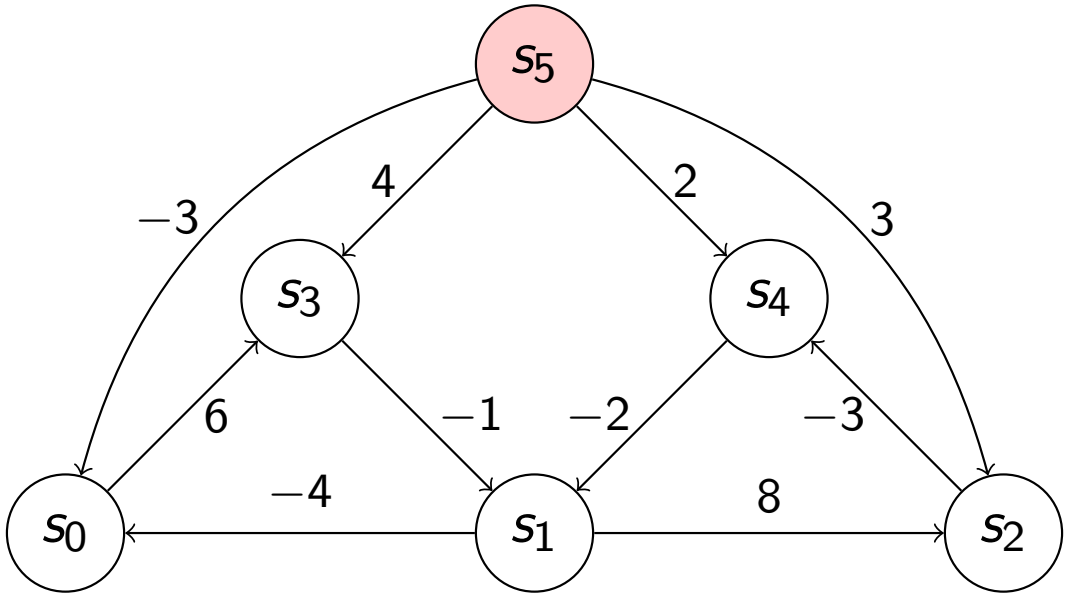
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3			

$$s_2 = min(\infty, min(\infty + 8, 0 + 3))$$

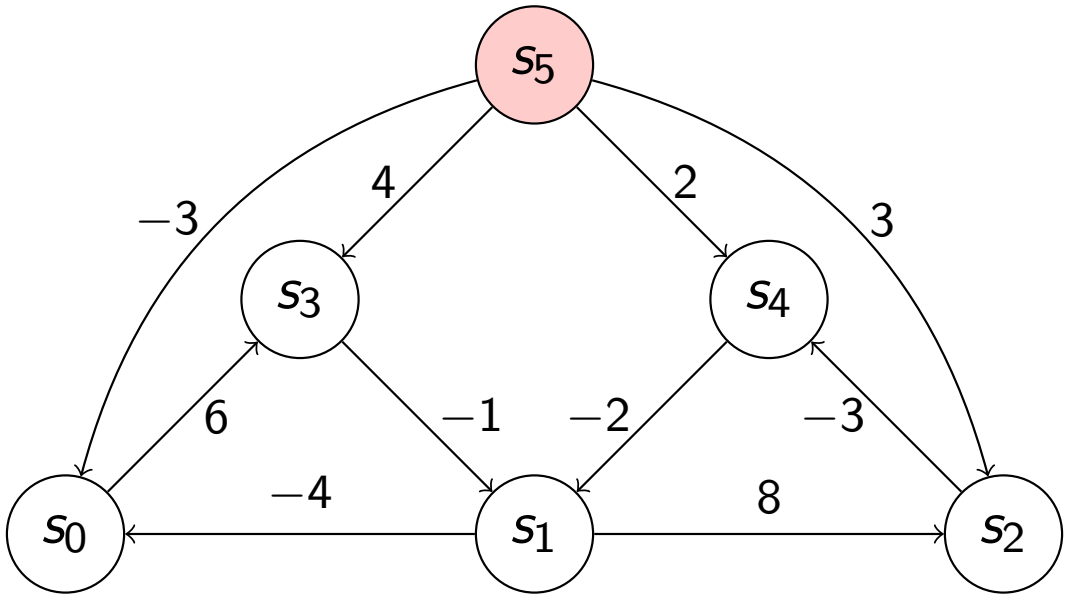
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0

on finit la ligne sur le même principe

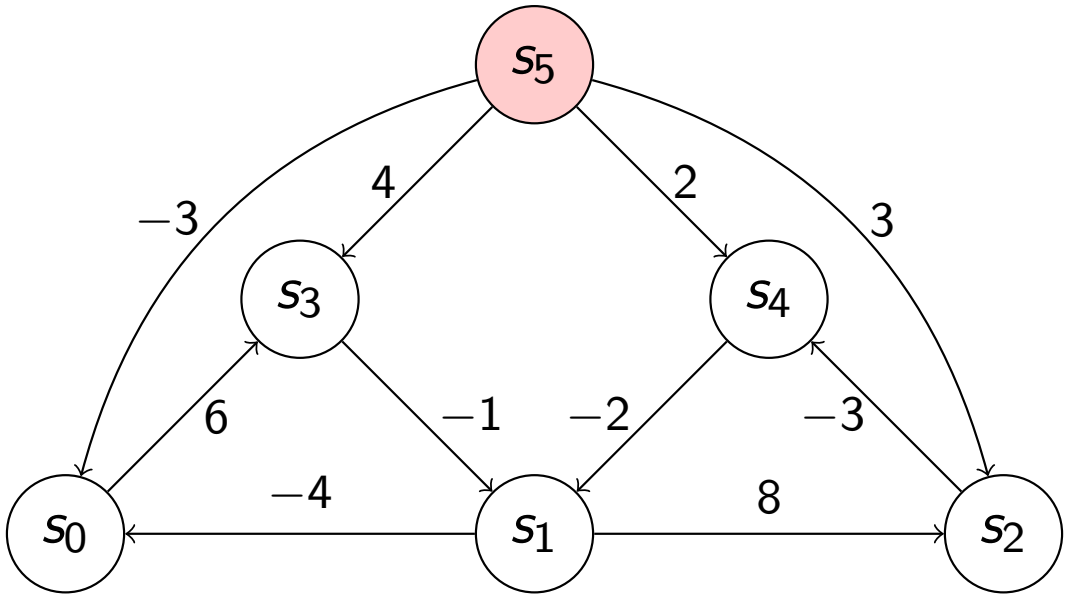
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0

puis on fait la ligne suivante

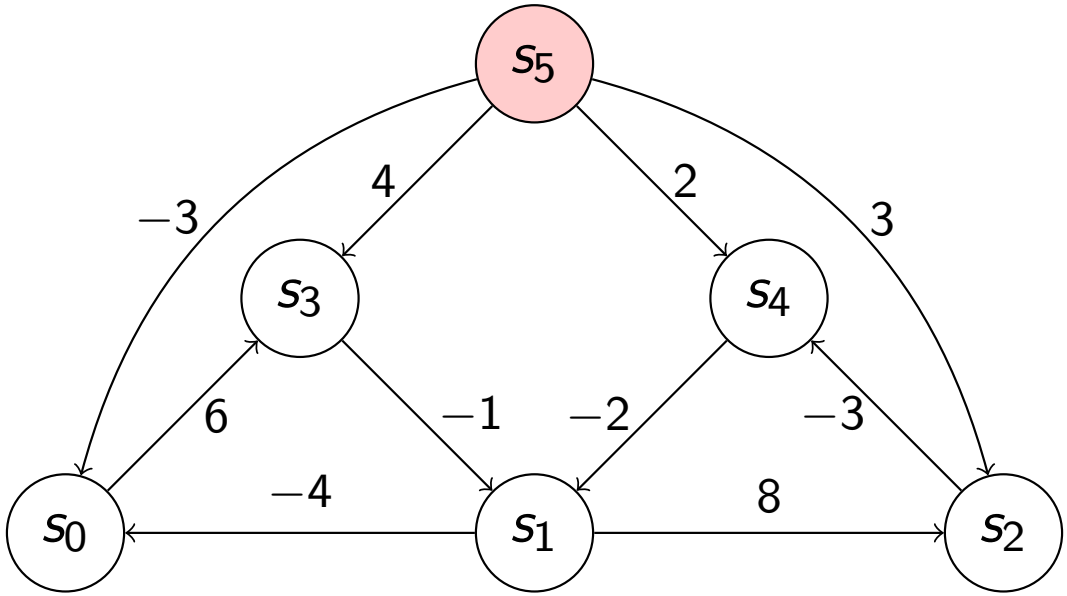
Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0

et ainsi de suite. . .

Bellman-Ford : exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	3	2	0	0
5	-6	-2	3	0	0	0

Comment trouver une relation de récurrence ?

Comment trouver la relation de récurrence pour un problème de programmation dynamique ?

- On commence par déterminer la dimension du problème (souvent 1D ou 2D)
- Trouver comment passer d'un état au suivant (la formule traduit souvent un choix)
- Identifier les cas de base

Subset Sum

Question : existe-t-il un sous-ensemble de $A = [a_1, a_2, ..., a_n]$ de somme T ?

$dp[i][s]$ = True si on peut faire la somme s avec les i premiers éléments

Relation de récurrence : $dp[i][s] = dp[i-1][s] \vee dp[i-1][s-a_i]$

Cas de base : $dp[0][0] =$ True

Conclusion de la section 5

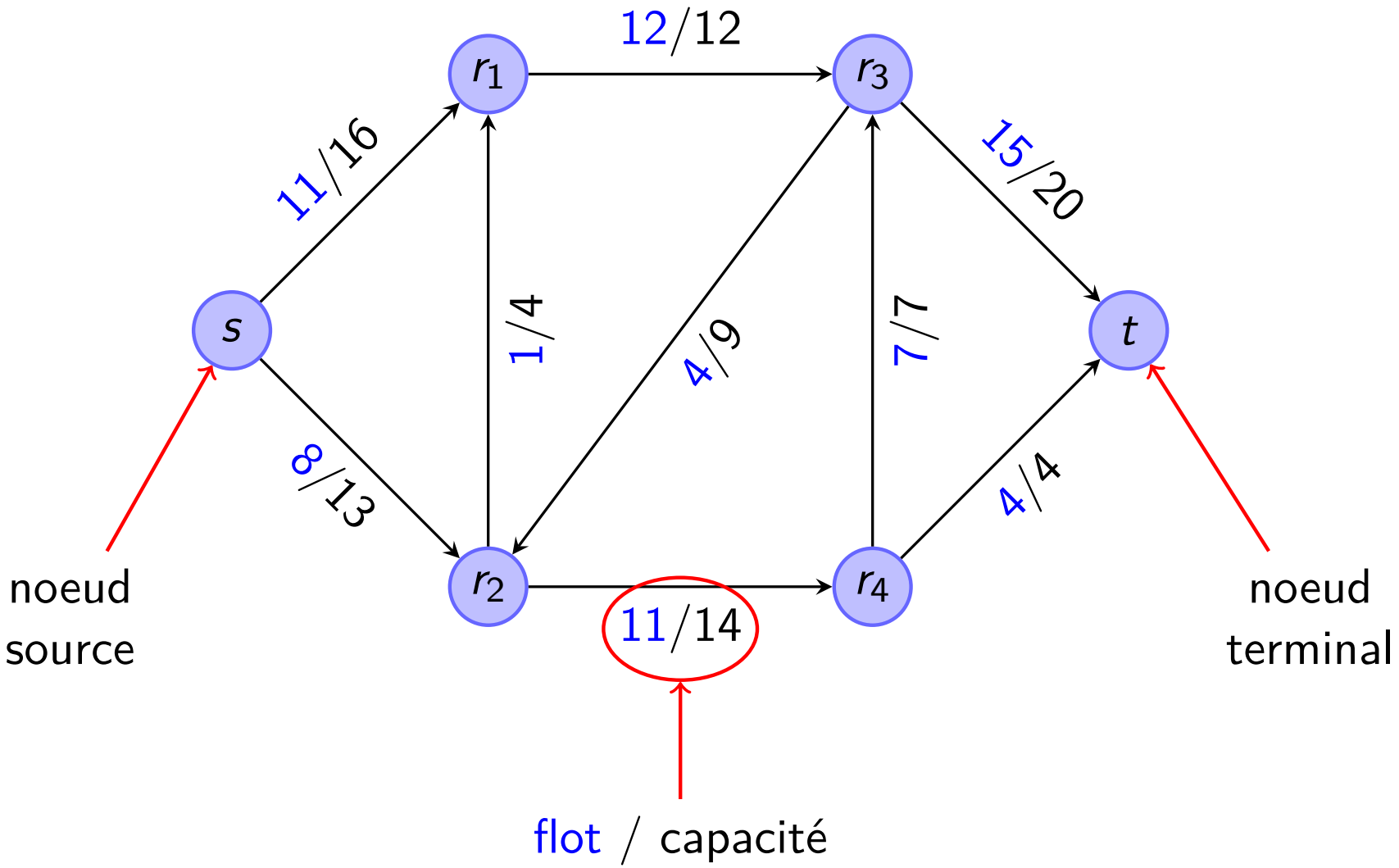
Choses à savoir faire pour l'examen :

- Trouver une relation de récurrence dans un problème et écrire l'algorithme itératif correspondant
- Modifier un algorithme récursif pour ajouter de la mémoïsation
- Faire tourner Bellman-Ford sur un exemple

Comment s'entraîner :

- Refaire le TD 5 (sauf la question 2.5)

Exemple de graphe de flot



Propriétés du flot

Un flot f réalisable doit vérifier les propriétés suivantes :

Règle flot-capacité

Le flot d'une arête ne peut pas dépasser sa capacité.

$$\forall\, u, v \in V^2 \quad 0 \leq f(u, v) \leq c(u, v)$$

Règle de conservation du flot

À l'exception de s et t, le flot entrant dans un nœud est égal au flot en sortant.

$$\forall\, u \in V \setminus \{s, t\} \quad \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

Flot total

Le flot sortant de s est égal à celui entrant dans t.

$$\sum_{u \in V} f(s,u) = \sum_{u \in V} f(u,t)$$

Valeur du flot

La valeur du flot f, notée φ est donnée par :

$$\varphi = \sum_{u \in V} f(s,u) = \sum_{u \in V} f(u,t)$$

Le problème du flot maximal consiste à trouver un flot f ayant la valeur du flot maximale φ_{max}

Ford-Fulkerson

L'algorithme de Ford-Fulkerson est un algorithme glouton qui permet de calculer le flot maximal dans un graphe de flot.

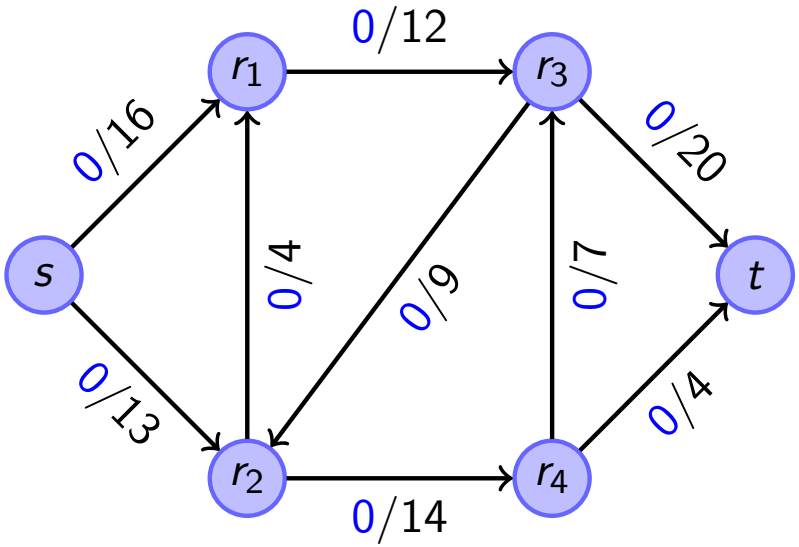
Complexité : $O((|V| + |E|) \times \varphi_{max})$

Remarque 1 : l'algorithme de Ford-Fulkerson n'impose aucune contrainte sur le type de parcours à effectuer. En pratique, on fait un DFS.

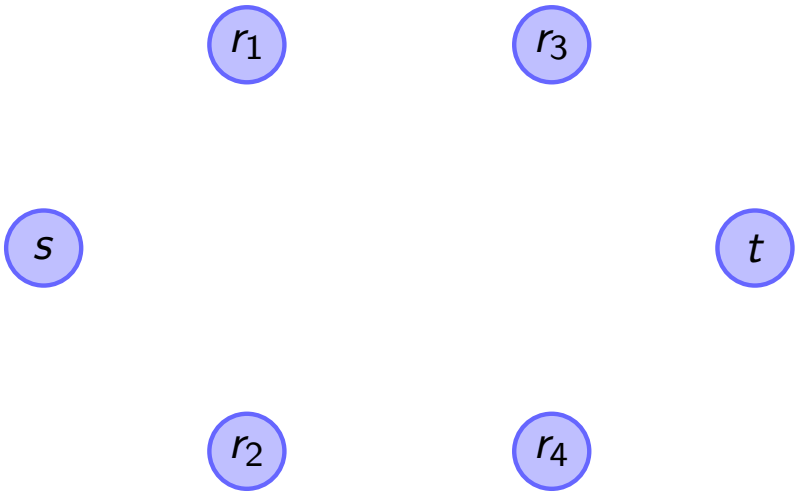
Remarque 2 : une variante de l'algorithme qui s'appuie sur le BFS (l'algorithme d'Edmonds-Karp) permet d'obtenir une complexité en $O(|V| \times |E|^2)$, indépendante de φ_{max}.

Algorithme de Ford-Fulkerson

Graphe de flot

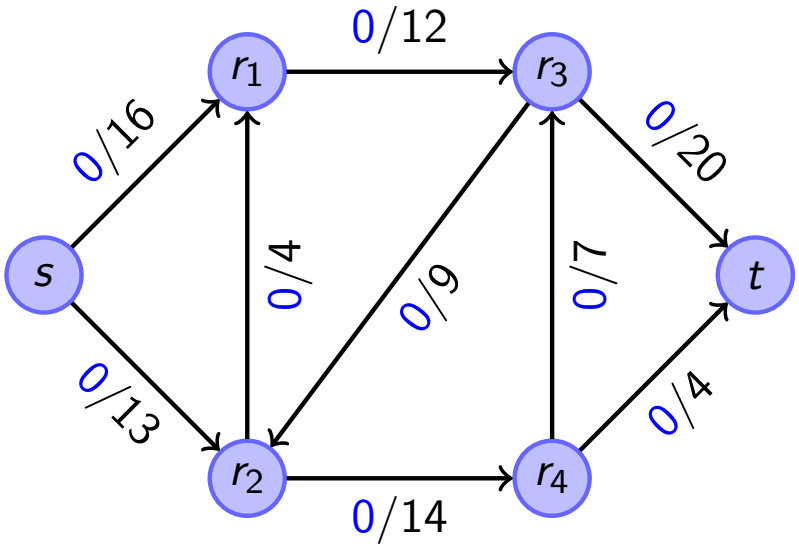


Graphe résiduel

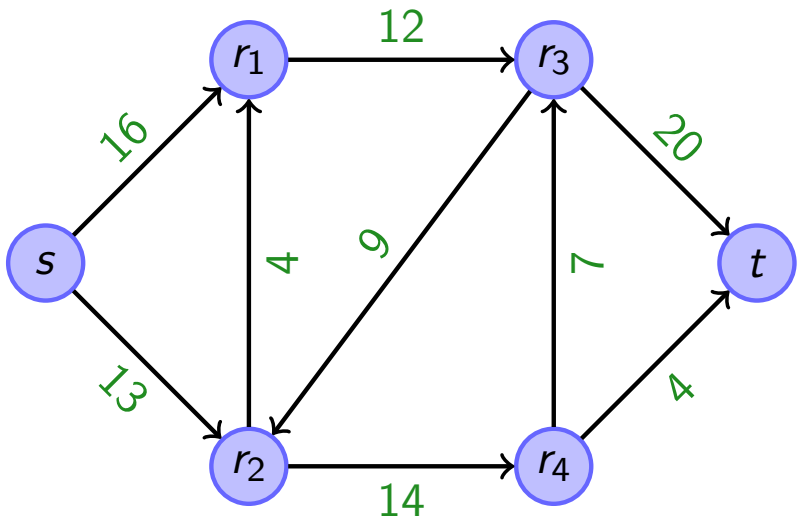


Algorithme de Ford-Fulkerson

Graphe de flot

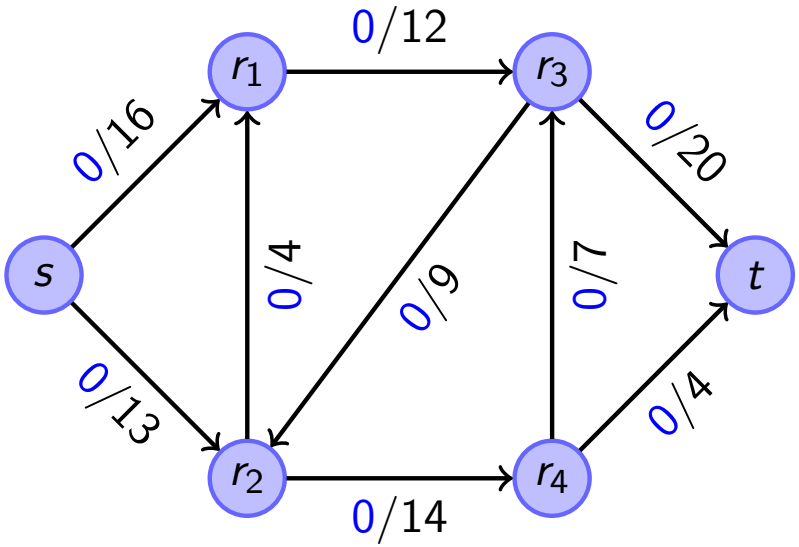


Graphe résiduel

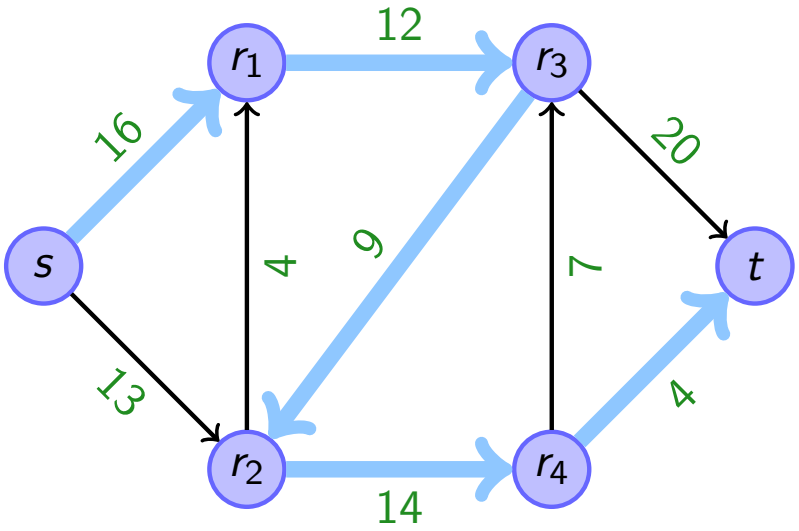


Algorithme de Ford-Fulkerson

Graphe de flot

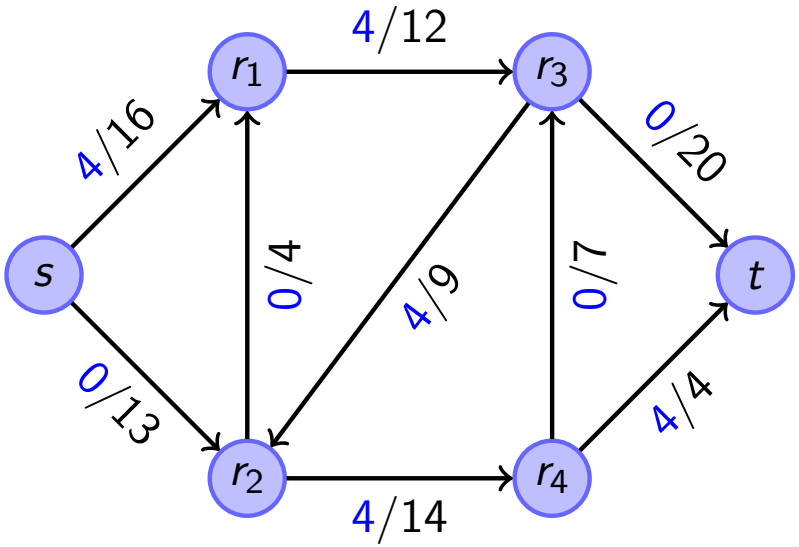


Graphe résiduel

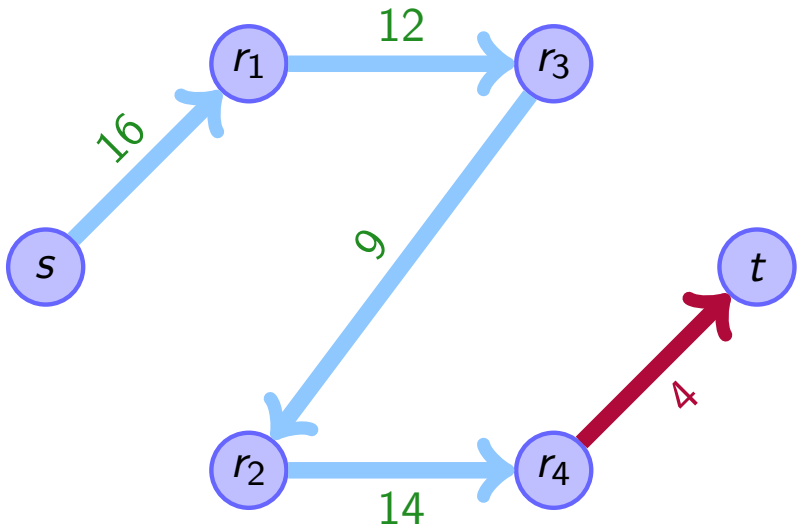


Algorithme de Ford-Fulkerson

Graphe de flot

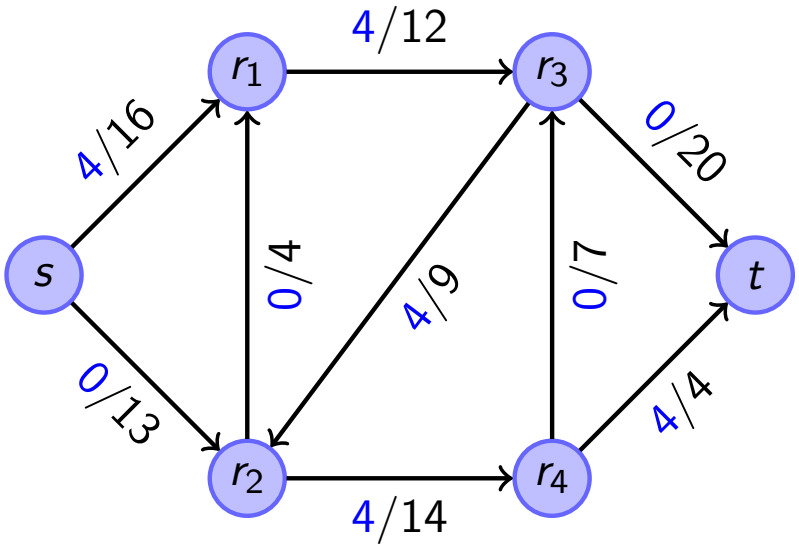


Graphe résiduel

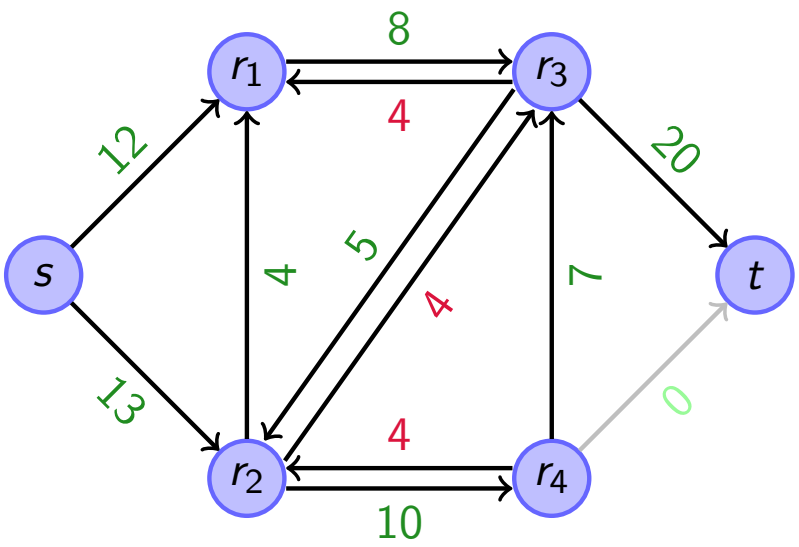


Algorithme de Ford-Fulkerson

Graphe de flot

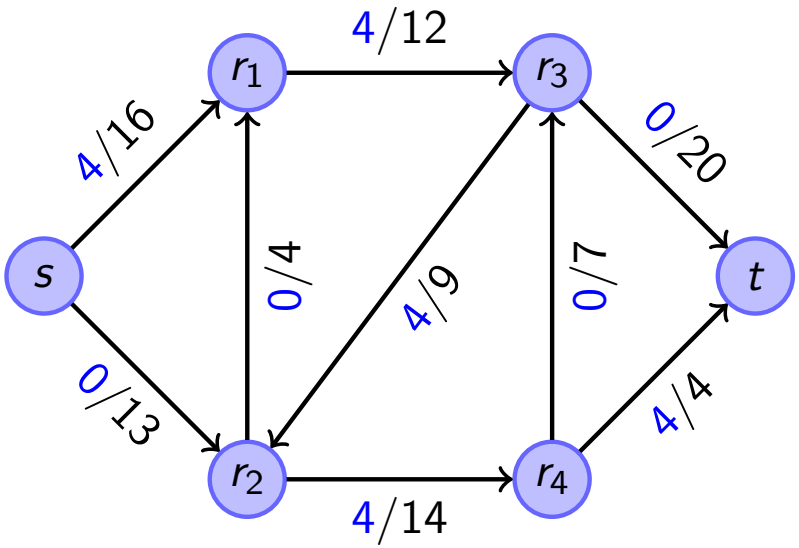


Graphe résiduel

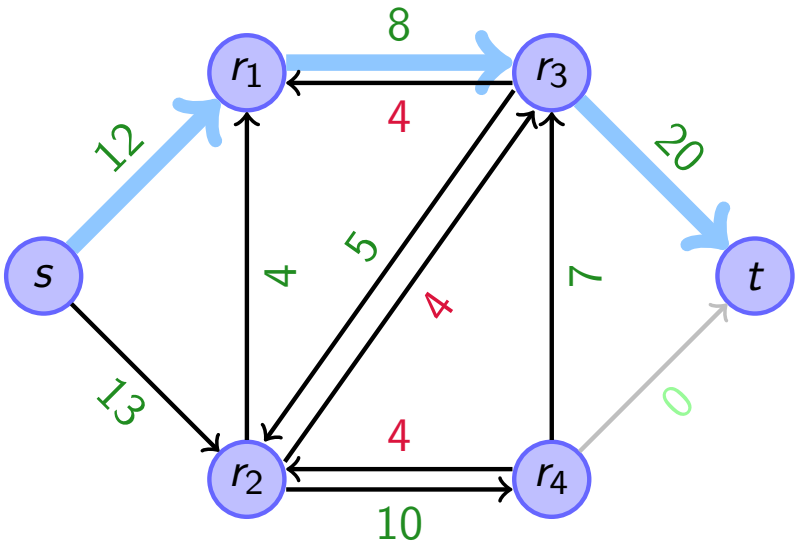


Algorithme de Ford-Fulkerson

Graphe de flot

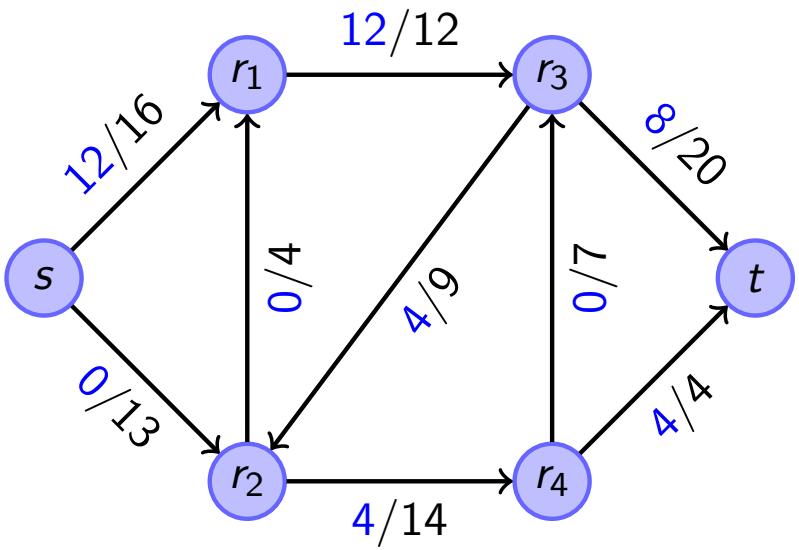


Graphe résiduel

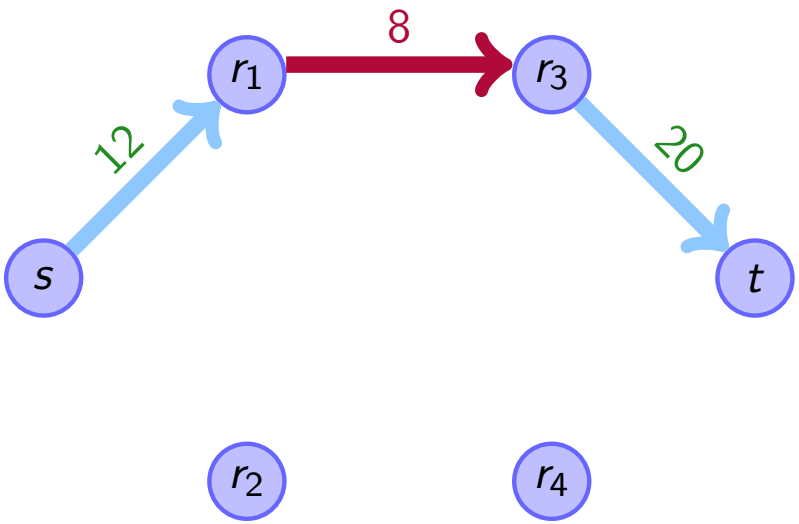


Algorithme de Ford-Fulkerson

Graphe de flot

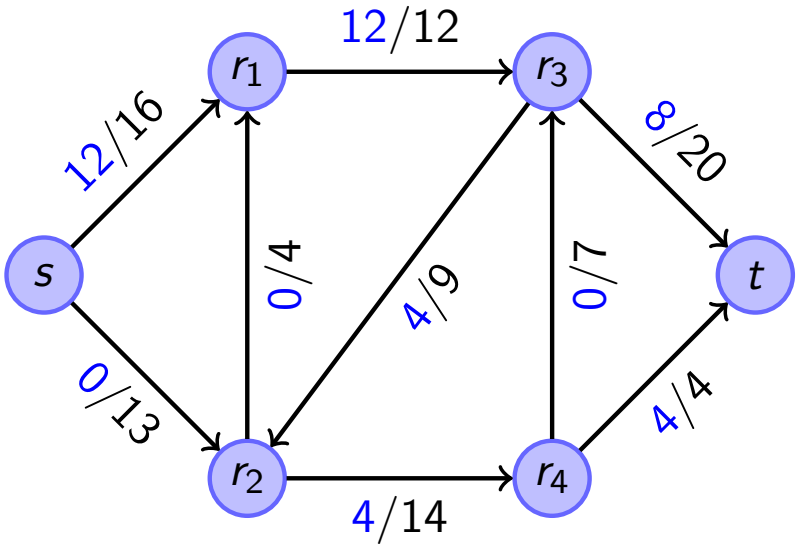


Graphe résiduel

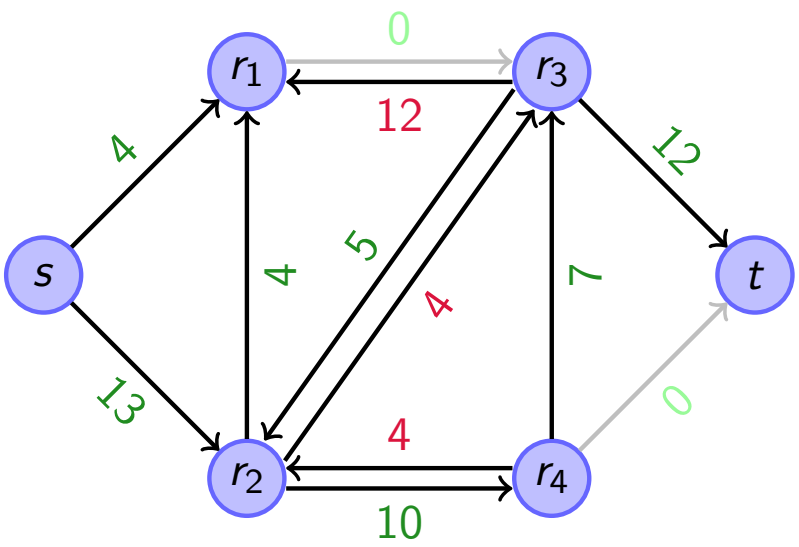


Algorithme de Ford-Fulkerson

Graphe de flot

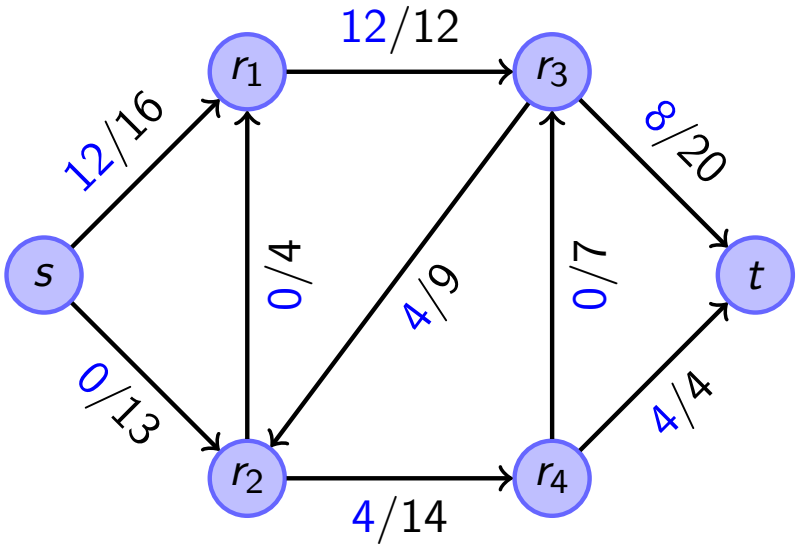


Graphe résiduel

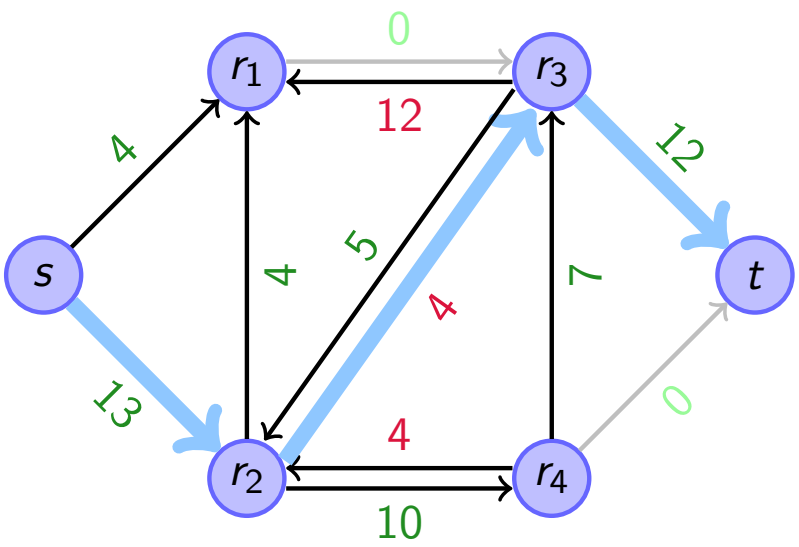


Algorithme de Ford-Fulkerson

Graphe de flot

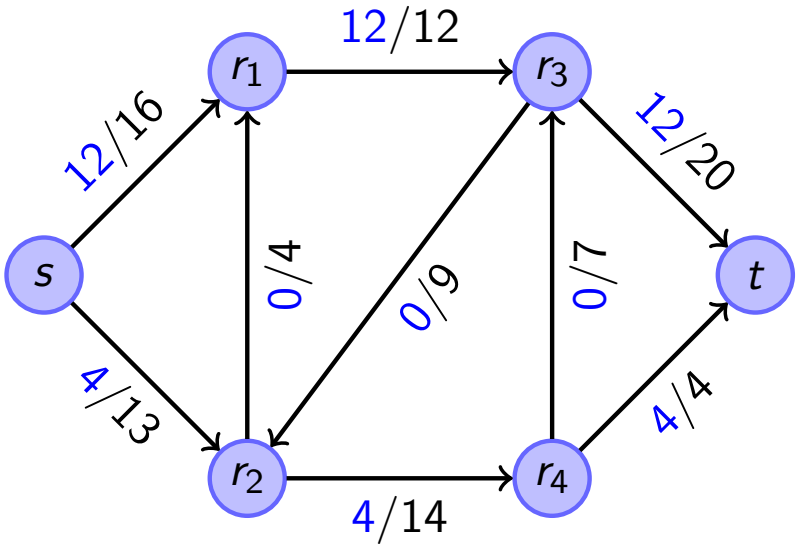


Graphe résiduel

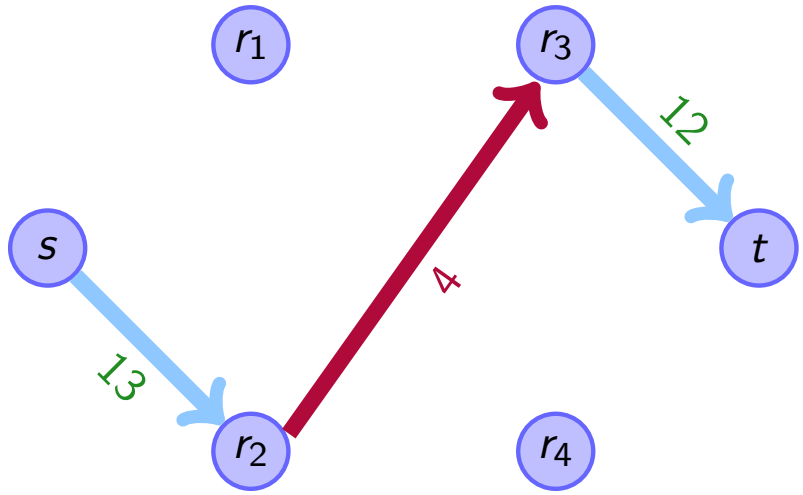


Algorithme de Ford-Fulkerson

Graphe de flot

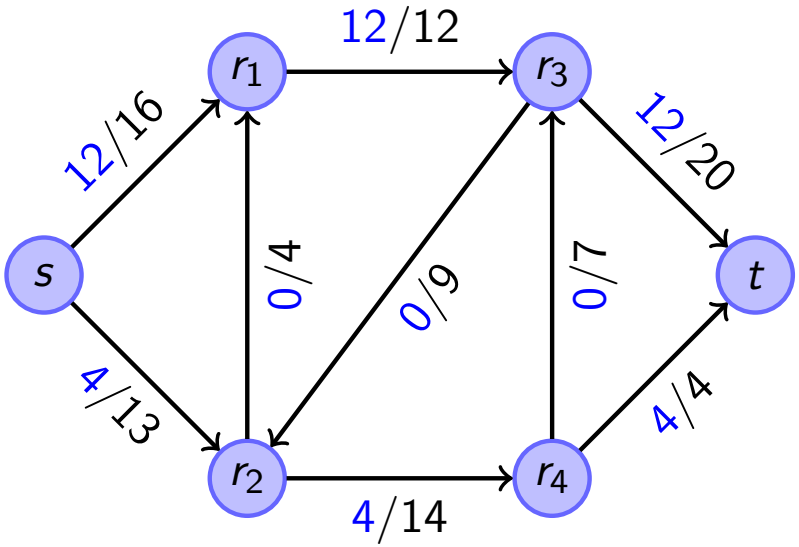


Graphe résiduel

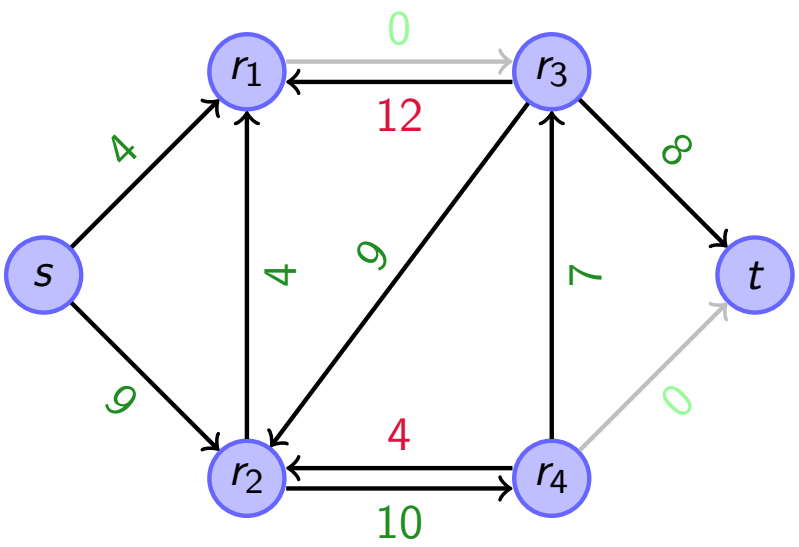


Algorithme de Ford-Fulkerson

Graphe de flot

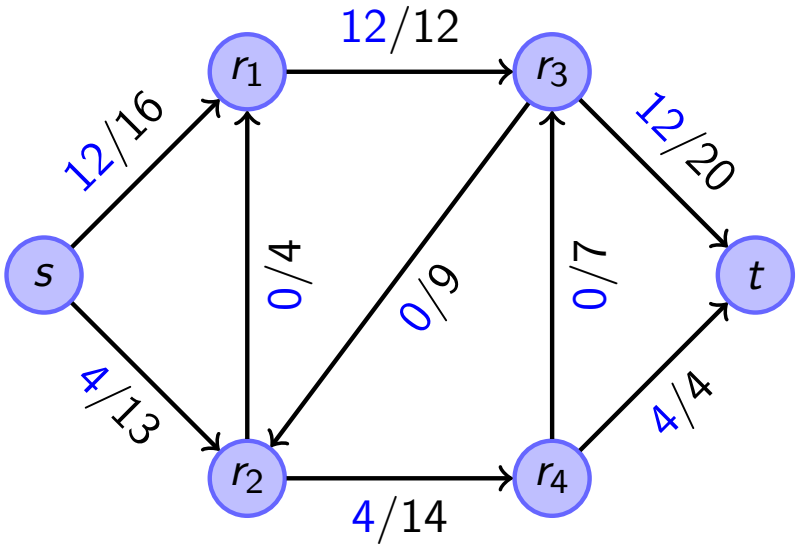


Graphe résiduel

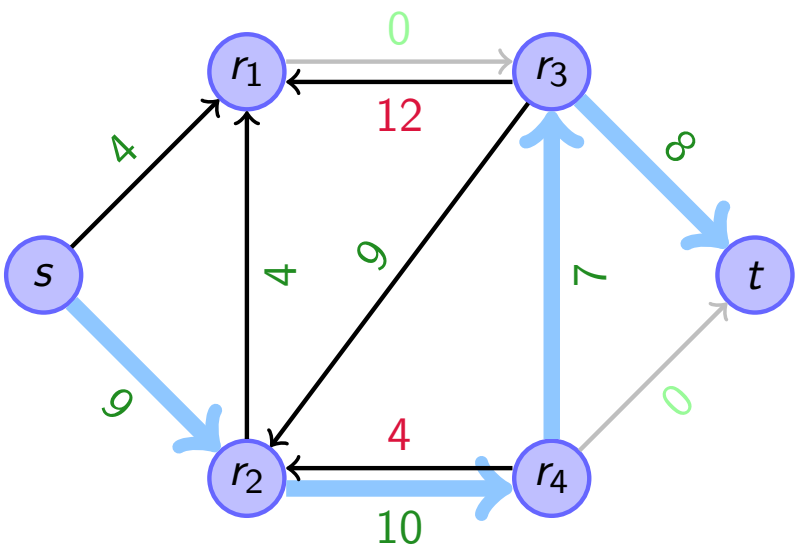


Algorithme de Ford-Fulkerson

Graphe de flot

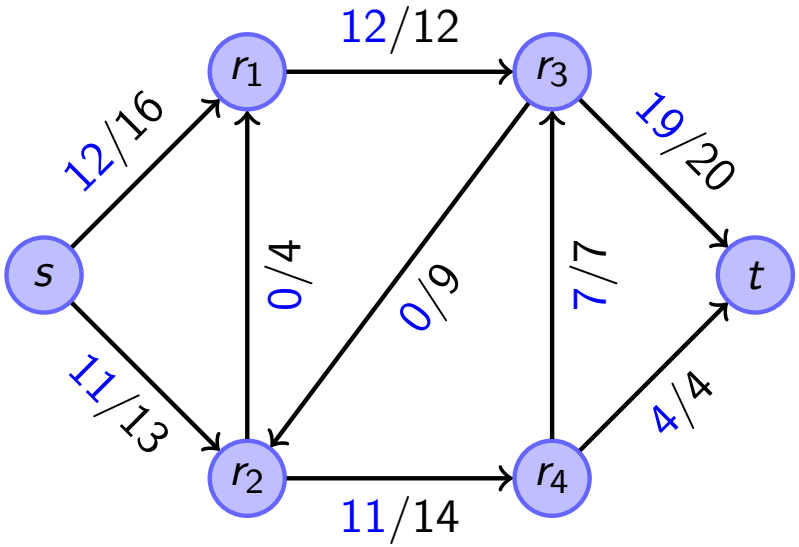


Graphe résiduel

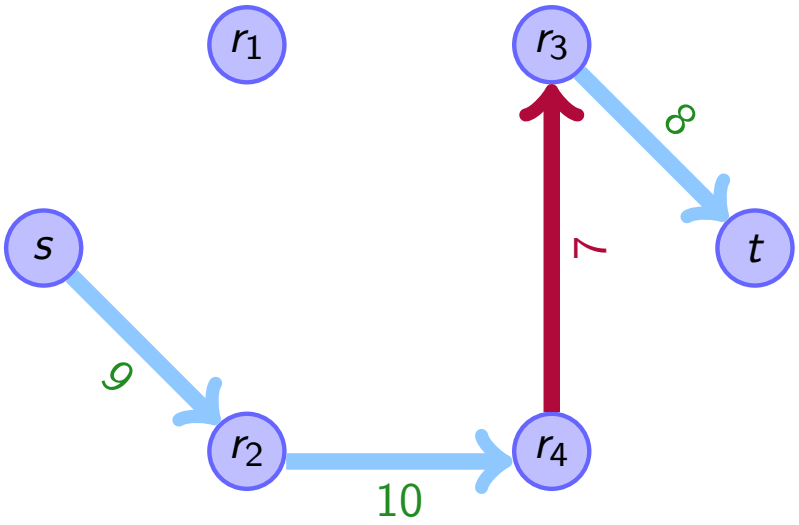


Algorithme de Ford-Fulkerson

Graphe de flot

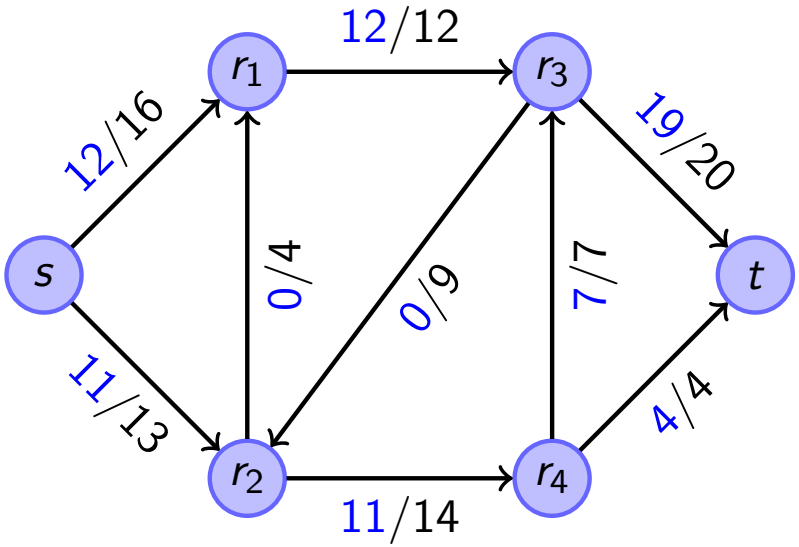


Graphe résiduel

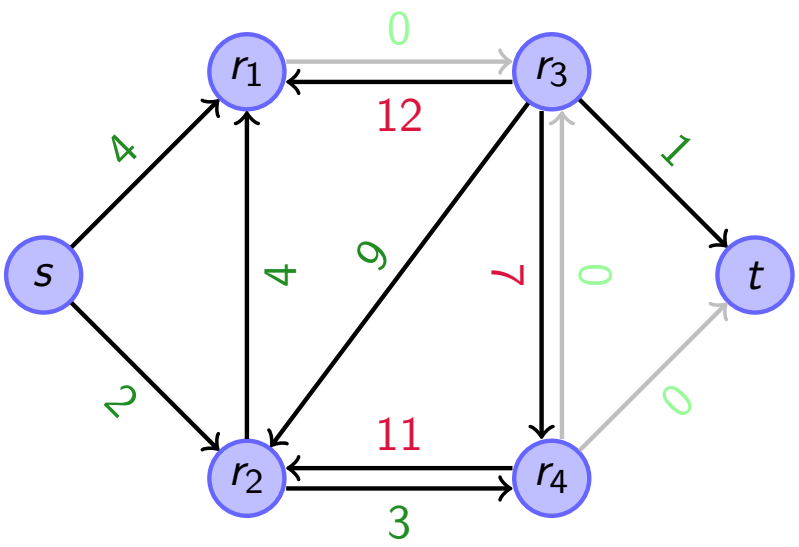


Algorithme de Ford-Fulkerson

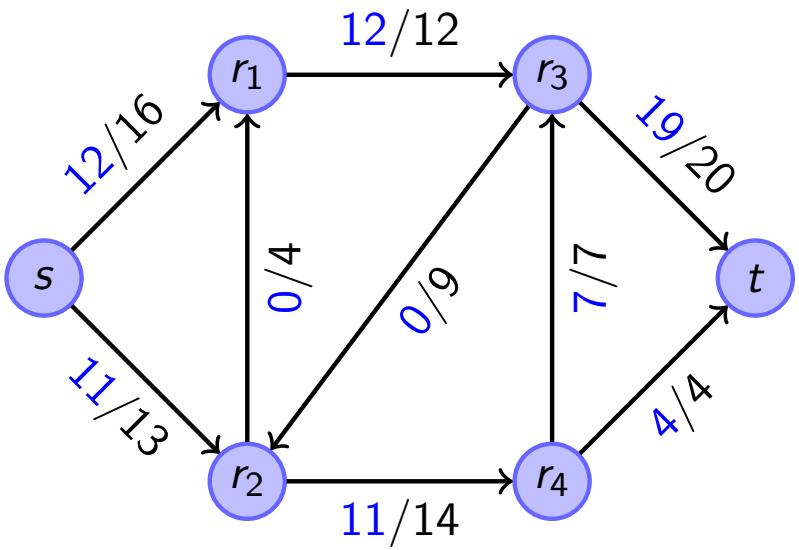
Graphe de flot



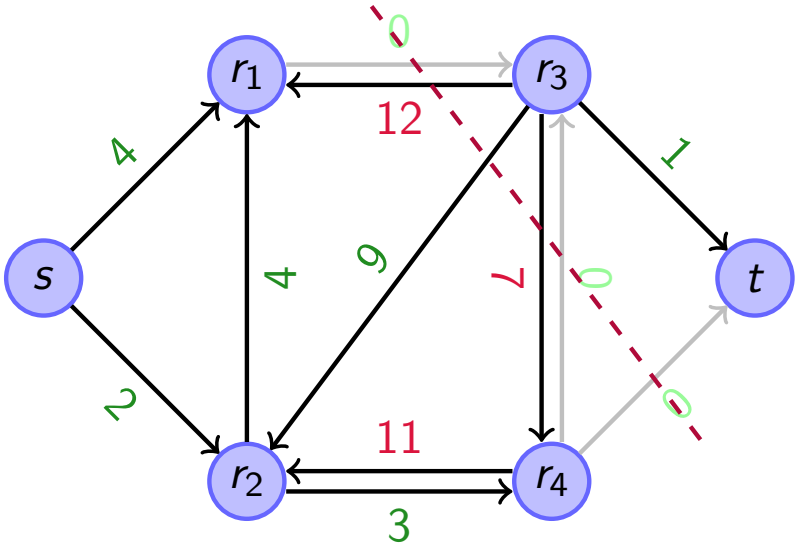
Graphe résiduel



Graphe de flot



Graphe résiduel



Terminaison

L'algorithme s'arrête lorsqu'il n'y a plus de chemin entre s et t dans le graphe résiduel

Coupe $s - t$

Partition des nœuds du graphe V en S et $T = V \setminus S$ telle que $s \in S$ et $t \in T$.

Sa capacité est donnée par :

$$c(S, T) = \sum_{(u,v) \in S \times T} c(u, v)$$

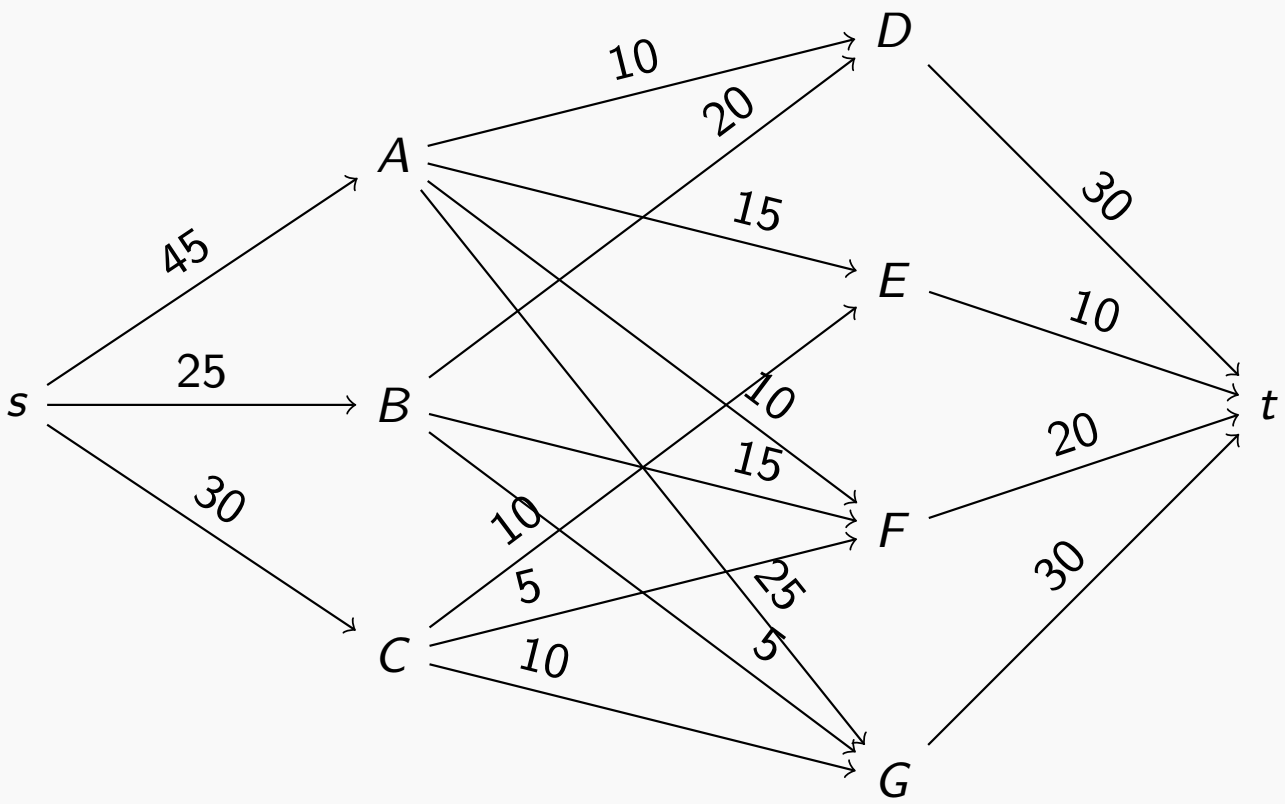
Théorème Max-flow Min-cut

Les trois propositions sont équivalentes :

1. Le flot φ entre s et t est maximal ;
2. Il n'existe aucun chemin augmentant ;
3. Il existe une coupe $s - t$ dont la capacité est égale à φ.

Les problèmes d'affectation ou de répartition de ressources peuvent très souvent être modélisés par des graphes de flots avec une composante bipartie.

Répartition de l'eau de puits entre des villes



Répartition de ressources

Pour les problèmes de répartition de ressources, on peut dégager une structure générale :

- Une composante bipartie au centre dont les arêtes modélisent les contraintes de répartition (e.g le débit max d'un tuyau entre un puits et une ville)
- Des arêtes entre s et les sources, modélisant les quantités de ressources allouables
- Des arêtes entre les destinations et t, modélisant les quantités cibles

Méthode générale de résolution d'un problème

Pour résoudre un problème se modélisant par un graphe de flot (souvent de répartition/affectation de ressources), on suit les étapes suivantes :

- Modéliser le problème par un graphe de flot
- Faire tourner l'algorithme de Ford-Fulkerson pour trouver le flot maximal
- Extraire la solution du problème du graphe de flots rempli

Conclusion de la section 6

Choses à savoir faire pour l'examen :

- Modéliser un problème par un graphe de flot
- Faire tourner Ford-Fulkerson à la main sur un exemple pour trouver la solution d'un problème
- Connaître et pouvoir raisonner sur les propriétés du flot

Comment s'entraîner :

- Refaire le TD 6 (exercices 1 et 2)

Des questions ?