République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
L'université Mohamed Boudiaf - M'Sila –

Faculté de mathématiques et d'informatique

2ème Année Master (M2RTIC)

Optimisation des Réseaux

Semestre : 03

2024/2025

Réalise par

Dr. DABBA ALI

➢ Contactez-nous

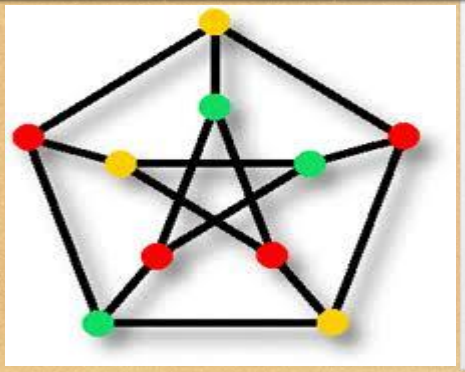
alidabba@gmail.com

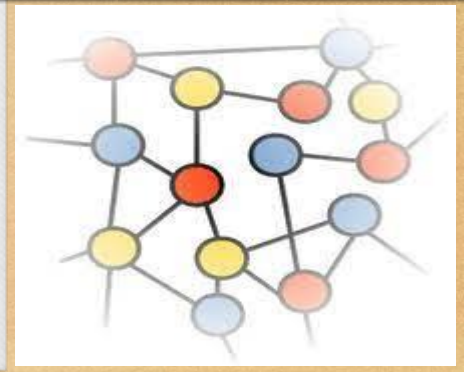
ali.dabba@univ-msila.dz

- En cas de problèmes ou de difficultés, me contacter ou contacter votre enseignant TD / TP
- Nous sommes à votre disposition pour vous aider

CHAPITRE II

Théorie des graphes et algorithmes fondamentaux





Plan

I. Introduction

- Pour résoudre de nombreux problèmes concrets, on est amené à tracer sur le papier des petits dessins qui représentent (partiellement) le problème à résoudre. Bien souvent, ces petits dessins se composent de points et de lignes continues reliant deux à deux certains de ces points. On appellera ces petits dessins des **graphes**, les points des **sommets** et les lignes des **arcs** ou **arêtes**, selon que la relation binaire sous-jacente est orientée ou non.

I. Introduction

- En résumé, les graphes constituent donc **une méthode de pensée** qui permet de **modéliser une grande variété de problèmes** en se ramenant à l'étude de **sommets et d'arcs**.
- Les derniers travaux en théorie des graphes sont souvent effectués par des **informaticiens**, du fait de l'importance qu'y revêt **l'aspect algorithmique**.

I. Introduction

Beaucoup de problèmes sur les graphes nécessitent que l'on **parcourt l'ensemble des sommets et des arcs/arêtes du graphe**. On étudie dans la suite les deux principales stratégies d'exploration :

- **le parcours en largeur** consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné ;
- **le parcours en profondeur** consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible (jusqu'à un cul-de-sac ou un cycle), puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

I. Introduction

- Dans la suite de ce chapitre, nous allons présenter les algorithmes de **Dijkstra** et **Ford-Bellman** qui résoudre le problème du **plus court chemin** (i.e., un chemin de poids minimal) d'un sommet u fixé à un sommet quelconque de **G**.
- A la fin, nous allons présenter les deux algorithmes **Kruskal** et **Prim** qui calculer **l'arbre couvrant minimal (ACM)** à partir d'un graphe.

II. Quelques exemples de modélisation par des graphes

- La notion de graphes et les problèmes liés aux graphes peuvent être rencontrés dans différentes situations de la vie réelle ainsi que dans des problèmes d'ingénierie notamment en informatique.
- Les graphes représentent, d'abord, **un moyen de modélisation**, ainsi qu'un moyen permettant de raisonner sur de nombreux problèmes.
- Les graphes permettent de modéliser des entités reliées par des liens. La disposition des entités et surtout des liens (ce que l'on appelle la topologie du graphe) permet d'induire plusieurs propriétés intéressantes. Pour illustrer cela, nous allons prendre quelques exemples introductifs.

II. Quelques exemples de modélisation par des graphes

Exemple 2.1 (modélisation d'un réseau d'amis)

Dans la vie courante, une personne est **amie avec plusieurs personnes** pouvant aussi être des amis à d'autres personnes, et ainsi de suite. Dans les réseaux sociaux par exemple, ces **relations** permettent d'établir des communautés et des règles s'appliquant au **partage des données**. Il est intéressant de noter (et ceci peut être montré par la théorie des graphes) que dans n'importe ensemble de groupes d'amis, il y a toujours au moins deux personnes ayant le même nombre d'amis.

II. Quelques exemples de modélisation par des graphes

Exemple 2.2 (modélisation d'un réseaux routier)

Le réseau routier d'un pays peut être représenté par un **graphe** dont **les sommets sont les villes**. Si l'on considère que toutes les routes sont à **double sens**, on utilisera un **graphe non orienté** et on reliera par une **arête** tout **couple de sommets** correspondant à deux villes reliées par une route (si l'on considère en revanche que certaines routes sont à **sens unique**, on utilisera **un graphe orienté)**. Ces arêtes pourront être valuées par la longueur des routes correspondantes.

II. Quelques exemples de modélisation par des graphes

Exemple 2.2 (modélisation d'un réseaux routier)

Etant donné un tel graphe, on pourra s'intéresser, par exemple, à la résolution des problèmes suivants :

➔ Quel est le plus court chemin, en nombre de kilomètres, passant par un certain nombre de villes données ?

➔ Quel est le chemin traversant le moins de villes pour aller d'une ville à une autre ?

➔ Est-il possible de passer par toutes les villes sans passer deux fois par une même route ?

II. Quelques exemples de modélisation par des graphes

Exemple 2.3 (modélisation d'un réseau de télécommunication)

Un réseau informatique est constitué d'un **ensemble de machines** (des ordinateurs, des hubs, des switches, des routeurs, des répétiteurs, etc.) et **des liaisons physiques** (câbles métalliques, fibres optiques, ondes radio, etc.). Un routeur permet d'acheminer un paquet vers la bonne sortie afin que ce dernier puisse retrouver son chemin vers la destination. On est notamment intéressée par **connaître le chemin optimal** dans le réseau (en termes de temps de transmission, énergie utilisée, nombre de sauts, etc.) ainsi que le débit maximal du réseau, ce qui permet d'éviter des situations agaçantes comme la congestion du réseau.

III. Différentes notions de graphes

- D'une manière **informelle**, on peut définir un graphe par une représentation figurative où l'on retrouve un ensemble de points (appelés nœuds ou sommets) reliés par un ensemble de courbes droites ou non.
- Ces liens représentent généralement une **relation ou une dépendance entre les sommets**. Ils sont appelés des **arêtes** si la relation est symétrique (on parle de graphe non-orienté), ou des **arcs** sinon (on parle de graphe orienté).

III. Différentes notions de graphes

Définition 2.1 (Graphe non orienté)

Un graphe non orienté **G** est la donnée d'un couple **G = (S, A)** tel que :

- **S** est un ensemble fini de sommets,
- **A** est un ensemble de couples non ordonnés de sommets $\{s_i , s_j\} \in S^2$.

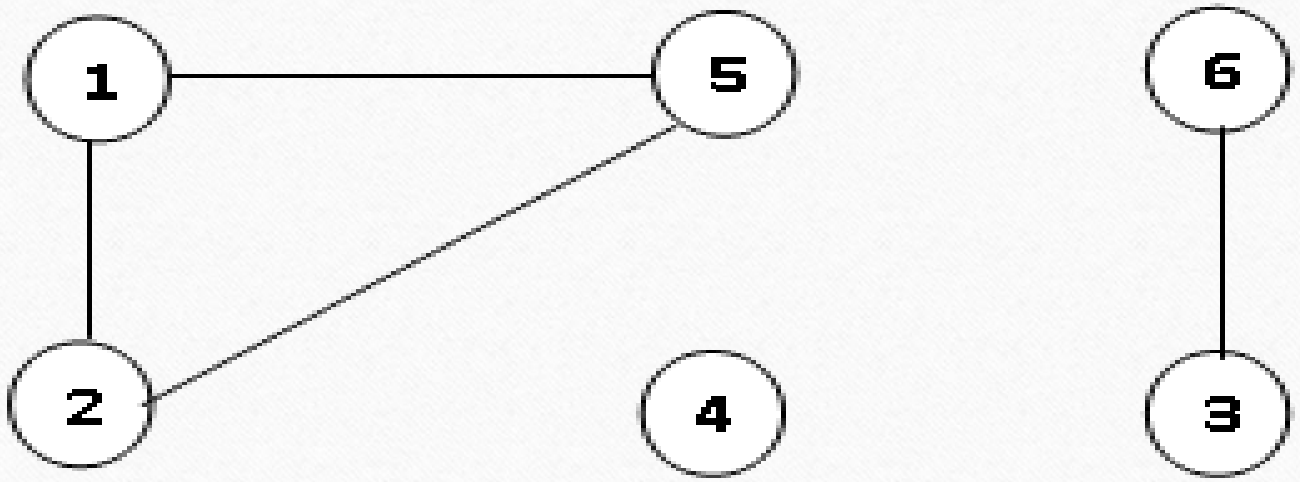
Une paire $\{s_i , s_j\}$ est appelée une **arête**, et est représentée graphiquement par $s_i - s_j$. On dit que les sommets s_i et s_j sont adjacents. L'ensemble des sommets adjacents au sommet $s_i \in S$ est noté $Adj(s_i) = \{s_i \in S, \ \{s_i , s_j\} \in A\}$.

III. Différentes notions de graphes

Exemple

Le graphe ci-contre représente un graphe non orienté

G = (S, A) avec **S = {1, 2, 3, 4, 5, 6}** et A = {{1, 2}, {1, 5}, {5, 2}, {3, 6}}



III. Différentes notions de graphes

Définition 2.2 (Boucle, simple graphe, multi-graphe, ordre, taille)

➔ Une **boucle** est une arête reliant un sommet à lui-même.

➔ Un graphe non-orienté est dit **simple** s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non-orienté qui n'est pas simple est un **multi-graphe**. Dans le cas d'un multi-graphe, A n'est plus un ensemble mais un multi-ensemble d'arêtes.

➔ On appelle **ordre d'un graphe** le nombre de ses sommets, i.e. c'est **card(S) ou |S|**.

➔ On appelle **taille d'un graphe** le nombre de ses arêtes, i.e. c'est **card(A) ou |A|**.

III. Différentes notions de graphes

Définition 2.3 (Graphe orienté)

Un graphe orienté **G** est la donnée d'un couple **G = (S, A)** tel que :

➔ **S** est un ensemble fini de sommets,

➔ **A** est un ensemble de couples ordonnés de sommets $(s_i , s_j) \in S^2$.

Un couple (s_i , s_j) est appelé un **arc**, et est représenté graphiquement par $s_i \rightarrow s_j$, s_i est le sommet initial ou origine, et s_j le sommet terminal ou extrémité. L'arc $a = (s_i , s_j)$ est dit **sortant** en s_i et **incident** en s_j, et s_j est un **successeur** de s_i, tandis que s_i est un **prédécesseur** de s_j.

➔ L'ensemble des successeurs d'un sommet $s_i \in S$ est noté $Succ(s_i) = \{s_j \in S, (s_i , s_j) \in A\}$.

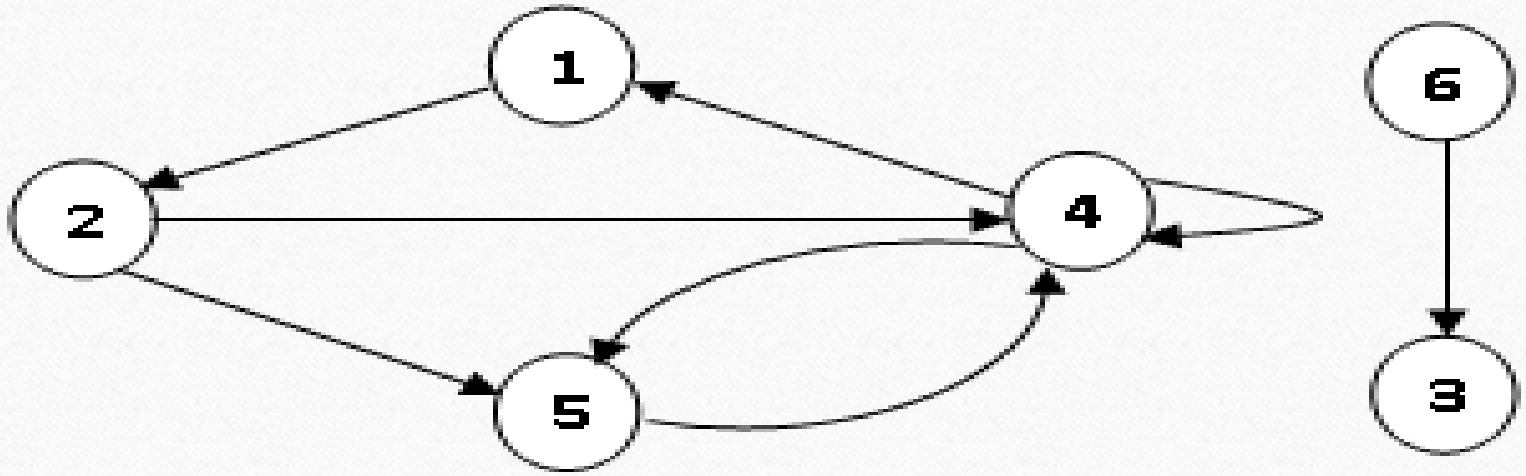
➔ L'ensemble des prédécesseurs d'un sommet $s_i \in S$ est noté $Pred(s_i) = \{s_j \in S, (s_j , s_i) \in A\}$.

III. Différentes notions de graphes

Exemple

Le graphe ci-contre représente un graphe orienté

G = (S, A) avec **S = {1, 2, 3, 4, 5, 6}** et **A = {(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)}.**



III. Différentes notions de graphes

Définition 2.4 (Boucle, élémentaire, p-graphe)

➔ Une **boucle** est une arc reliant un sommet à lui-même.

➔ Un graphe orienté est dit **élémentaire** s'il ne contient pas de boucle.

➔ Un graphe orienté est un **p-graphe** s'il comporte au plus **p** arcs entre deux sommets. Le plus souvent, on étudiera des **1-graphes**.

IV. Arbres et Arborescences

Définition 2.5 (arbre)

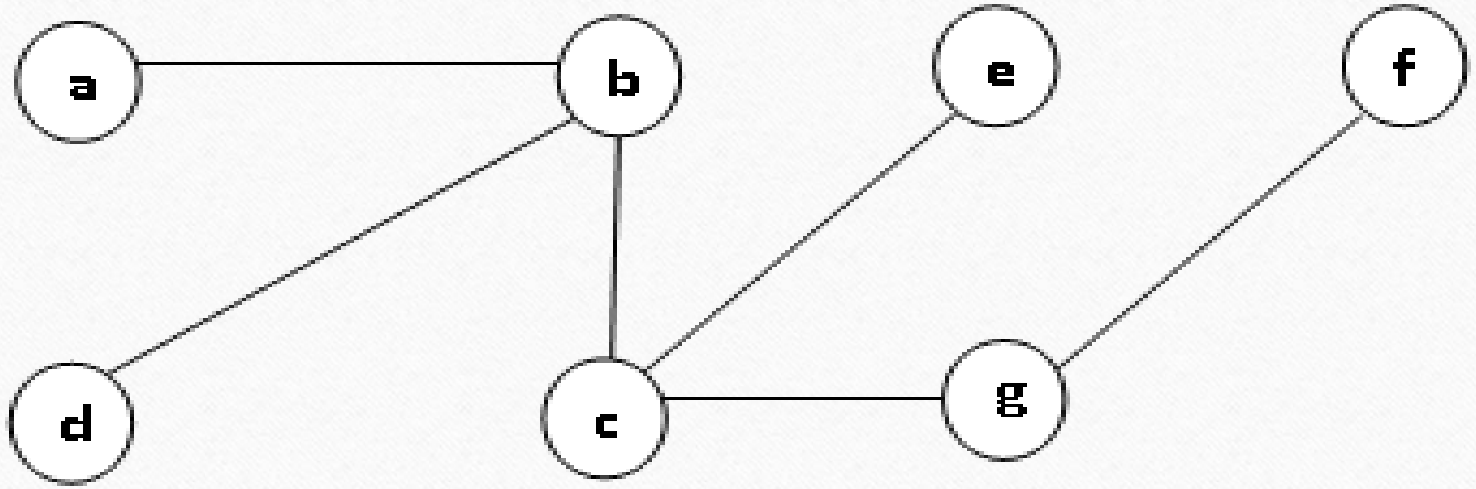
Un **arbre** est un graphe non orienté **G** qui vérifie une des conditions équivalentes suivantes :

- ➔ **G** est connexe et acyclique
- ➔ **G** est sans cycle et possède **n - 1** arêtes
- ➔ **G** est connexe et admet **n - 1** arêtes
- ➔ **G** est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
- ➔ **G** est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
- ➔ Il existe une chaîne et une seule entre **2** sommets quelconques de **G**.

IV. Arbres et Arborescences

Exemple

Le graphe suivant est un arbre



IV. Arbres et Arborescences

Définition 2.6 (forêt, arborescence)

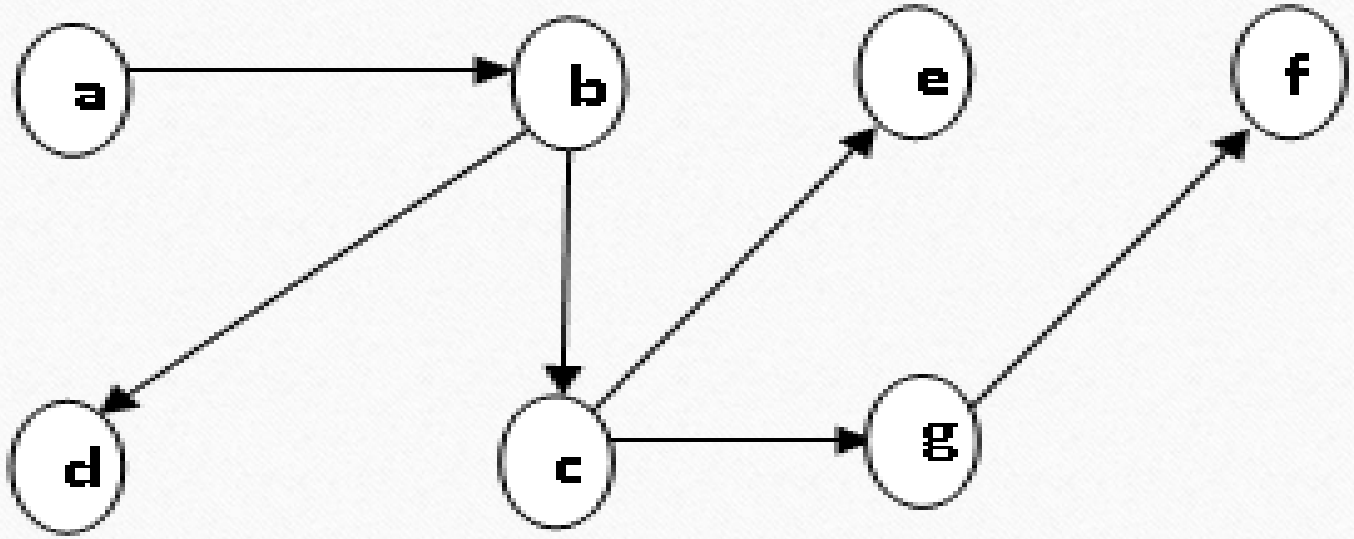
➔ On appelle **forêt** un graphe dont chaque composante connexe est un arbre.

➔ Une **arborescence** est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i. Si l'arborescence comporte **n** sommets, alors elle comporte exactement **n - 1** arcs.

IV. Arbres et Arborescences

Exemple

Le graphe suivant est une arborescence de racine **a** :



II. Parcours de graphes

Définition 2.7 (parcours d'un graphe)

Soit **G = (S, A)** un graphe et **x ∈ S** un sommet, un parcours du graphe **G** à partir de **x** est une **visite de chaque sommet accessible depuis x**.

Le résultat d'un parcours est un ensemble de chemins partants de **x** allant vers les sommets accessibles depuis **x**.

Un parcours peut être représenté par sous-graphe de **G** qui est **un arbre de racine x** (ou une forêt si certains sommets de **G** ne sont pas accessibles depuis **x**).

II. Parcours de graphes

- D'un point de vue algorithmique, un parcours correspond à la procédure suivante :
- A chaque étape de la boucle **Tant que** un seul sommet **y** est traité qui engendre le début du traitement d'un ou plusieurs autre sommets . . .
- On peut définir un **ordre de parcours** en numérotant à un endroit précis de la boucle **Tant que** le sommet **y** en cours de traitement.

II. Parcours de graphes

Procédure parcours(G, x)
    L = liste des sommets à traiter (vide au départ)
    mettre x dans L (début du traitement de x)
    Tant que L ≠ ∅ **Faire**
        sortir le **1er** sommet y de L (y en cours de traitement)
        V = successeurs non traités de y
        Pour tout $z \in V$ **Faire**
            mettre z dans L (début du traitement de z)
        Fin Pour
        fin du traitement de y
    Fin Tant que
Fin

II. Parcours de graphes

Qu'est-ce qu'un parcours de graphe (orienté ou non)?

- Visite de tous les sommets accessibles depuis un sommet de départ donné

II. Parcours de graphes

Comment parcourir un graphe?

- Marquage des sommets par des couleurs :
  - Blanc = Sommet pas encore visité
  - Gris = Sommet en cours d'exploitation
  - Noir = Sommet que l'on a fini d'exploiter
- Au début, le sommet de **départ est gris** et tous les **autres** sont **blancs**
- A chaque étape, un sommet gris est sélectionné
  - Si tous ses voisins sont déjà gris ou noirs, alors il est colorié en noir
  - Sinon, il colorie un (ou plusieurs) de ses voisins blancs en gris
    - Jusqu'à ce que **tous les sommets** soient **noirs ou blancs**

II. Parcours de graphes

Mise en œuvre : Stockage des sommets gris dans une structure

- Si on utilise une file (FIFO), alors **parcours en largeur**
- Si on utilise une pile (LIFO), alors **parcours en profondeur**

II. Parcours de graphes

Spécification d'un algorithme de parcours

1 **Fonction** *Parcours*(g, s_0)
  Entrée : Un graphe g et un sommet s_0 de g
  Sortie : Arborescence π du parcours de g à partir de s_0

Arborescence associée à un parcours :

- s_i est le père de s_j si c'est s_i qui a colorié s_j en gris
- s_i est racine si $s_i = s_0$ ou si pas de chemin de s_0 jusque s_i

II. Parcours de graphes

Quelle structure de données pour représenter l'arborescence?

Tableau π tel que $\pi[s_i]$ = *null* si s_i est racine, et $\pi[s_i]$ = père de s_i sinon

Exemple d'arborescence associée à un parcours au départ de *a* :

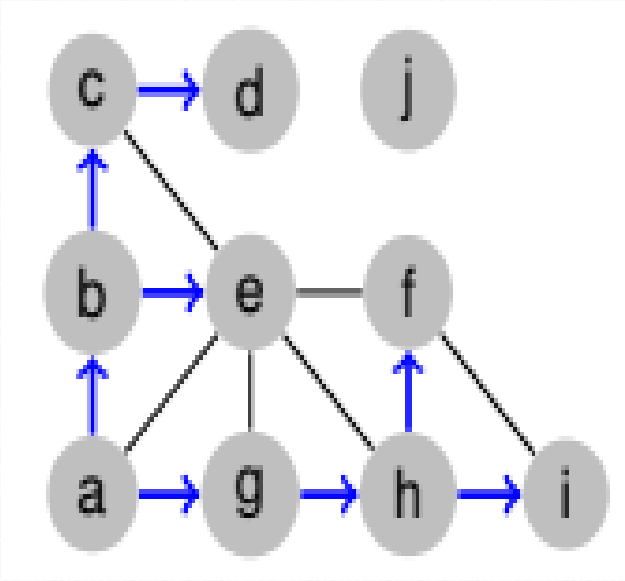


Tableau π correspondant :

-	a	b	c	b	h	a	g	h	-
a	b	c	d	e	f	g	h	i	j

Algorithme 2 : Parcours en largeur d'un graphe

1 : **Fonction** *BFS(g, s_0)*
Entrée : Un graphe **g** et un sommet s_0 de **g**
Postcondition : Retourne une arborescence π d'un parcours en largeur de **g** à partir de s_0
Déclaration : Une file (FIFO) **f** initialisée à vide
2 : **Pour** chaque sommet s_i de *g* **Faire**
3 : $\pi[s_i] \leftarrow null$
4 : Colorier s_i en blanc
5 : **Fin Pour**
6 : Ajouter s_0 dans la file **f** et colorier s_0 en gris
7 : **Tant que** la file **f** n'est pas vide **Faire**
8 : Soit $s_{FirstOut}$ le sommet le plus ancien dans **f**
9 : **Pour** tout sommet $s_i \in succ(s_{FirstOut})$ **Faire**
10 : **Si** s_i est blanc **Alors**
11 : Ajouter s_i dans la file **f** et colorier s_i en gris
12 : $\pi[s_i] \leftarrow s_{FirstOut}$
13 : **Fin Si**
14 : **Fin Pour**
15 : Enlever $s_{FirstOut}$ de **f** et colorier $s_{FirstOut}$ en noir
16 : **Fin Tant que**
17 : Retourne π
18 : **Fin**

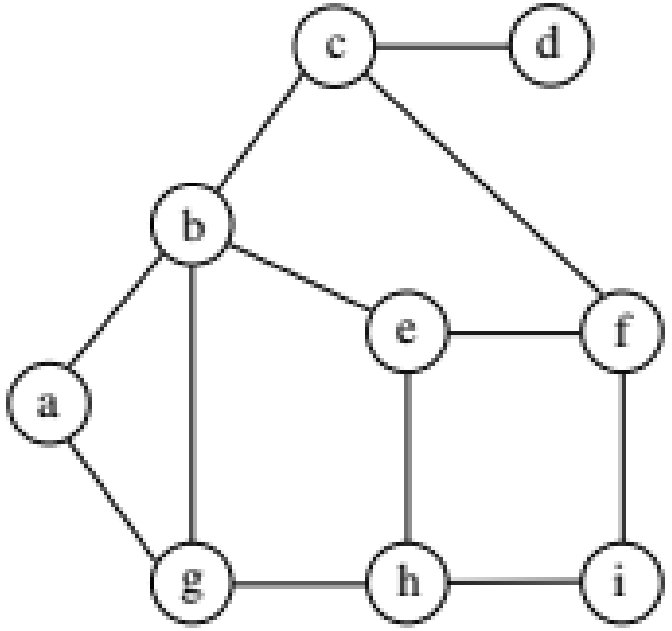
II. Parcours de graphes

Complexité. Soient **n** et **p** le nombre de sommets et arcs de **g**, respectivement. Chaque sommet accessible depuis $\mathbf{s_0}$ est mis au plus une fois dans la file **f**. En effet, seuls les sommets blancs entrent dans la file, et un sommet blanc est colorié en gris quand il entre dans la file, puis en noir quand il en sort, et ne pourra jamais redevenir blanc. À chaque passage dans la boucle lignes **7** à **16**, il y a exactement un sommet qui est enlevé de la file. Cette boucle sera donc exécutée au plus **n** fois. À chaque fois qu'un sommet est enlevé de la file, la boucle lignes **9** à **14** parcourt tous les successeurs du sommet enlevé, de sorte que les lignes 10 à 13 seront exécutées au plus une fois pour chaque arc. Par conséquent, la complexité de **l'algorithme 2 (BFS)** est :

- $\mathbf{O(n^2)}$ dans le cas d'une implémentation par matrice d'adjacence
- $\mathbf{O(n + p)}$ dans le cas d'une implémentation par listes d'adjacence.

Parcours en largeur (Breadth First Search / BFS)

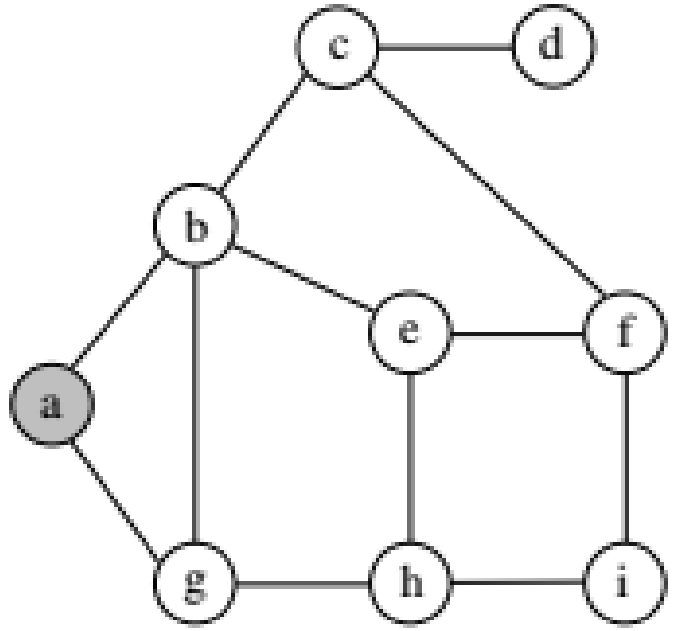
```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



$f = <>$

Parcours en largeur (Breadth First Search / BFS)

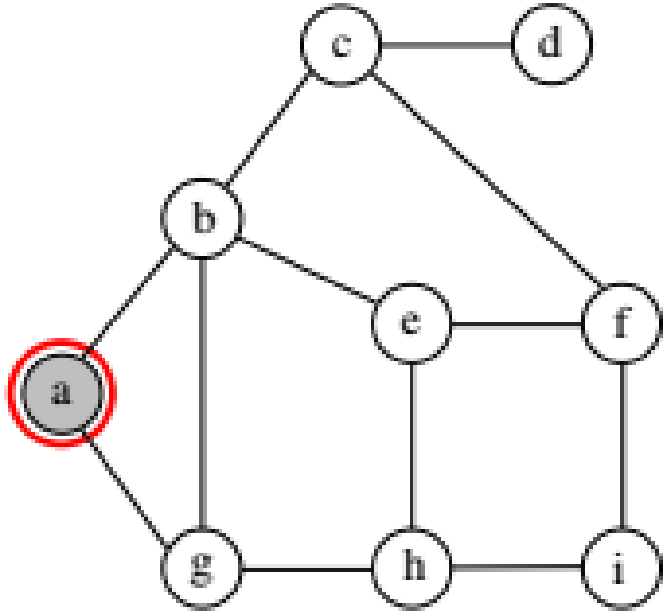
```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



$f =< a >$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

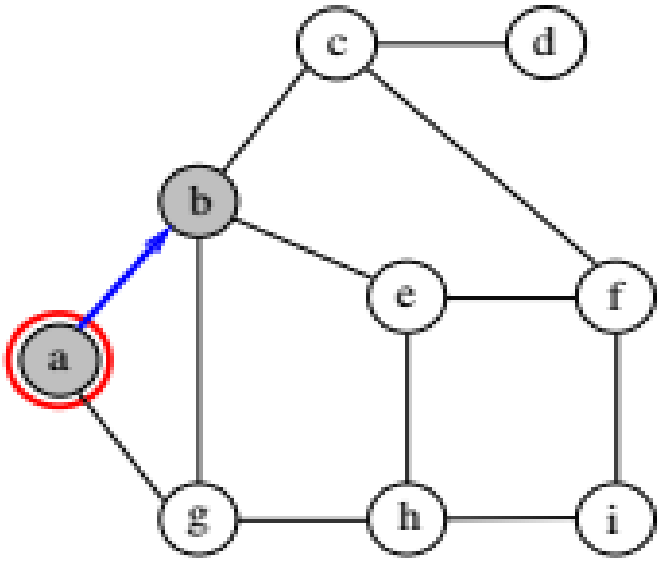


$f =< a >$

$s_k = a$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



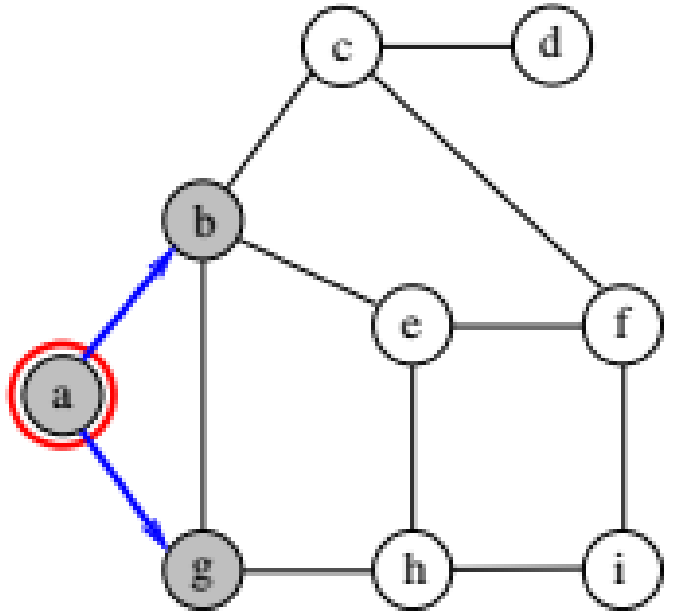
$f =< b, a >$

$s_k = a, s_i = b$

Complexité de BFS pour un graphe ayant n sommets et p arcs ?

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

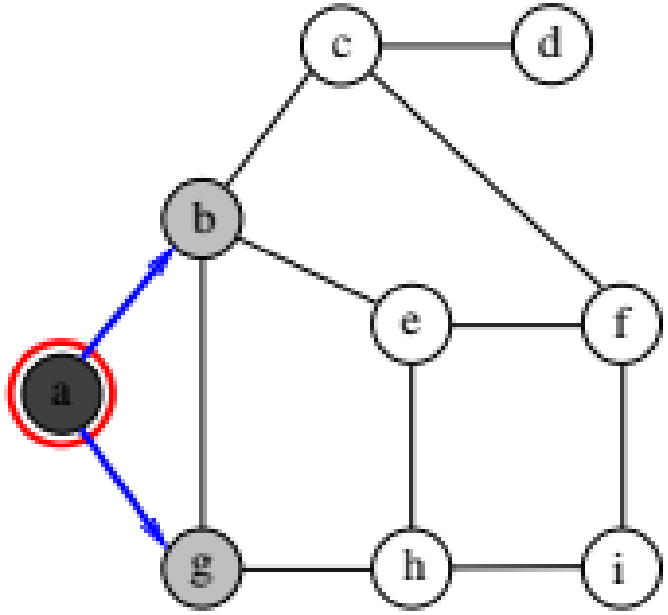


$f =< g, b, a >$

$s_k = a, s_i = g$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

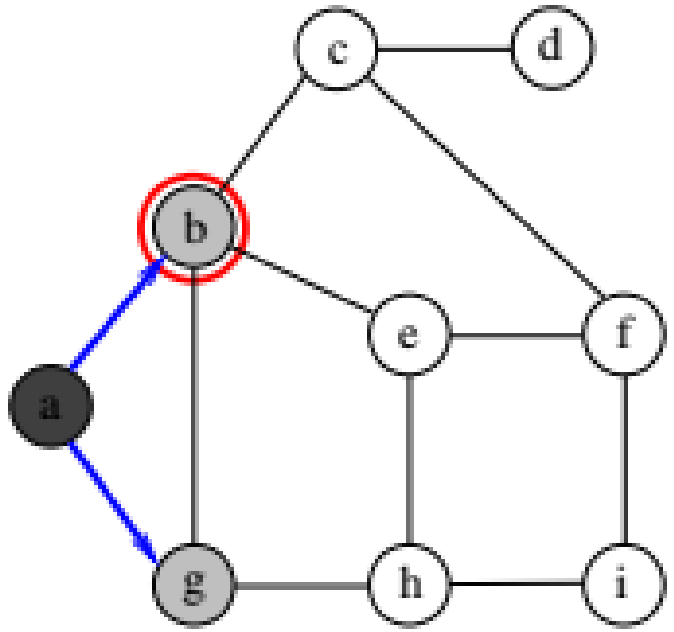


$f =< g, b >$

$s_k = a$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

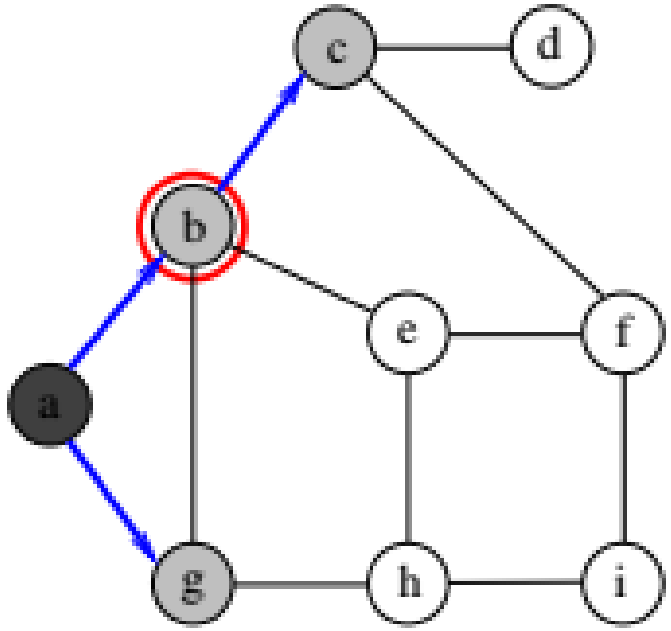


$f =< g, b >$

$s_k = b$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

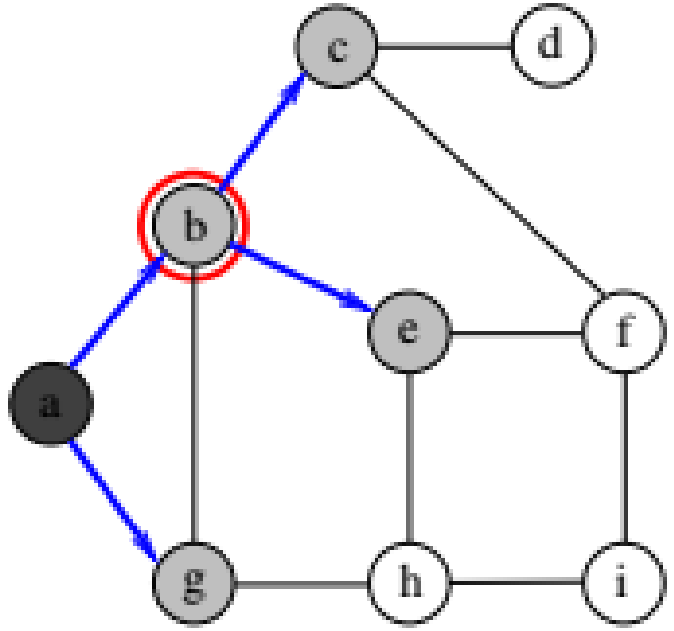


$f =< c, g, b >$

$s_k = b, s_i = c$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



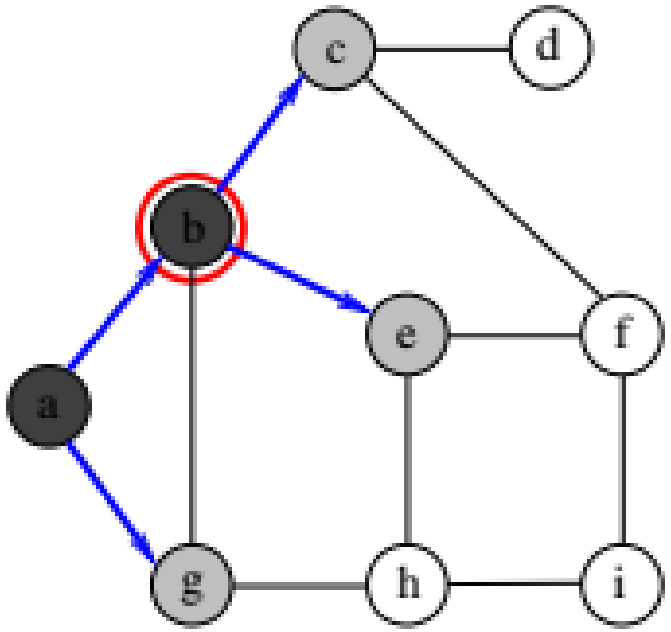
$f =< e, c, g, b >$

$s_k = b,\ s_i = e$

Complexité de BFS pour un graphe ayant n sommets et p arcs ?

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



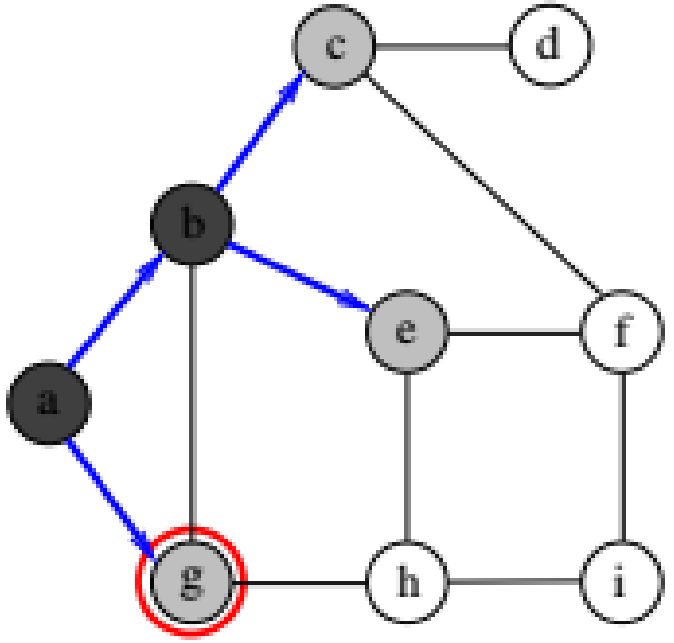
$f =< e, c, g >$

$s_k = b$

Complexité de BFS pour un graphe ayant n sommets et a arcs ?

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



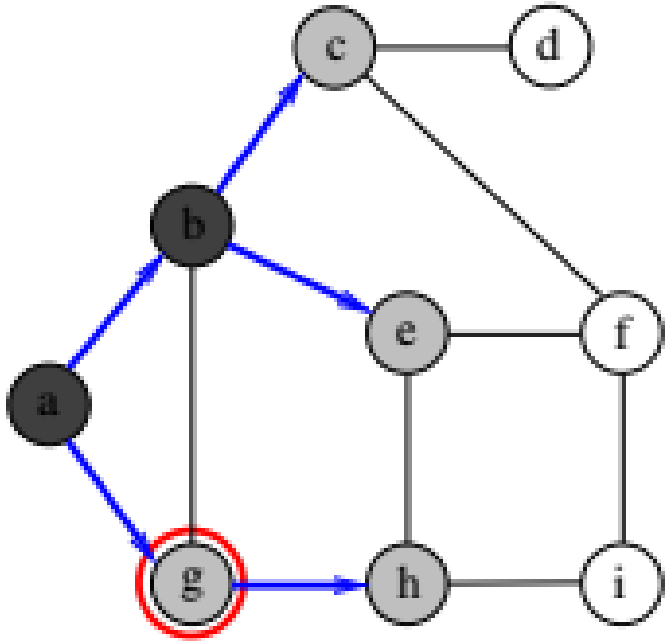
$f =< e, c, g >$

$s_k = g$

Complexité de BFS pour un graphe ayant n sommets et p arcs ?

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

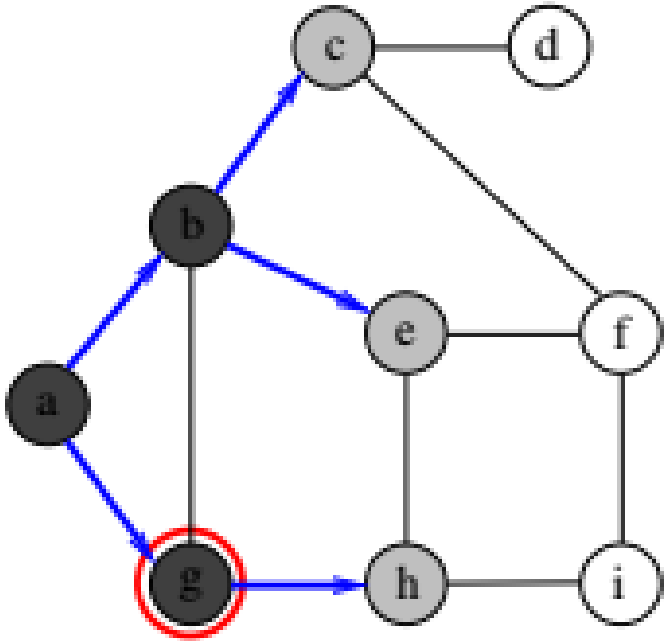


$f =< h, e, c, g >$

$s_k = g,\ s_i = h$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

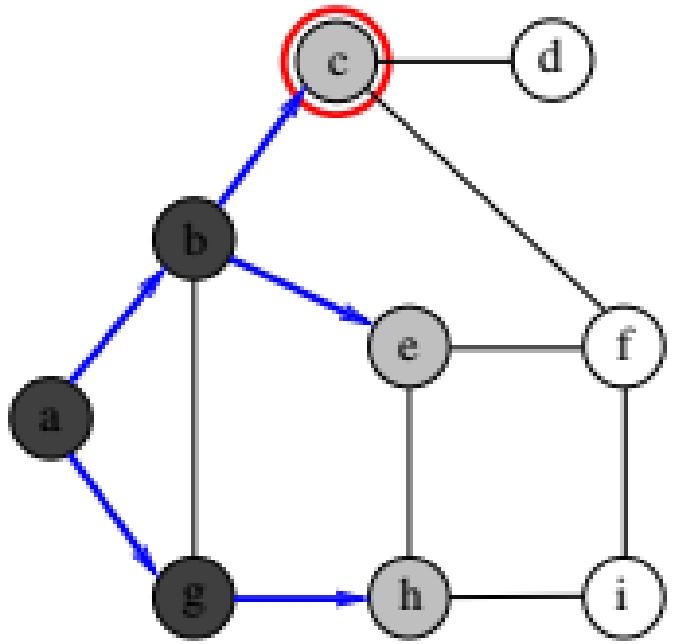


$f =< h, e, c >$

$s_k = g$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

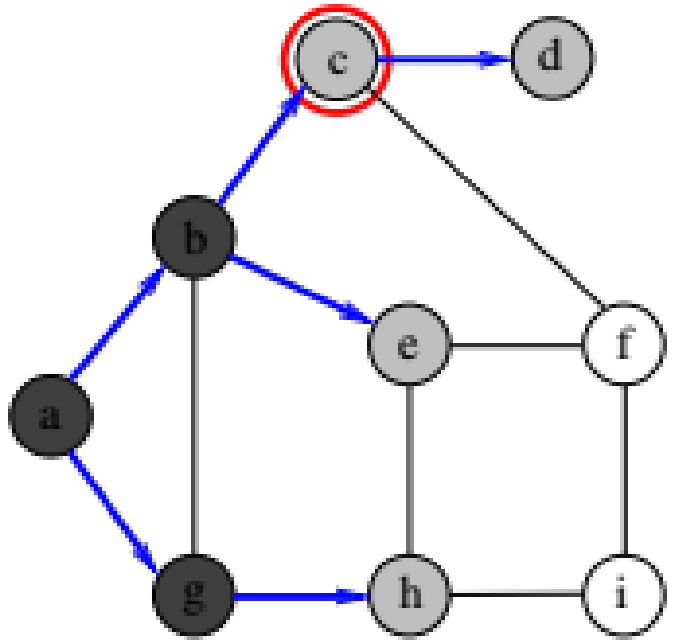


$f =< h, e, c >$

$s_k = c$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

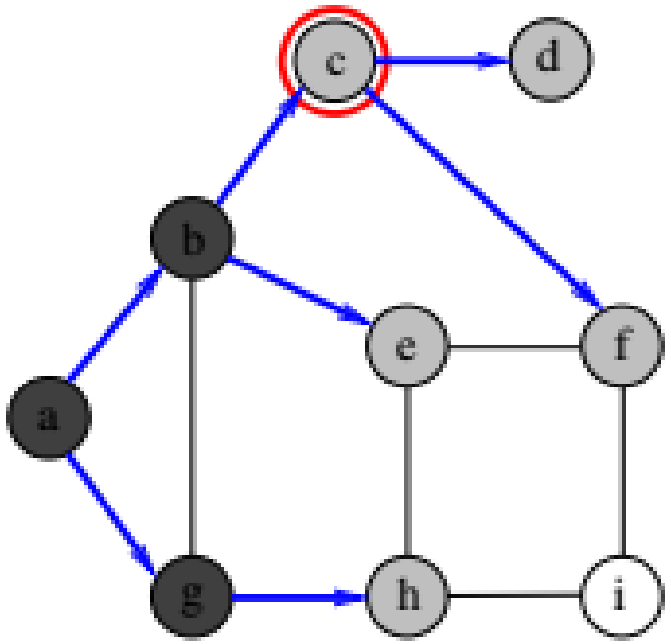


$f =< d, h, e, c >$

$s_k = c,\ s_i = d$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

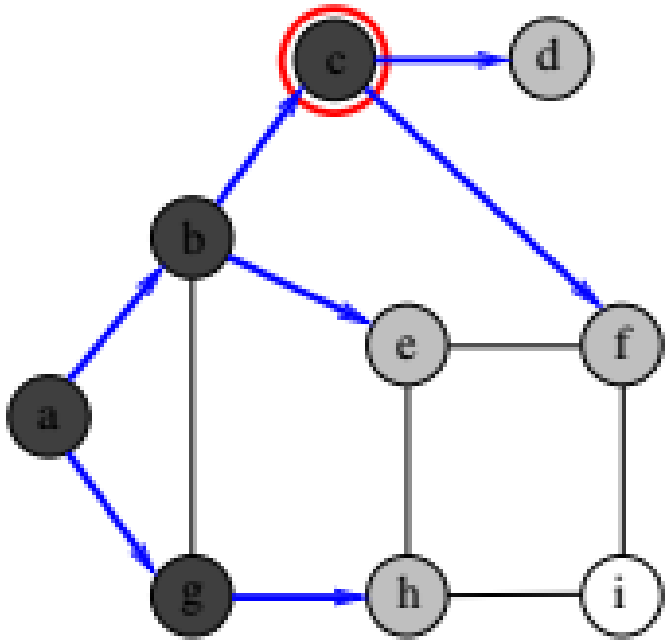


$f =< f, d, h, e, c >$

$s_k = c,\ s_i = f$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

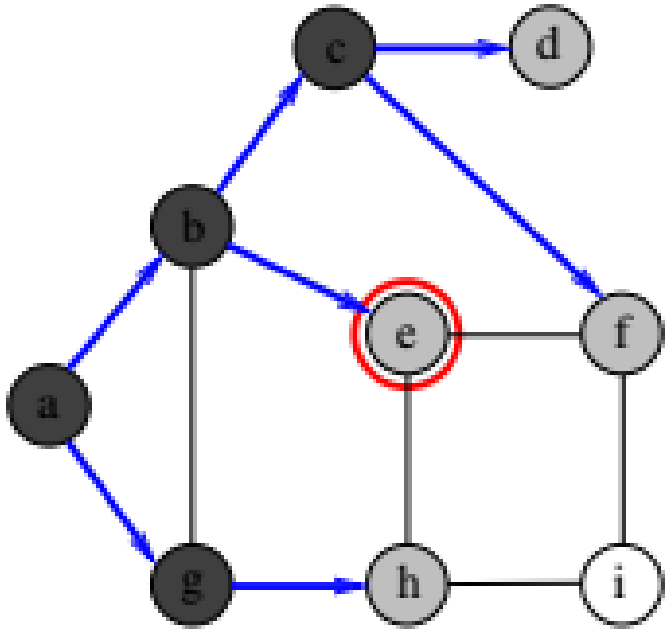


$f =< f, d, h, e >$

$s_k = c$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

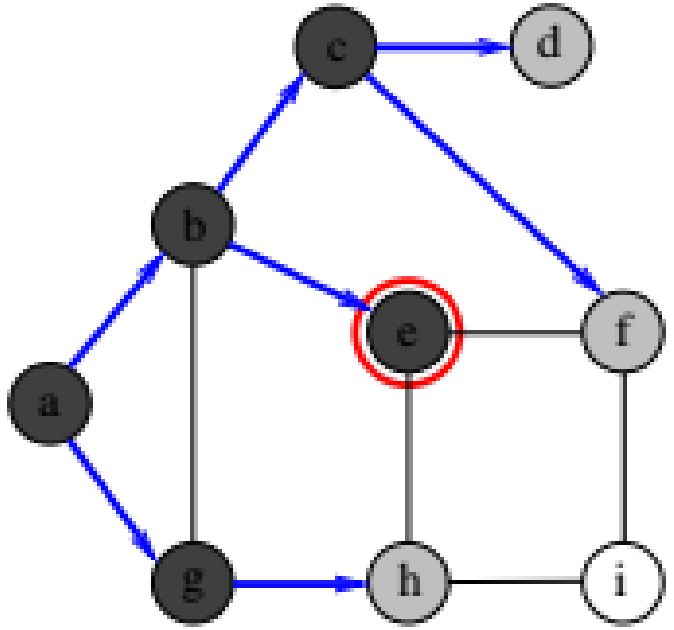


$f =< f, d, h, e >$

$s_k = e$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

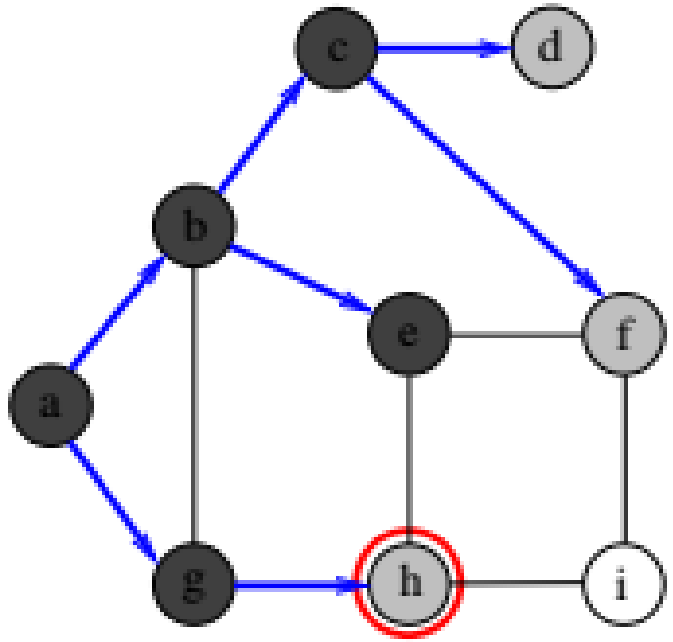


$f =< f, d, h >$

$s_k = e$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

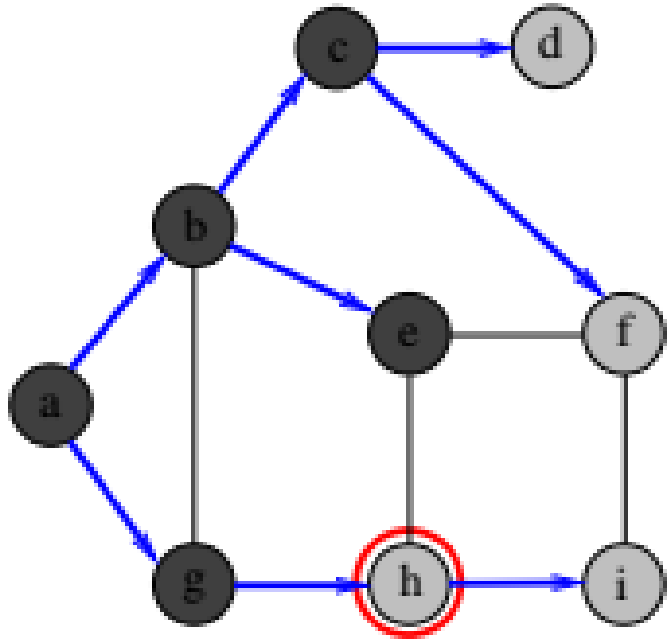


$f =< f, d, h >$

$s_k = h$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

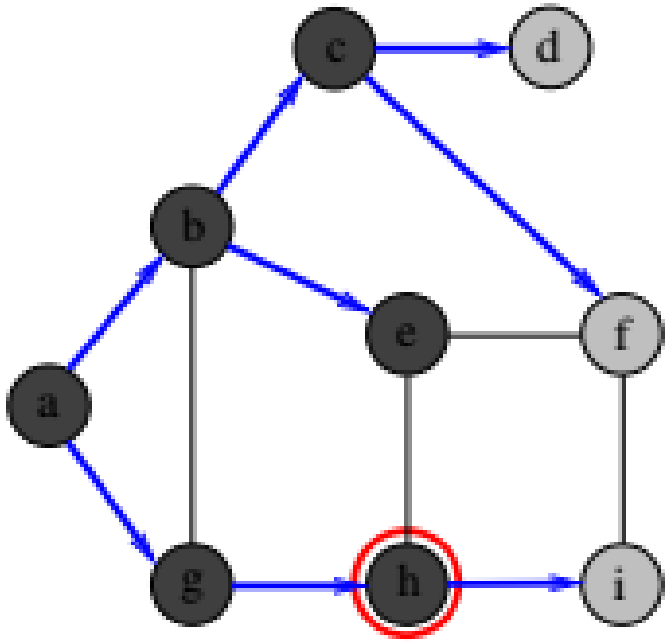


$f =< i, f, d, h >$

$s_k = h, s_i = i$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

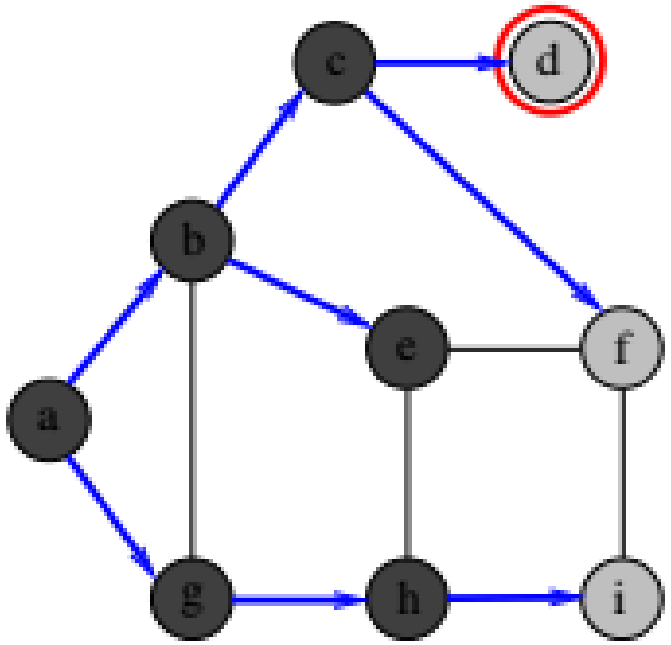


$f =< i, f, d >$

$s_k = h$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

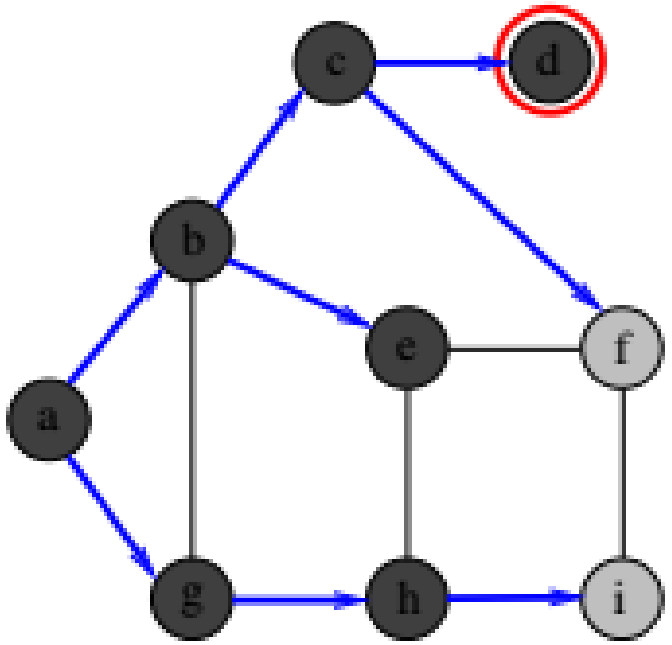


$f =< i, f, d >$

$s_k = d$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

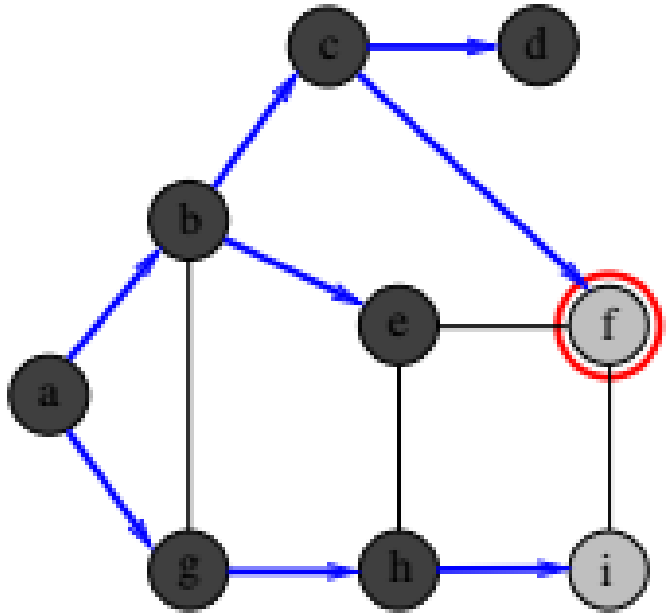


$f =< i, f >$

$s_k = d$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

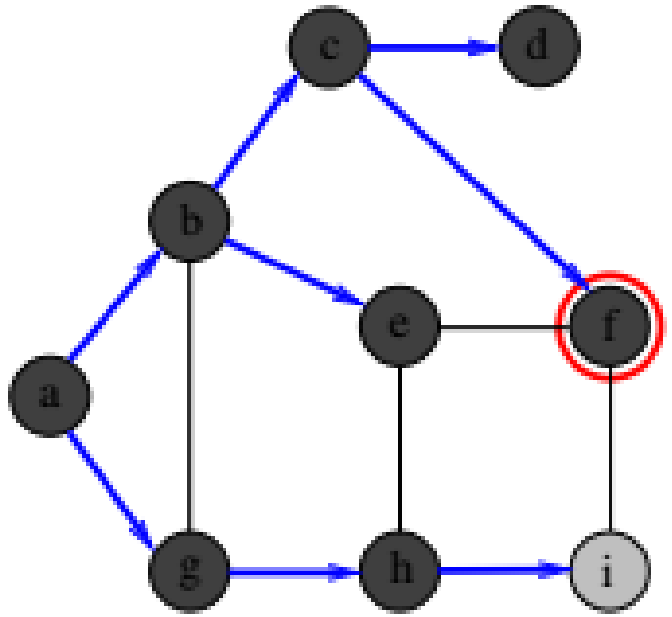


$f =< i, f >$

$s_k = f$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

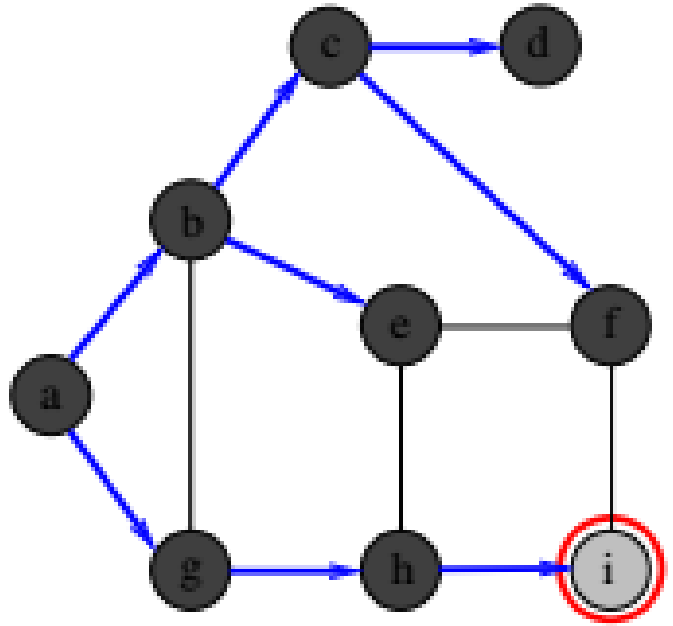


$f =< i >$

$s_k = f$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

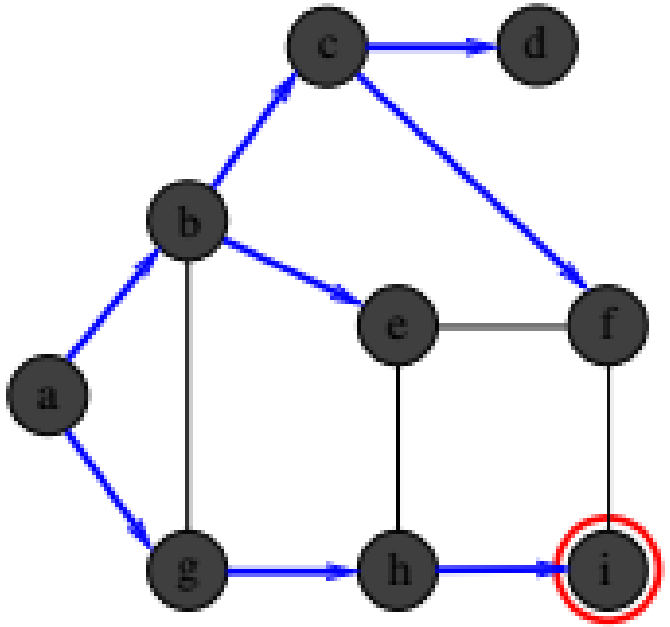


$f =< i >$

$s_k = i$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

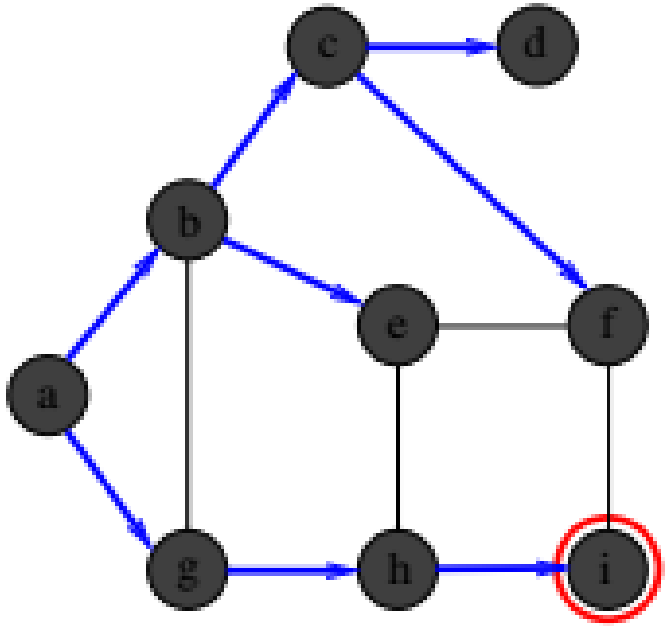


$f = <>$

$s_k = i$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```

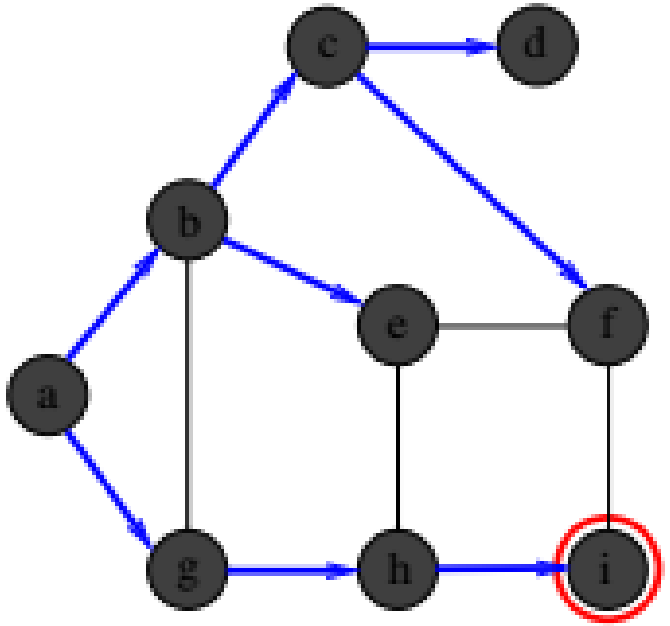


$f = <>$

$s_k = i$

Parcours en largeur (Breadth First Search / BFS)

```
1  Fonction BFS(g, s0)
2      Soit f une file (FIFO) initialisée à vide
3      pour chaque sommet si de g faire
4          π[si] ← null
5          Colorier si en blanc
6      Ajouter s0 dans f et colorier s0 en gris
7      tant que f n'est pas vide faire
8          Soit sk le sommet le plus ancien dans f
9          pour chaque si ∈ succ(sk) tq si est blanc faire
10             Ajouter si dans f et colorier si en gris
11             π[si] ← sk
12         Enlever sk de f et colorier sk en noir
13     retourne π
```



$f = <>$

$s_k = i$

Algorithme 3 : Parcours en profondeur d'un graphe

1 : **Fonction** *DFS(g, s_0)*
Entrée : Un graphe **g** et un sommet s_0 de **g**
Postcondition : Retourne une arborescence π d'un parcours en profondeur de **g** à partir de s_0
Déclaration : Une pile (LIFO) **p** initialisée à vide
2 : **Pour** chaque sommet s_i de *g* **Faire**
3 : $\pi[s_i] \leftarrow null$
4 : Colorier s_i en blanc
5 : **Fin Pour**
6 : Empiler s_0 dans la pile **p** et colorier s_0 en gris
7 : **Tant que** la pile **p** n'est pas vide **Faire**
8 : Soit s_i le dernier sommet entré dans **p** (au sommet de **p**)
9 : **Si** $\exists\ s_j \in succ(s_i)$ tel que s_j soit blanc **Alors**
10 : Empiler s_j dans la pile **p** et colorier s_j en gris
11 : $\pi[s_j] \leftarrow s_i$
12 : **Sinon**
13 : Dépiler s_i de **p** et colorier s_i en noir
14 : **Fin Si**
15 : **Fin Tant que**
16 : **Retourne** π
17 : **Fin**

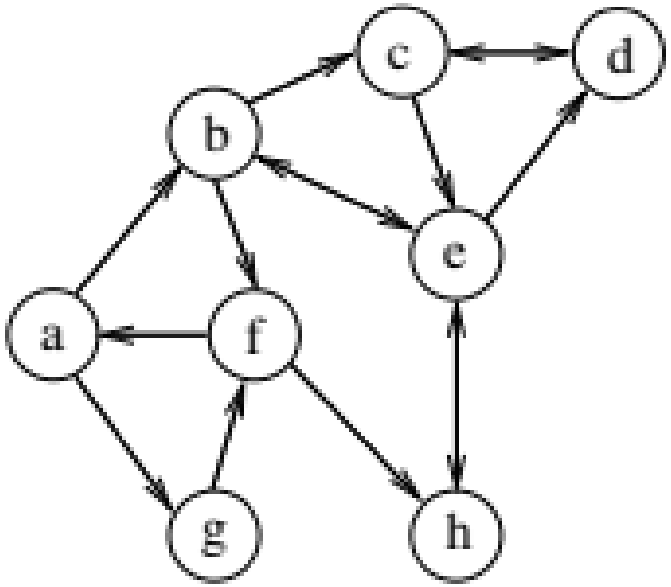
II. Parcours de graphes

Complexité : Chaque sommet accessible depuis $\mathbf{s_0}$ est mis, puis enlevé, exactement une fois dans la pile, comme dans BFS, et à chaque passage dans la boucle lignes **7** à **15**, soit un sommet est empilé (si $\mathbf{s_i}$ a encore un successeur blanc), soit un sommet est dépilé (si $\mathbf{s_i}$ n'a plus de successeur blanc). Par conséquent, l'algorithme passera au plus **2n** fois dans la boucle lignes **7** à **15**. À chaque passage, il faut parcourir la liste des successeurs de $\mathbf{s_i}$ pour chercher un successeur blanc. Si nous utilisons un itérateur qui mémorise pour chaque sommet de la pile le dernier successeur de ce sommet qui était blanc. Par conséquent, si le graphe contient **n** sommets (accessibles à partir de $\mathbf{s_0}$) et **p** arcs/arêtes, alors la complexité de **l'algorithme 3 (DFS)** est :

- $\mathbf{O(n^2)}$ dans le cas d'une implémentation par matrice d'adjacence,
- $\mathbf{O(n + p)}$ dans le cas d'une implémentation par listes d'adjacence.

Parcours en profondeur (Depth First Search = DFS)

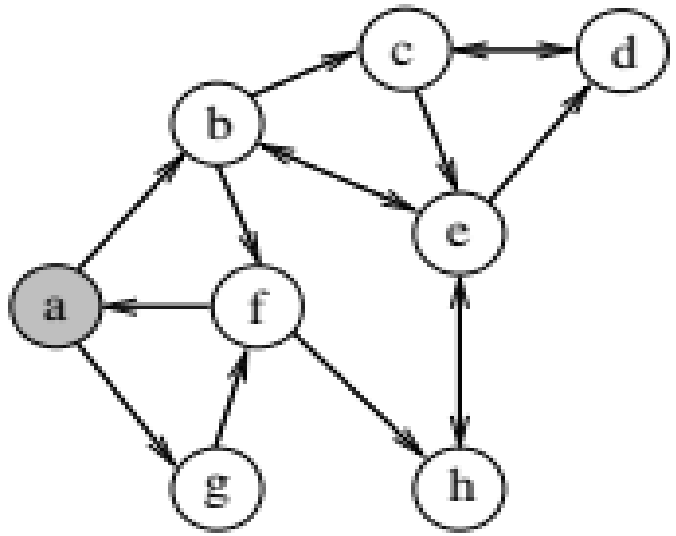
```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p = <>$

Parcours en profondeur (Depth First Search = DFS)

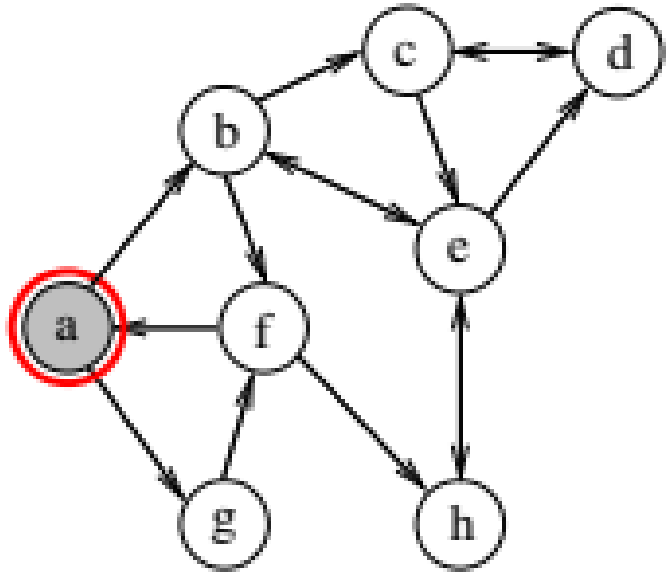
```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p =< a >$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

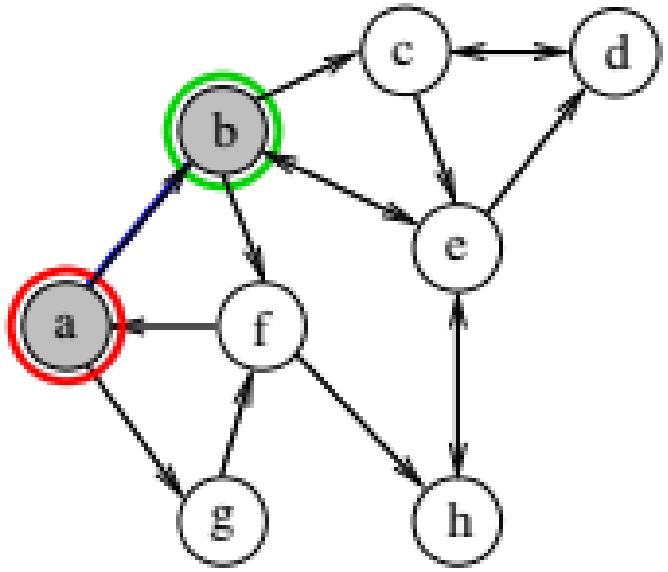


$p =< a >$

$s_i = a$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

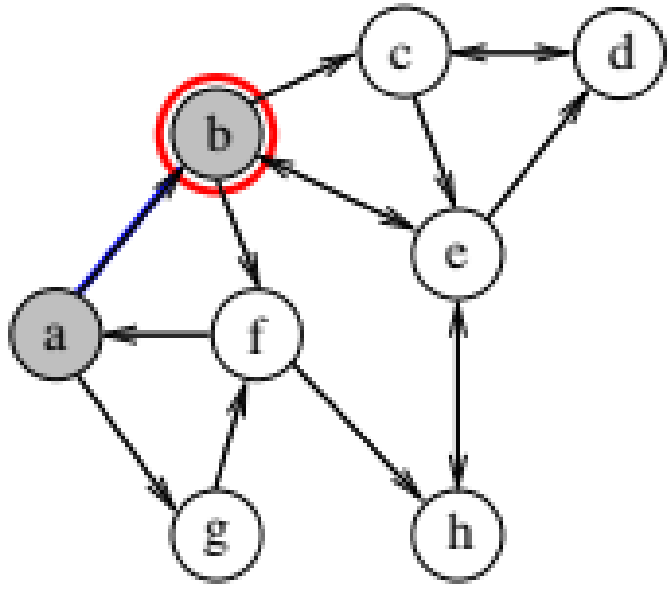


$p =< a, b >$

$s_i = a, s_j = b$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

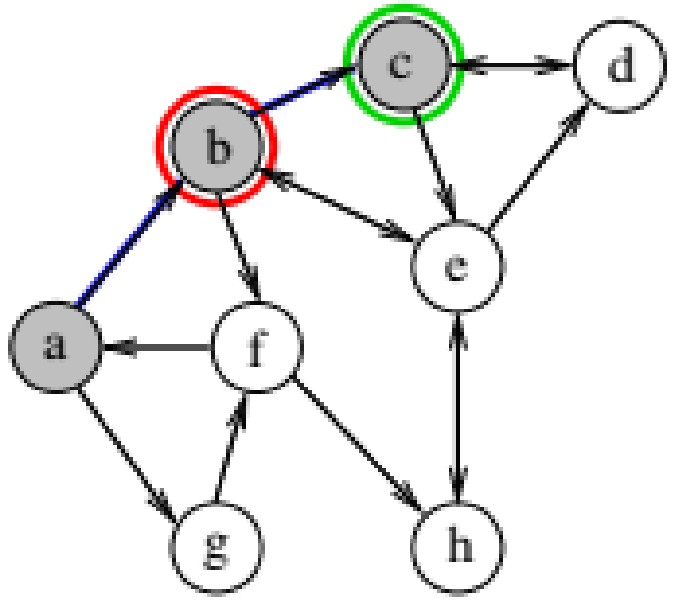


$p =< a, b >$

$s_i = b$

Parcours en profondeur (Depth First Search = DFS)

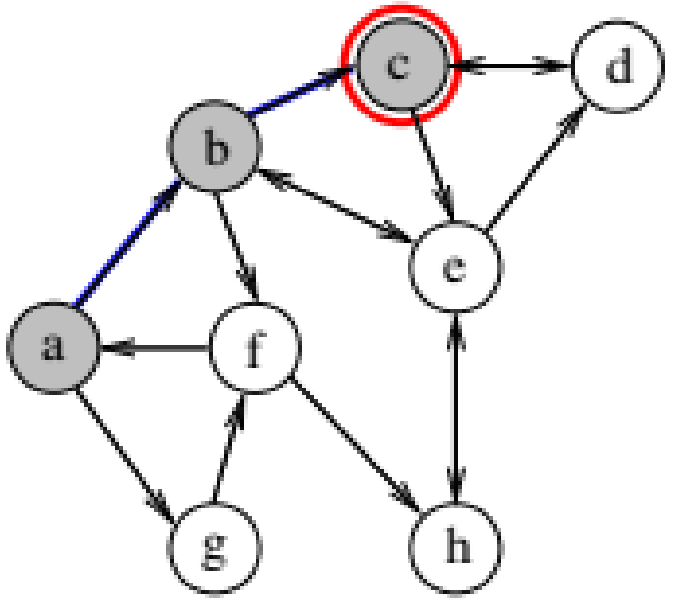
```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p =< a, b, c >$
$s_i = b, s_j = c$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

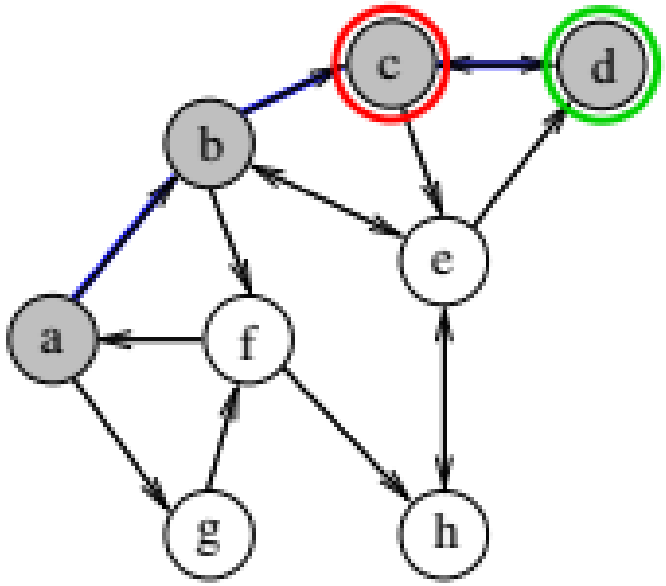


$p =< a, b, c >$

$s_i = c$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

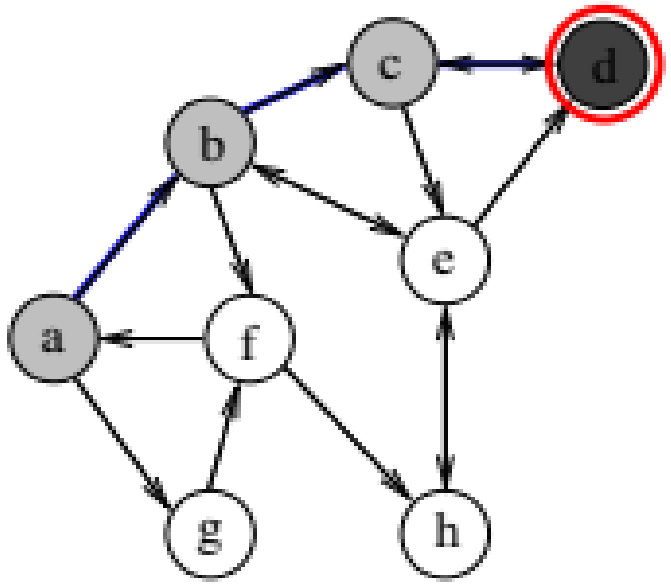


$p =< a, b, c, d >$

$s_i = c, s_j = d$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

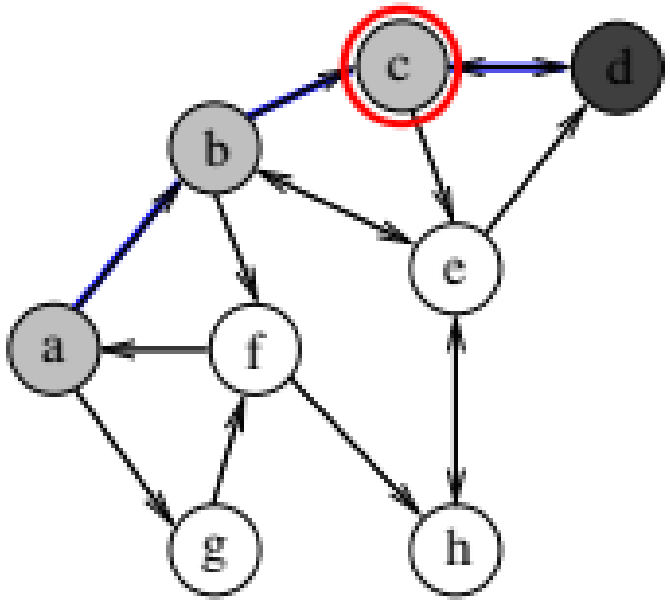


$p =< a, b, c >$

$s_i = d$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

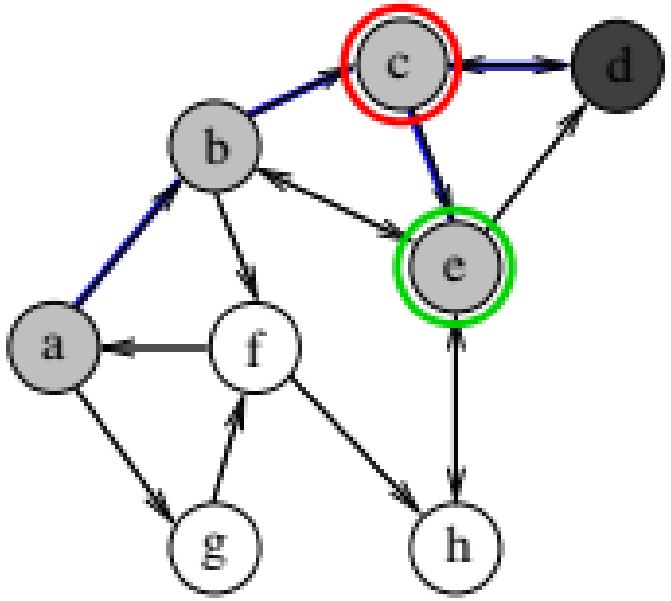


$p =< a, b, c >$

$s_i = c$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

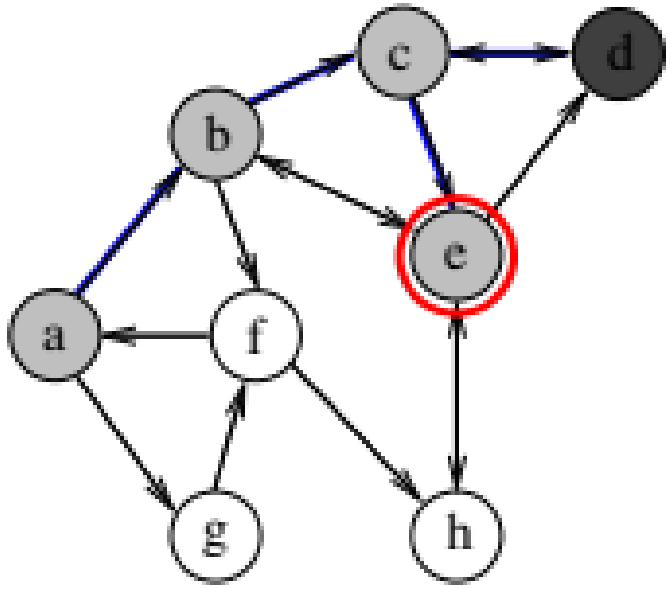


$p =< a, b, c, e >$

$s_i = c,\ s_j = e$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

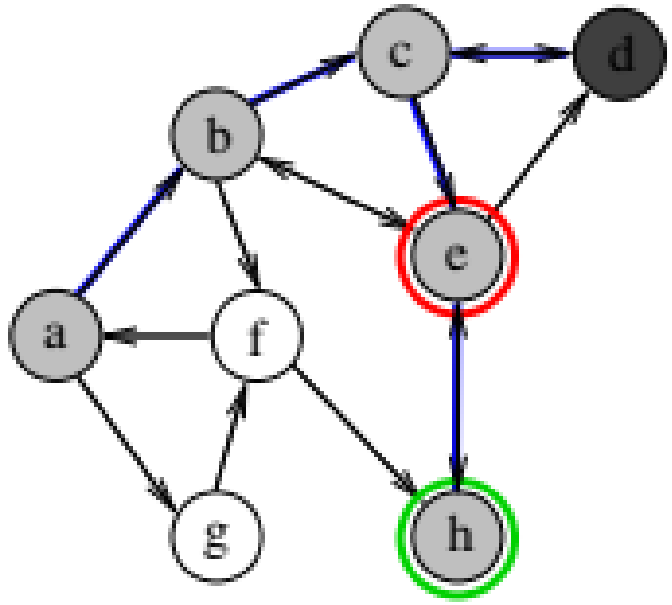


$p =< a, b, c, e >$

$s_i = e$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

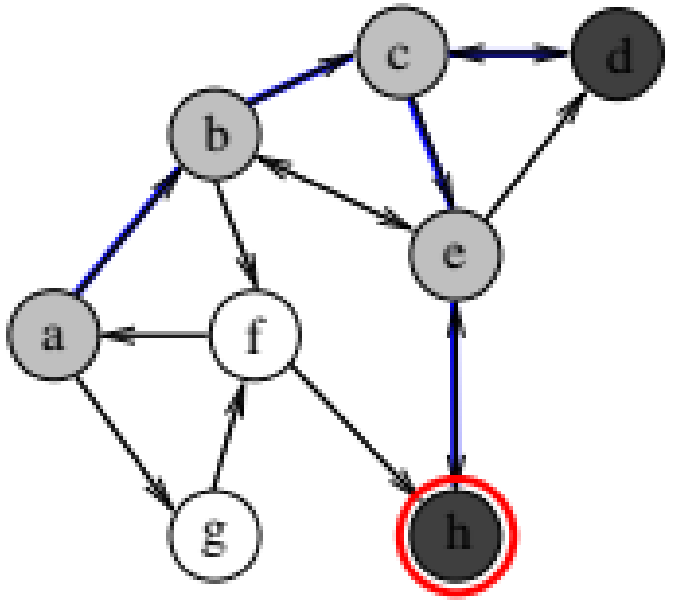


$p =< a, b, c, e, h >$

$s_i = e, s_j = h$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

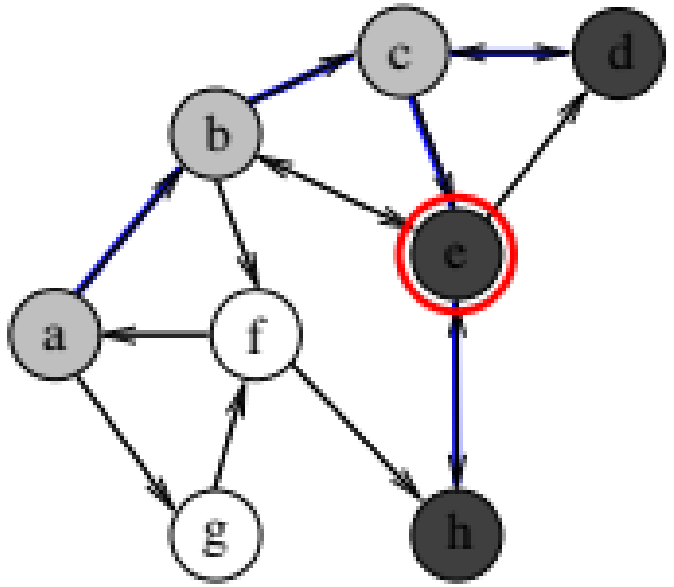


$p =< a, b, c, e >$

$s_i = h$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

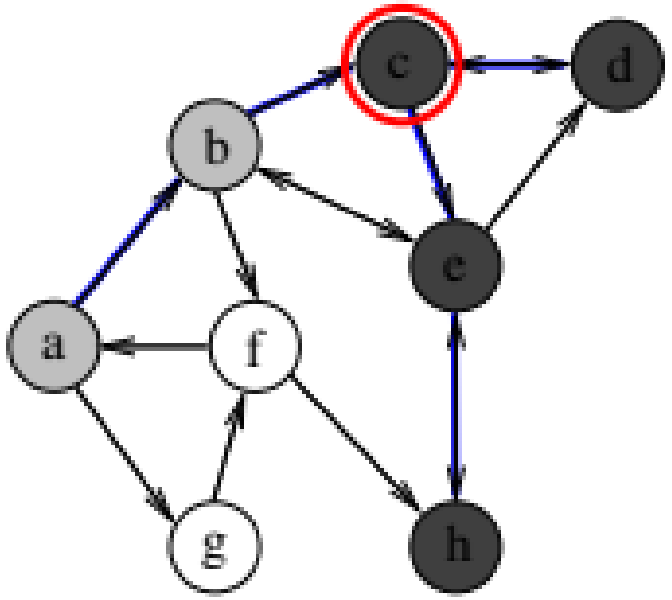


$p =< a, b, c >$

$s_i = e$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

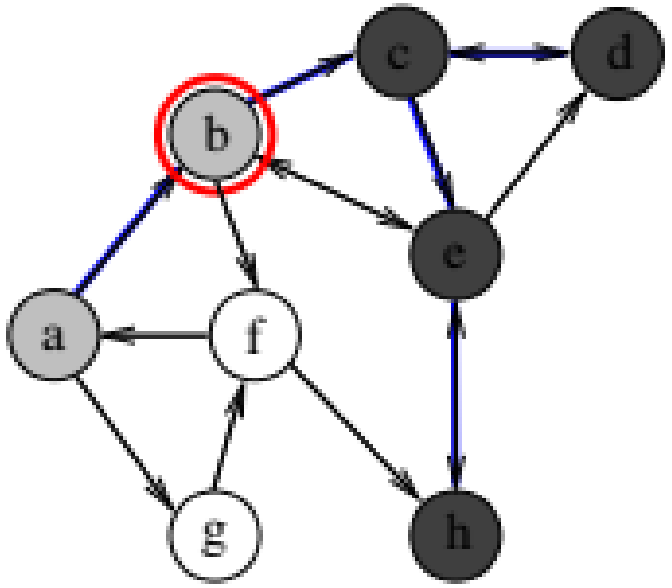


$p =< a, b >$

$s_i = c$

Parcours en profondeur (Depth First Search = DFS)

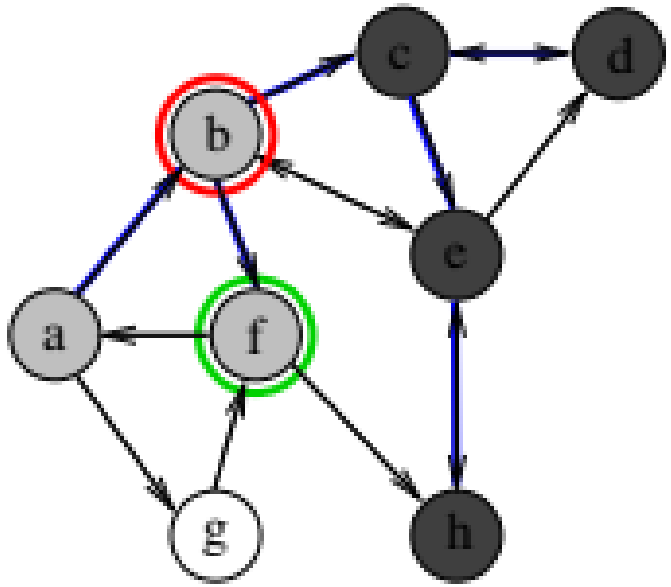
```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p =< a, b >$
$s_i = b$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

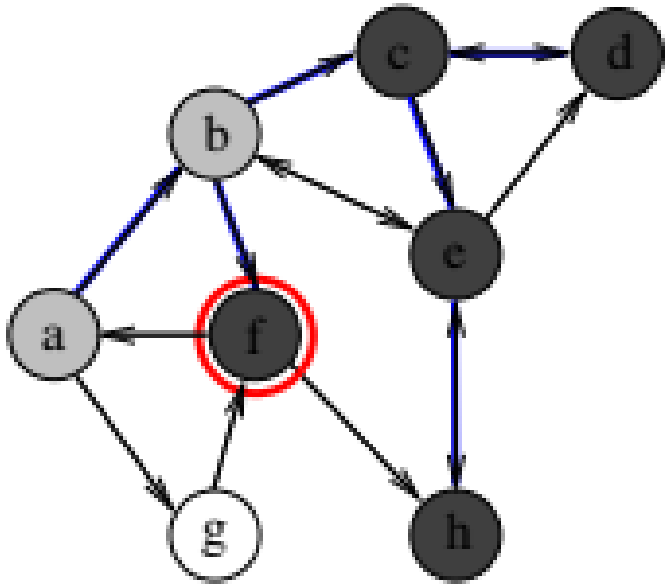


$p =< a, b, f >$

$s_i = b, s_j = f$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

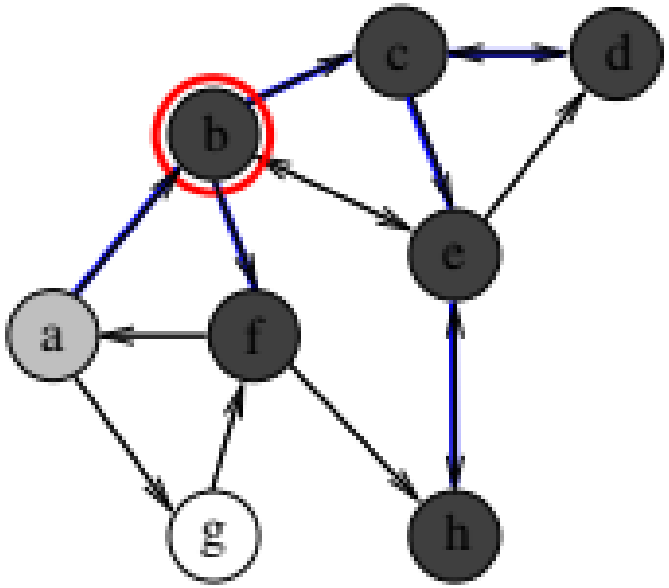


$p =< a, b >$

$s_i = f$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

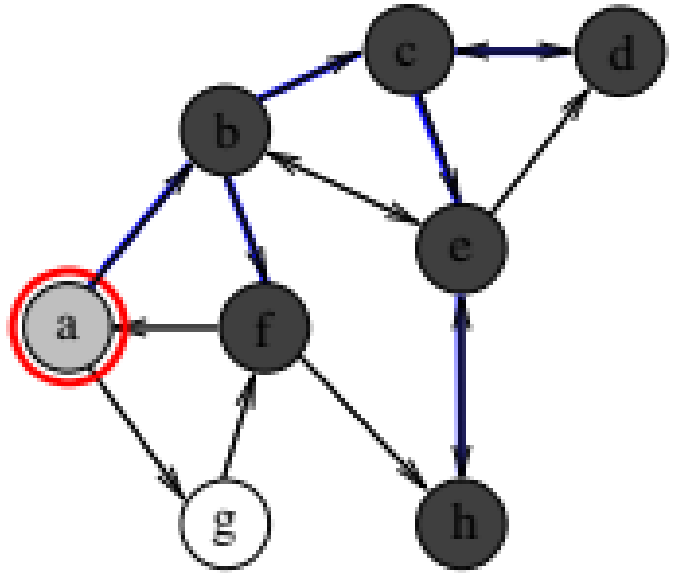


$p =< a >$

$s_i = b$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

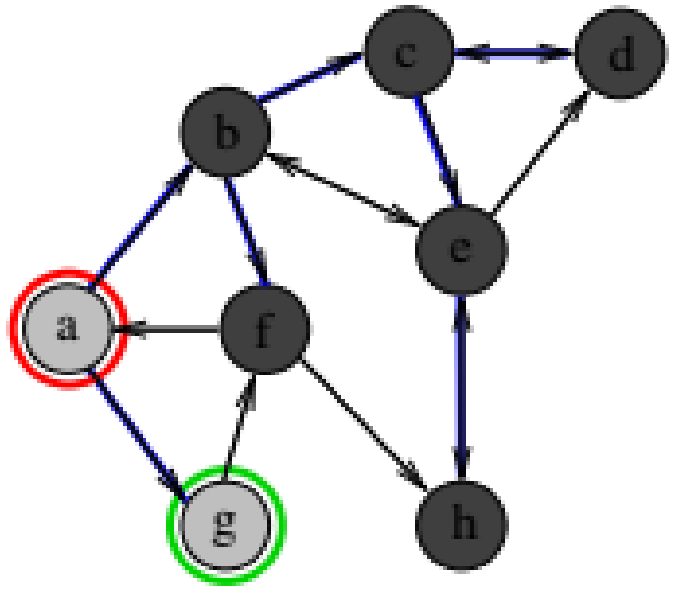


$p =< a >$

$s_i = a$

Parcours en profondeur (Depth First Search = DFS)

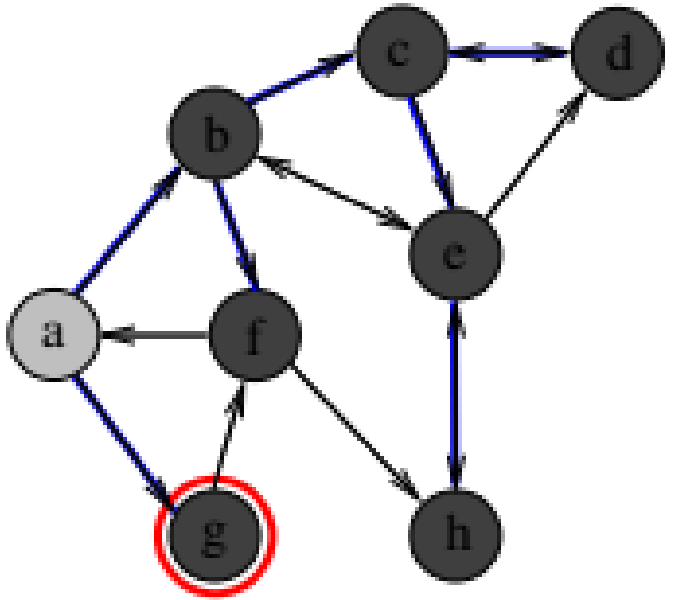
```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p =< a, g >$
$s_i = a, s_j = g$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

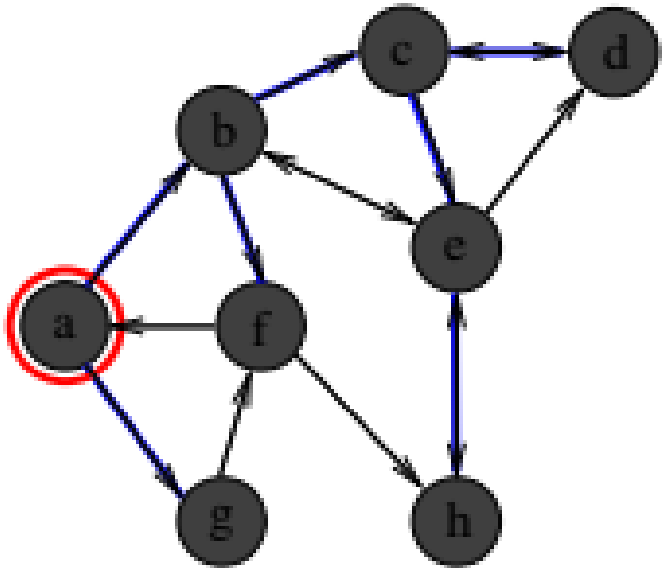


$p =< a >$

$s_i = g$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```

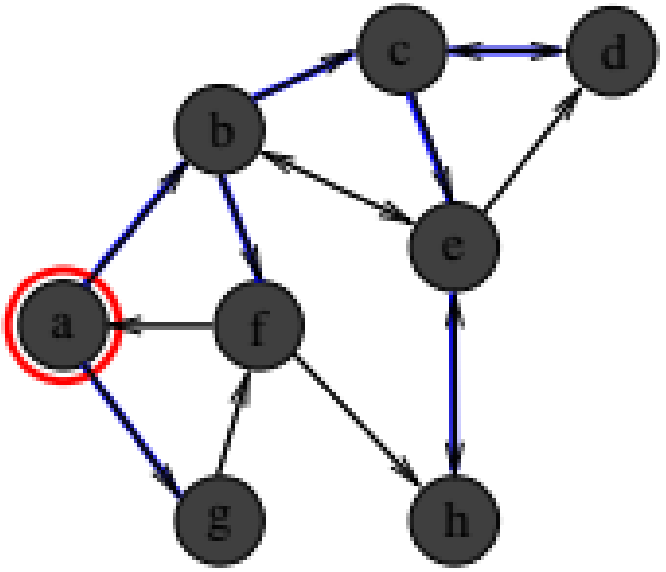


$p = <>$

$s_i = a$

Parcours en profondeur (Depth First Search = DFS)

```
1  Fonction DFS(g, s0)
2      Soit p une pile (LIFO) initialisée à vide
3      pour tout sommet si ∈ S faire
4          π[si] ← null
5          Colorier si en blanc
6      Empiler s0 dans p et colorier s0 en gris
7      tant que p n'est pas vide faire
8          Soit si le dernier sommet entré dans p
9          si ∃sj ∈ succ(si) tel que sj soit blanc alors
10             Empiler sj dans p et colorier sj en gris
11             π[sj] ← si
12         sinon
13             Dépiler si de p et colorier si en noir
14     retourne π
```



$p = <>$

$s_i = a$

III. Problème du plus court chemin

On se place dans le cas des graphes orientés valués **G = (S, A, ν).** Mais les résultats et les algorithmes présentés se généralisent facilement aux cas des graphes non orientés valués. Une autre solution consiste à transformer le graphe non-orienté en un graphe orienté en remplaçant une arête entre deux sommets par deux arcs de sens inverse entre ces sommets.

III. Problème du plus court chemin

Définition 2.8 (coût ou poids d'un chemin, coût d'un plus court chemin)

Le **coût ou poids d'un chemin c =< s_0 , s_1 , s_2 , … , s_k >** est égale à la somme des valuations des arcs composant le chemin, c'est à dire,

$$L(c) = \sum_{i=1}^{k} v(s_{i-1}, s_i)$$

Le **coût d'un plus court chemin** entre deux sommets **s_i** et **s_j** est noté **$\delta(s_i , s_j)$** est défini par :

$$\delta'(s_i, s_j) = \begin{cases} \min\{(L(c) / c = \text{chemin de } s_i \text{ à } s_j\} & s'il\ existe\ au\ moins\ un\ chemin\ entre\ s_i \text{ et } s_j \\ +\infty & Sinon \end{cases}$$

III. Problème du plus court chemin

Dans la recherche d'un plus court chemin de **x** à **y**, trois cas peuvent se présenter :

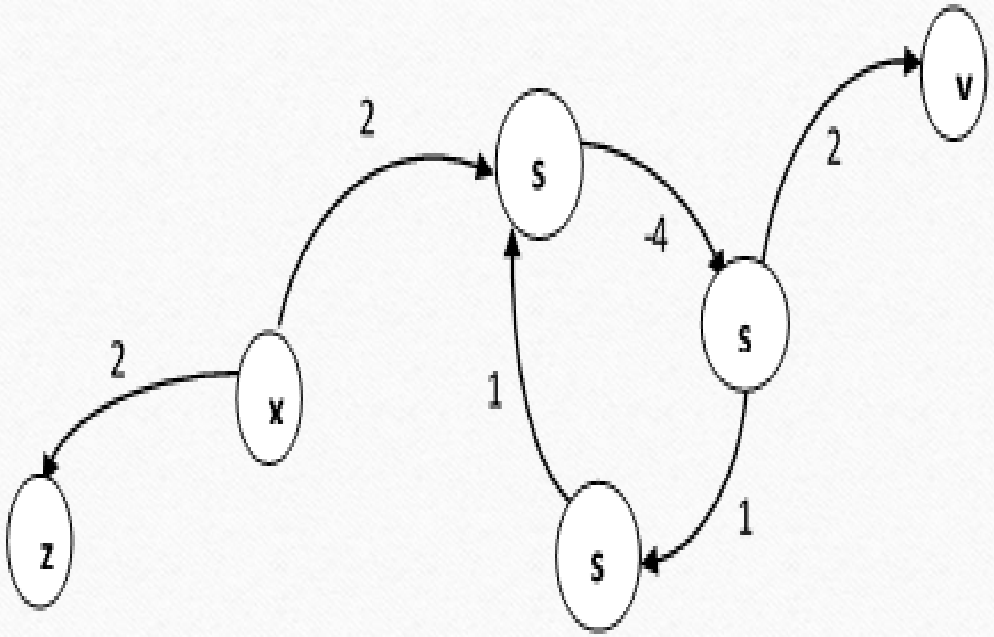
- Il n'existe aucun chemin de **x** à **y** (par exemple, si **x** et **y** appartiennent à deux composantes fortement connexes différentes de **G**).
- Il existe des chemins de **x** à **y** mais pas de plus court chemin.
- Il existe un plus court chemin de **x** à **y**.

III. Problème du plus court chemin

Exemple

On considère le graphe suivant :

Il existe des chemins de **x** à **y** (et même une infinité), mais il n'existe pas de plus court chemin de **x** à **y** (il suffit d'emprunter le cycle de poids négatif autant de fois que nécessaire). Il existe un plus court chemin de **x** à **z** mais pas de chemin de **z** à **x**.



III. Problème du plus court chemin

Définition 2.9 (circuit absorbant.)

Un circuit de longueur négative est appelé **circuit absorbant.**

Et alors, une condition nécessaire et suffisante d'existence de plus court chemin est donnée par le résultat suivant.

Proposition 2.1

Dans un graphe orienté valué fortement connexe **G = (S, A, ν)**, il existe un plus court chemin entre tout couple de sommets si et seulement s'il n'existe pas de circuit absorbant dans **G**.

III. Problème du plus court chemin

Preuve : On montre que si un chemin entre deux sommets **x** et **y** possède un circuit absorbant, alors $\pi(x, y) = -\infty$.

En effet, dès que l'on parcourt le circuit absorbant, on diminue la longueur du chemin, et on peut donc diminuer cette longueur à l'infini. En augmentant "infiniment" le nombre de tours dans le circuit, on obtient $\pi(x , y) = -\infty$.

III. Problème du plus court chemin

Définition 2.10 (problème des plus courts chemins à origine unique)

Étant donné un graphe orienté valué **G = (S, A, ν)** et un sommet origine $\mathbf{s_0 \in S}$, le **problème des plus courts chemins à origine unique** consiste à calculer pour chaque sommet $\mathbf{s_j \in S}$ le coût $\mathbf{\delta(s_0, s_j)}$ du plus court chemin de $\mathbf{s_0}$ à $\mathbf{s_j}$.

➔ On supposera que le graphe **G** ne comporte pas de circuit absorbant.

III. Problème du plus court chemin

Variantes du problème :

- Si l'on souhaite calculer le plus court chemin allant d'un sommet $\mathbf{s_0}$ vers un autre sommet $\mathbf{s_i}$ (la destination est unique), on pourra utiliser la résolution du problème précédant (qui calcule tous les plus courts chemins partant de $\mathbf{s_0}$). En effet, on ne connaît pas d'algorithme plus efficace pour résoudre ce problème.
- Si l'on souhaite calculer tous les plus courts chemins entre tous les couples de sommets possibles, on pourrait aussi utiliser la résolution du problème précédent, mais dans ce cas, on n'obtiendrait pas un algorithme optimal. Il faudra utiliser dans ce cas un algorithme spécifique, par exemple l'algorithme de Floyd-Warshall.

III. Problème du plus court chemin

Spécification d'un algorithme de plus courts chemins à origine unique

Fonction *PlusCourtsChemins*(g, c, s_0)

Entrée : Un graphe (orienté ou non) $g = (S, A)$
Une fonction de coût $c : A \rightarrow \mathbb{R}$
Un sommet de départ $s_0 \in S$

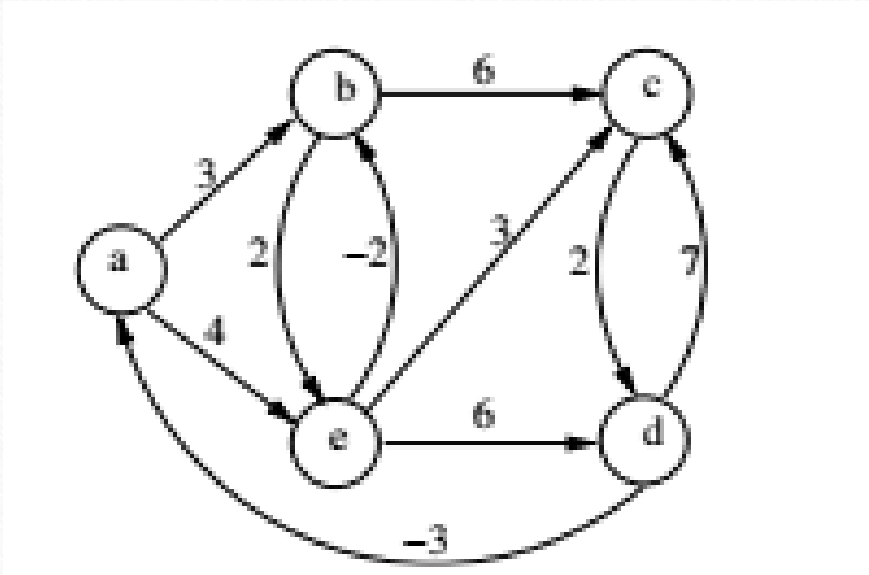
Sortie : Une arborescence des plus courts chemins partant de s_0
Un tableau d tel que $\forall s_i \in S, d[s_i] = \delta(s_0, s_i)$

III. Problème du plus court chemin

Arborescence des plus courts chemins

- Tableau π tq $\pi[s_0] = null$ et $\pi[s_j] = s_i$ si $s_i \rightarrow s_j$ est un arc de l'arbo
- Rm : $\forall s_i \in S; \pi[s_i] \neq null \Rightarrow \delta(s_0, s_i) = \delta(s_0, \pi[s_i]) + c(\pi[s_i], s_i)$

Exemple :



III. Problème du plus court chemin

Principe des algorithmes de recherche de plus courts chemins

- Initialiser $d[s_0]$ à 0
- $\forall s_i \in S \setminus \{ s_0 \}$, initialiser $d[s_i]$ à $+1$
- $\rightarrow d[s_i]$ = borne supérieure de $\delta(s_0, s_i)$
- Grignoter itérativement les bornes d en **relâchant** des arcs

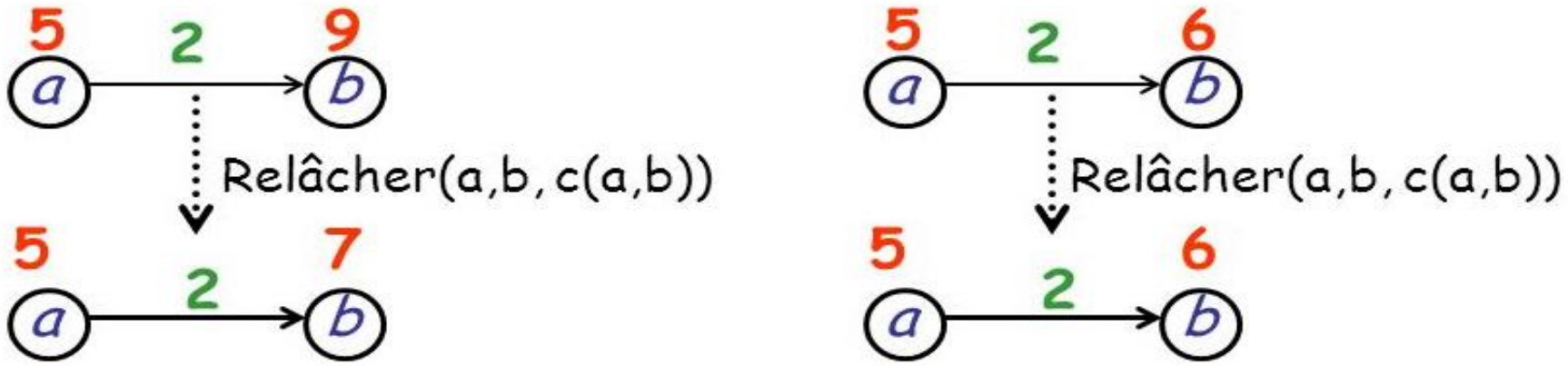
III. Problème du plus court chemin

Algorithme 4 : Relâchement de l'arc (s_i, s_j)

1 : **Procédure** relâcher((s_i, s_j), v , d, π)
Entrée : Un arc (s_i, s_j) ; v : valuation des arcs ($v(s_i, s_j) = cout(s_i, s_j)$)
Entrée/Sortie : **d** tableau des bornes maximum des coûts,
π arborescence couvrante
Précondition : $d[s_i] \geq \delta(s_0, s_i)$ et $d[s_j] \geq \delta(s_0, s_j)$
Postcondition : $\delta(s_0, s_j) \leq d[s_j] \leq d[s_i] + v(s_i, s_j)$ et $\pi[s_j] = s_i$
si la borne $d[s_j]$ a été diminuée
2 : **Si** $d[s_j] > d[s_i] + v(s_i, s_j)$ **Alors**
3 : $d[s_j] \leftarrow d[s_i] + v(s_i, s_j)$
4 : $\pi[s_j] \leftarrow s_i$
5 : **Fin Si**
6 : **Fin**

III. Problème du plus court chemin

- *RELÂCHER (a,b, c (a,b))*
  - Si $y_b > y_a + c(a,b)$ *Alors* $y_b \leftarrow y_a + c(a,b)$



XII. Problème du plus court chemin

Principe commun aux algorithmes

Nous allons étudier deux algorithmes qui permettent de résoudre des problèmes de recherche de plus courts chemins à origine unique :

- un algorithme (Dijkstra) qui peut être utilisé dès lors que tous les coûts sont positifs ou nuls;
- un algorithme (Ford-Bellman) qui peut être utilisé pour n'importe quel graphe ne comportant pas de circuit absorbant.

III. Problème du plus court chemin

Principe commun aux algorithmes

- Grignotage progressif de d en relâchant des arcs.

Question : Dans quel ordre relâcher les arcs?

- Dijkstra relâche les arcs partant du sommet minimisant d

  → **Chaque arc est relâché exactement une fois**

  → **Ne marche que si tous les coûts sont positifs**

- Bellman-Ford relâche tous les arcs à chaque itération, jusqu'à convergence

  → Chaque arc est relâché plusieurs fois

  → **Marche dans tous les cas**

III. Problème du plus court chemin

Principe commun aux algorithmes

Les algorithmes de Dijkstra et Bellman-Ford procèdent tous les deux par relâchements successifs d'arcs. La différence entre les deux est que dans l'algorithme de Dijkstra, chaque arc est relâché une et une seule fois, tandis que dans l'algorithme de Bellman-Ford, chaque arc peut être relâché plusieurs fois.

Algorithme 5 : Dijkstra(S , A , v , s_0 , d , π)

Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **v** valuations des arcs,
s_0 sommet de départ

Sorties : **d** tableau des bornes maximum des coûts, **π** arborescence couvrante

1 : **Pour** chaque sommet $s_i \in S$ **Faire**
2 : $d[s_i] \leftarrow +\infty$
3 : $\Pi[s_i] \leftarrow$ nil
4 : **Fin Pour**
5 : $d[s_0] \leftarrow 0$
6 : $E \leftarrow \varnothing$
7 : $F \leftarrow S$
8 : **Tant que** $F \neq \varnothing$ **Faire**
9 : soit s_i le sommet de F tel que $d[s_i]$ soit minimal
10 : /* $d[s_i] = \delta(s_i , s_i)$ */
11 : $F \leftarrow F - \{ s_i \}$
12 : $E \leftarrow E \cup \{ s_i \}$
13 **Pour** tout sommet $s_j \in succ(s_i) \cap F$ **Faire**
14 : relâcher((s_i, s_j), v , d, π)
15 : **Fin Pour**
16 : **Fin Tant que**
17 : **Fin**

Principe de l'algorithme de Dijkstra

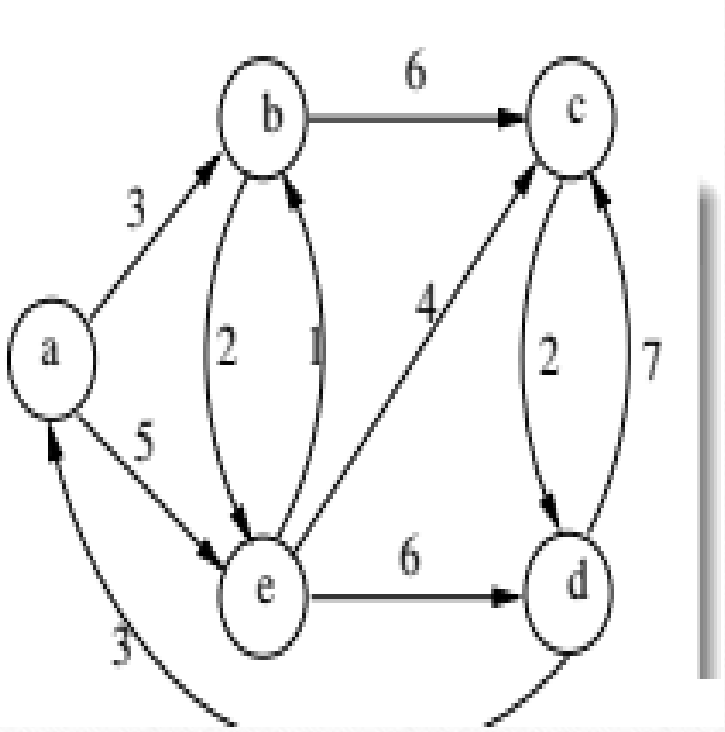
Propriété

Pour tout sommet s_i :

- Si s_i est gris alors $d[s_i]$ = longueur du plus court chemin de s_0 à s_i ne passant que par des sommets noirs
- Si s_i est noir alors $d[s_j] = \delta(s_0, s_i)$
- Les succ. d'un sommet noir sont gris ou noirs

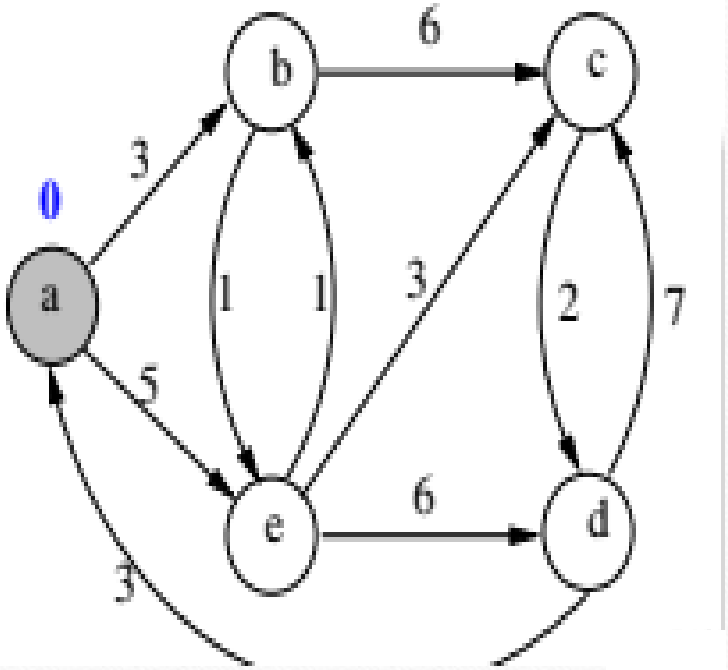
Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```

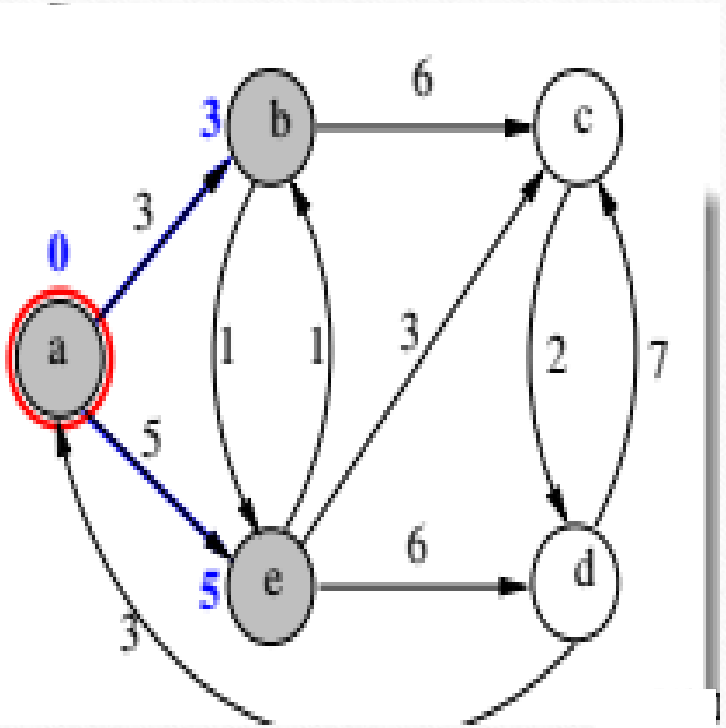


Exemple

$s_0 = a$

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



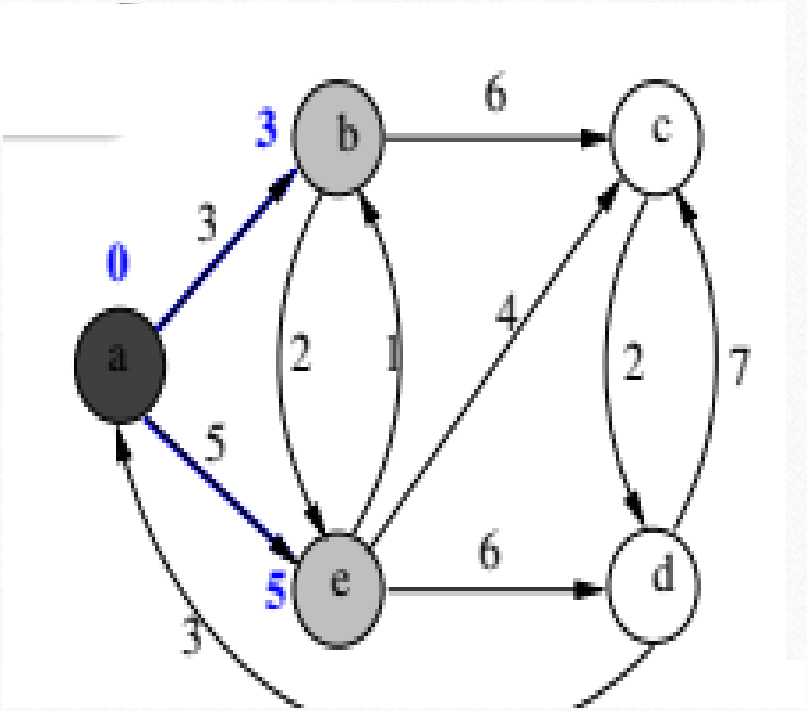
Exemple

$s_i = a$

Arcs relâchés : (a, b), (a, e)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



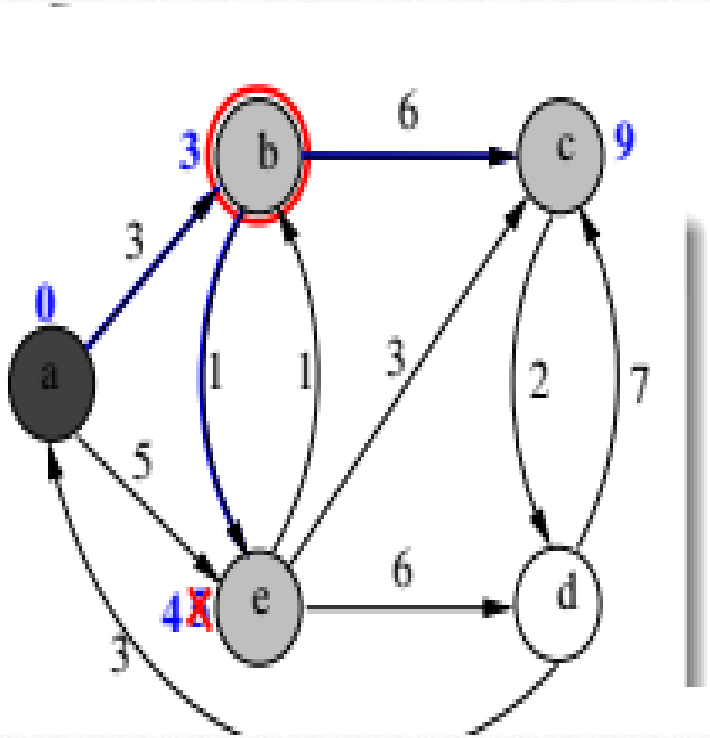
Exemple

$s_i = a$

Arcs relâchés : (a, b), (a, e)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



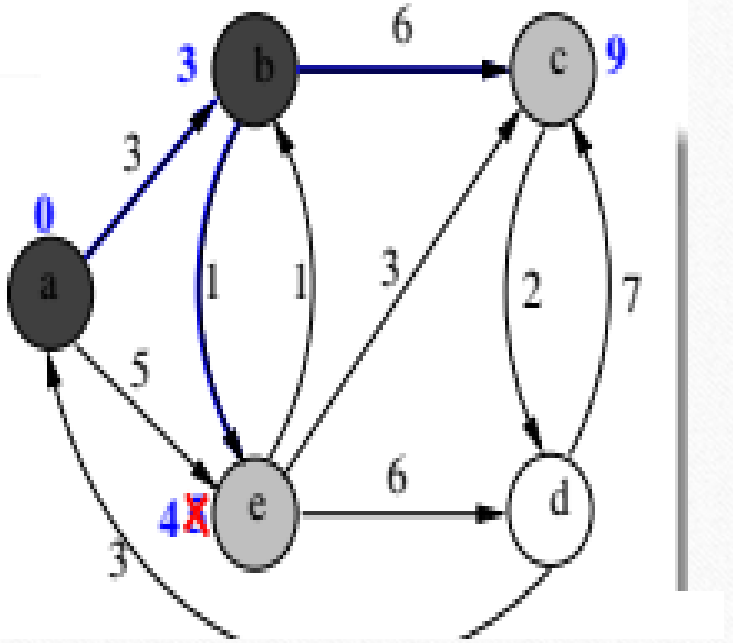
Exemple

$s_i = b$

Arcs relâchés : (a, b), (a, e), (b, c)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



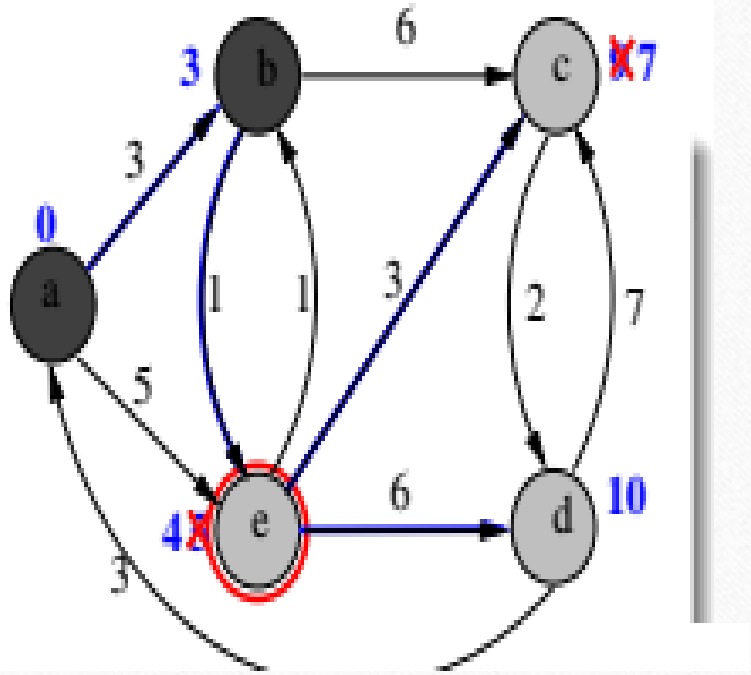
Exemple

$s_i = b$

Arcs relâchés : (a, b), (a, e), (b, c)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



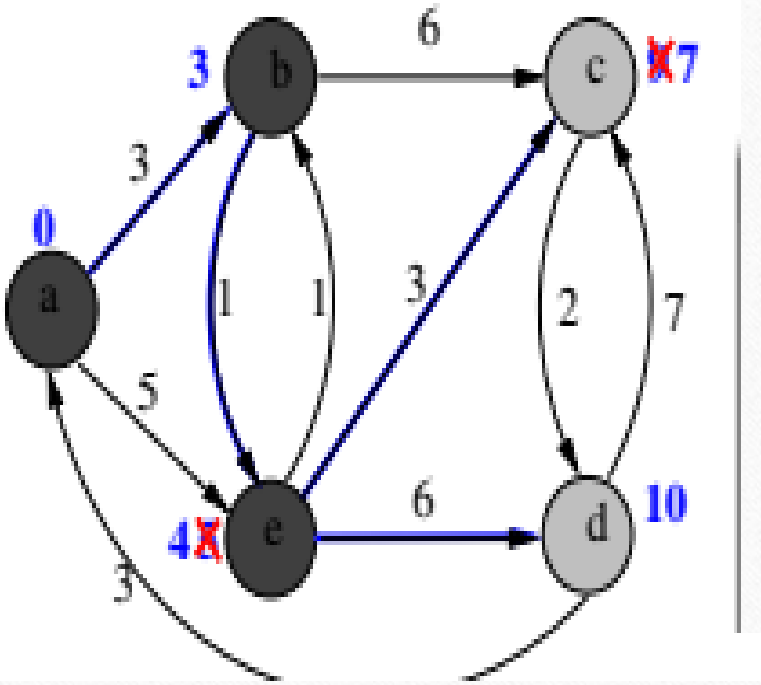
Exemple

$s_i = e$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



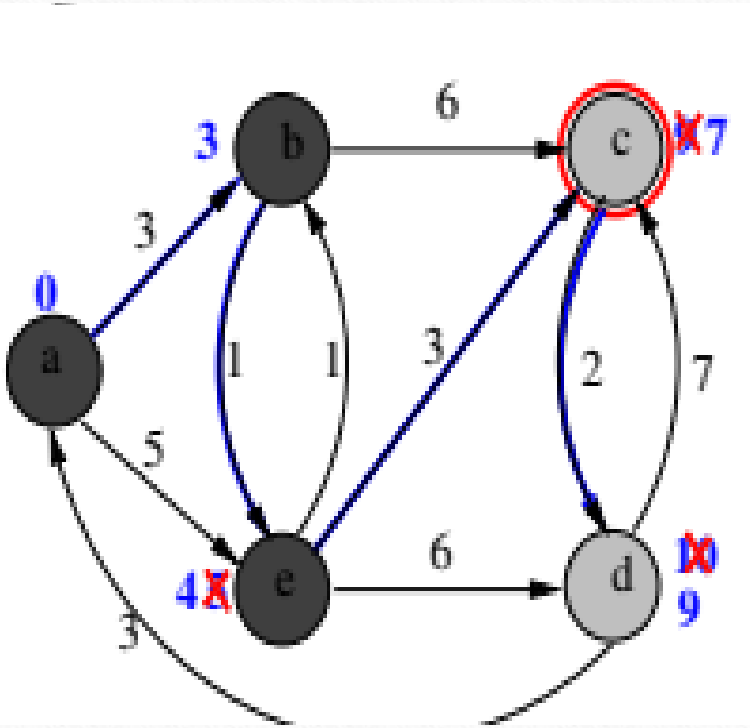
Exemple

$s_i = e$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



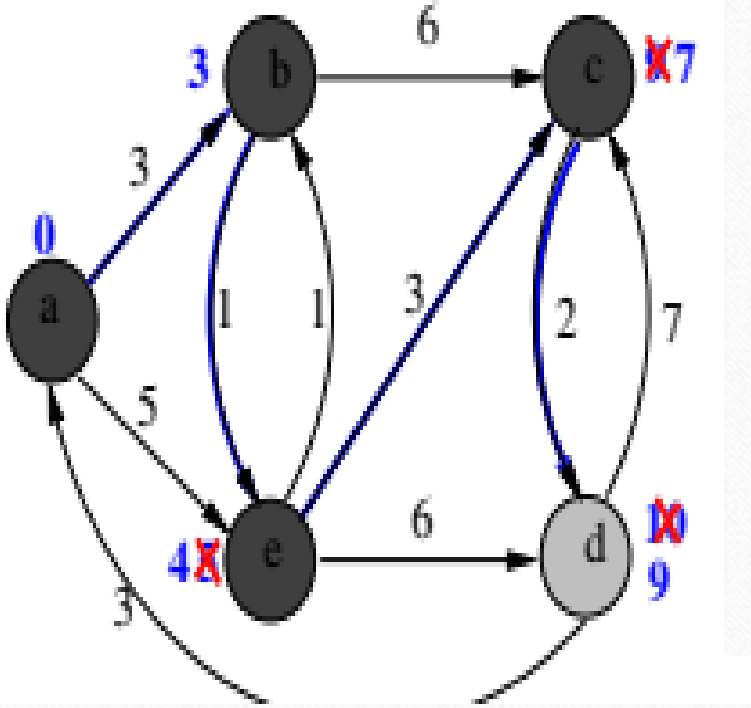
Exemple

$s_i = c$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d), (c, d)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



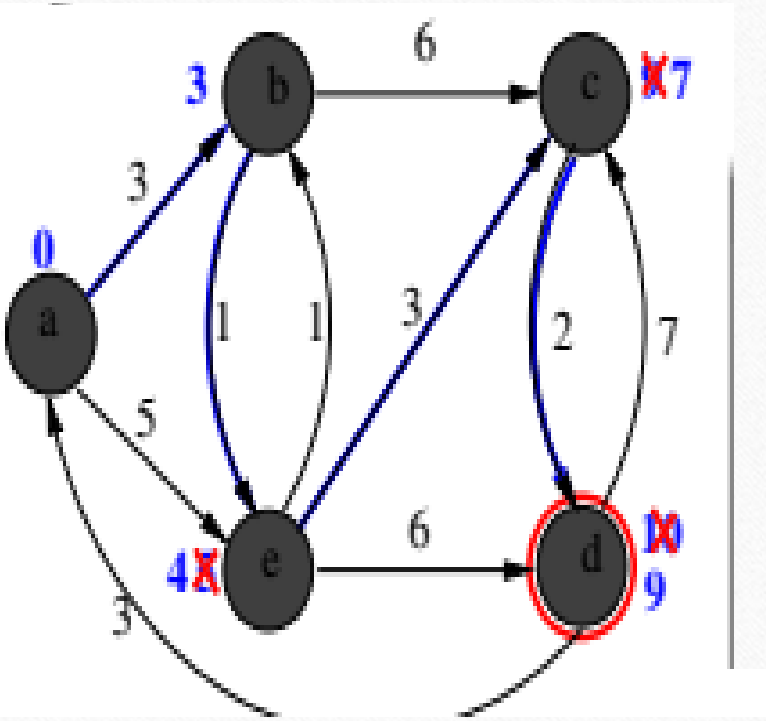
Exemple

$s_i = c$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d), (c, d)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



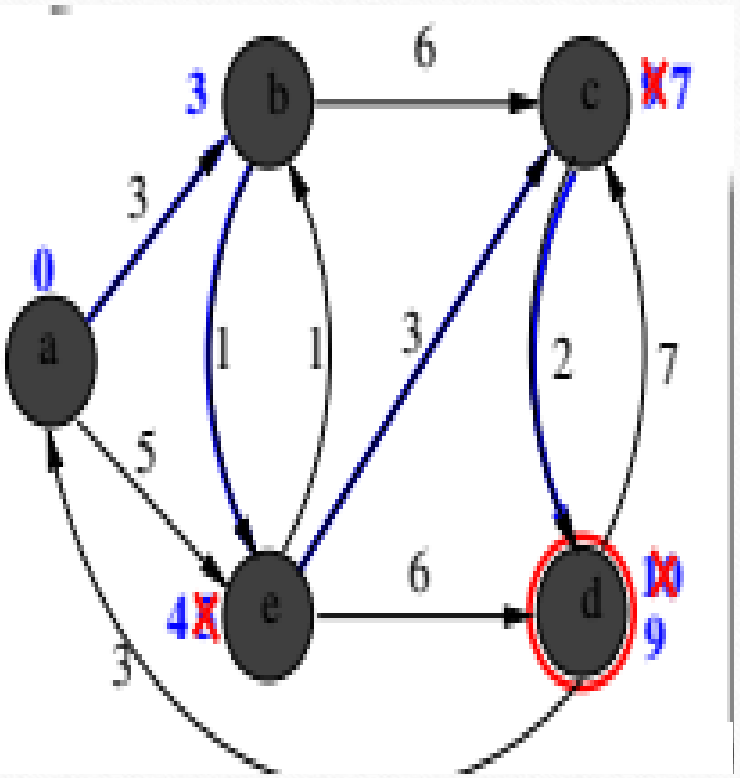
Exemple

$s_i = d$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d), (c, d)

Principe de l'algorithme de Dijkstra

```
1  Fonction Dijkstra(g, c, s0)
2      pour chaque sommet si ∈ S faire
3          d[si] ← +∞ ; π[si] ← null ; Colorier si en blanc
4      d[s0] ← 0 ; Colorier s0 en gris
5      tant que il existe un sommet gris faire
6          Soit si le sommet gris tq d[si] soit minimal
7          pour tout sommet sj ∈ succ(si) faire
8              si sj est blanc ou gris alors
9                  relacher((si, sj), π, d)
10                 si sj est blanc alors
11                     Colorier sj en gris
12         Colorier si en noir
13     retourne π et d
```



Exemple

$s_i = d$

Arcs relâchés : (a, b), (a, e), (b, c), (e, c) , (e, d), (c, d)

Principe de l'algorithme de Dijkstra

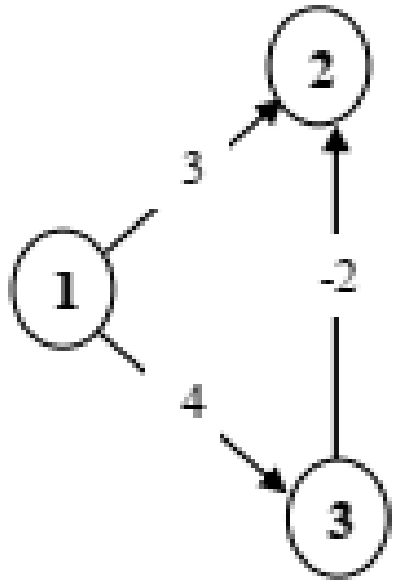
Complexité :

- On suppose que le graphe possède **n** sommets et **m** arcs. Si on utilise une matrice d'adjacence, l'algorithme sera en $\mathbf{O(n^2)}$.
- Par conséquent, si on implémente **F** avec un tas binaire, on obtient une complexité pour Dijkstra en $\mathbf{O(m * log(n))}$.

Principe de l'algorithme de Bellman-Ford

L'algorithme de Dijkstra ne marche pas toujours quand le graphe contient des arcs dont les coûts sont négatifs.

Exemple



			Dist			Préd		
s	E	F	1	2	3	1	2	3
0	{}	{1, 2, 3}	0	∞	∞	0	0	0
1	{1}	{2, 3}	0	3	4	0	1	1
2	{1, 2}	{3}	0	3	4	0	1	1
3	{1, 2, 3}	{}	0	3	4	0	1	1

Principe de l’algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants).

Algorithme 6 : Bellman-ford (S , A , ν , s_0 , d , π)

Entrées : S ensemble des sommets, A ensemble des arcs, ν valuations des arcs,
s_0 sommet de départ
Sorties : d tableau des bornes maximum des coûts, π arborescence couvrante

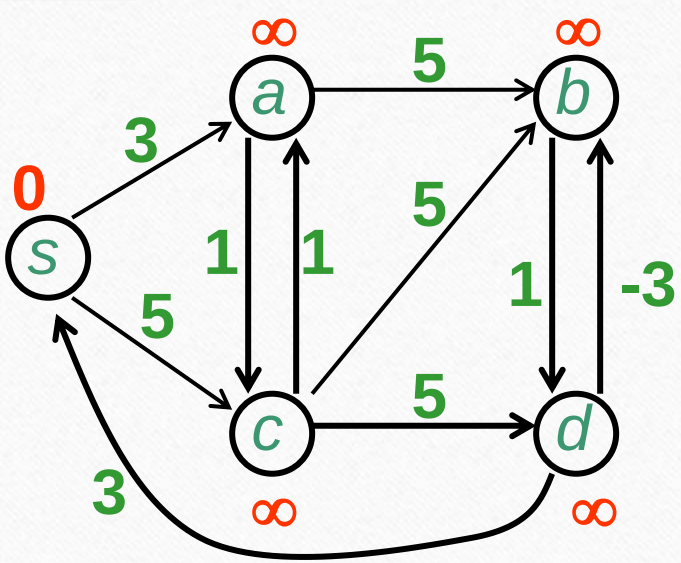
```
1 :   Pour chaque sommet s_i ∈ S Faire
2 :      d[s_i] ← +∞
3 :      Π[s_i] ← nil
4 :   Fin Pour
5 :      d[s_0] ← 0
6 :    Pour k variant de 1 à |S| - 1   Faire
7 :      Pour chaque arc (s_i, s_j) ∈ A Faire
8 :                relâcher((s_i, s_j), ν , d, π)
9 :       Fin Pour
10 :  Fin Pour
11 :  Pour chaque arc (s_i, s_j) ∈ A Faire
12 :     relâcher((s_i, s_j), ν , d, π)
13 :     Si d[s_j] > d[s_i] + ν(s_i, s_j) Alors
14 :              afficher("circuit absorbant")
15 :     Fin Si
16 :  Fin Pour
17 : Fin
```

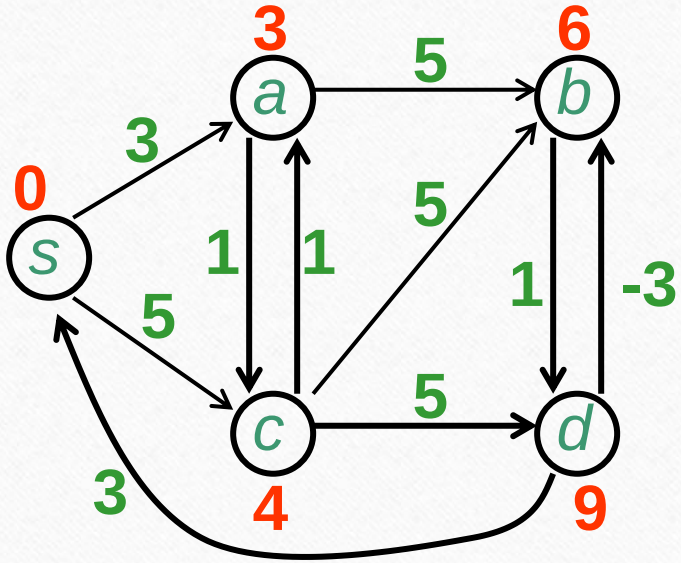
Principe de l'algorithme de Bellman-Ford

Complexité : Si le graphe comporte **n** sommets et **m** arcs, chaque arc sera relâché **n-1** fois, et on effectuera donc au total **(n-1)m** relâchements successifs. Si le graphe est représenté par une matrice d'adjacence, on aura une complexité en $\mathbf{O(n^3)}$, alors que s'il est représenté par des listes d'adjacence, on aura une complexité en **O(nm)**.

Principe de l'algorithme de Bellman-Ford

Exemple 1

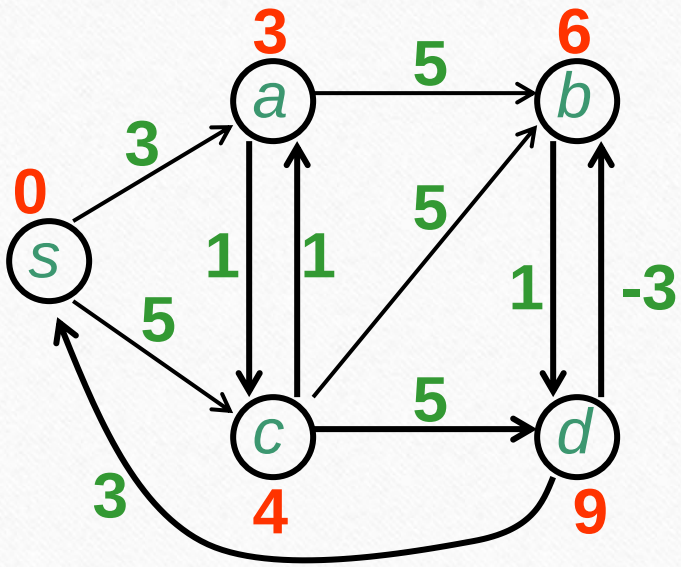


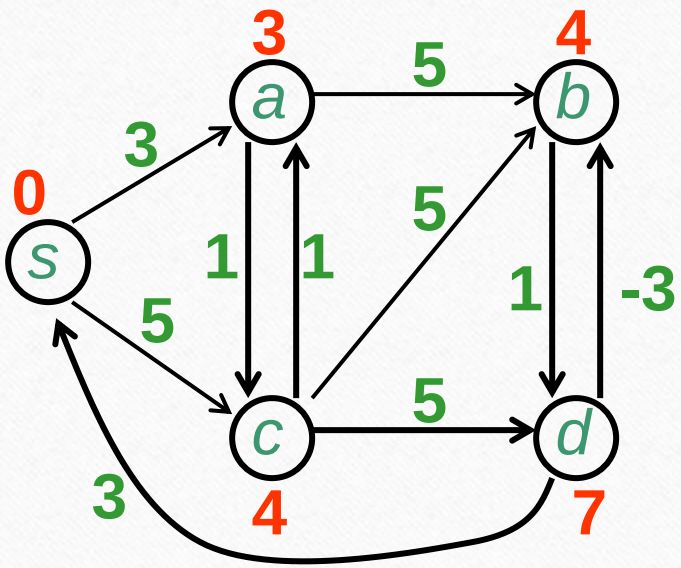


Étape 1 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Principe de l'algorithme de Bellman-Ford

Exemple 1 (suite)

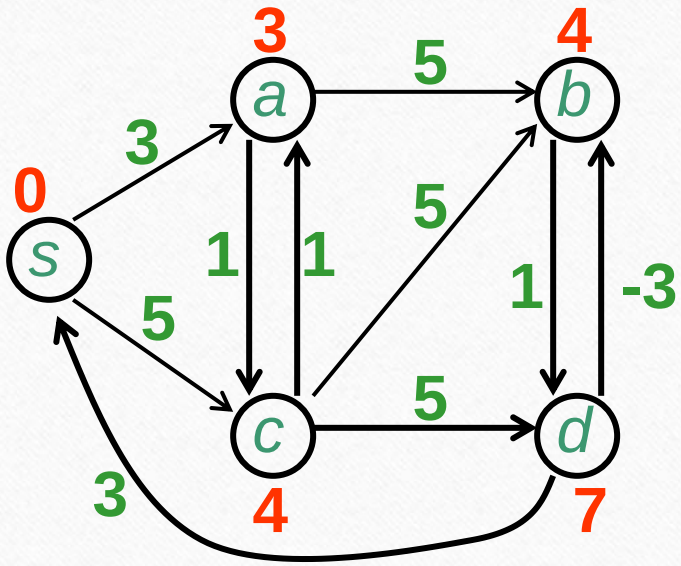


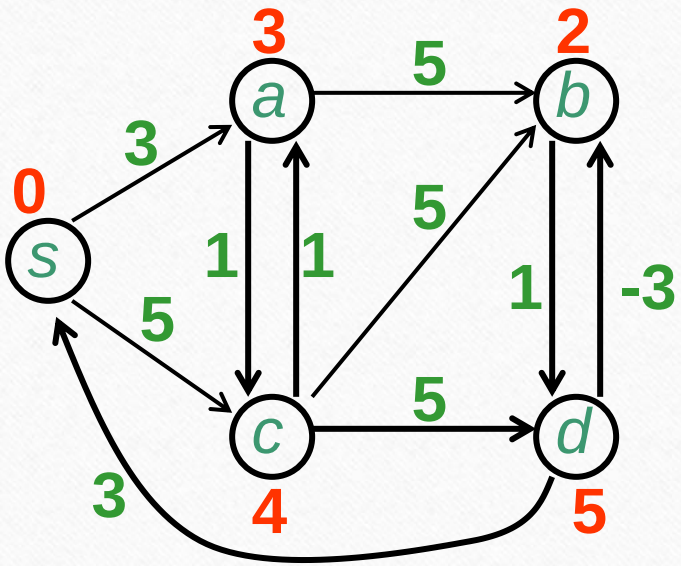


Étape 2 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Principe de l'algorithme de Bellman-Ford

Exemple 1 (suite)

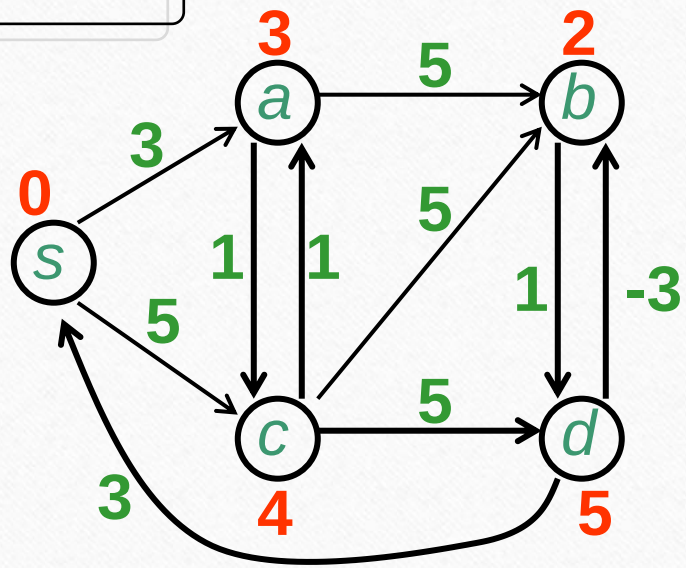


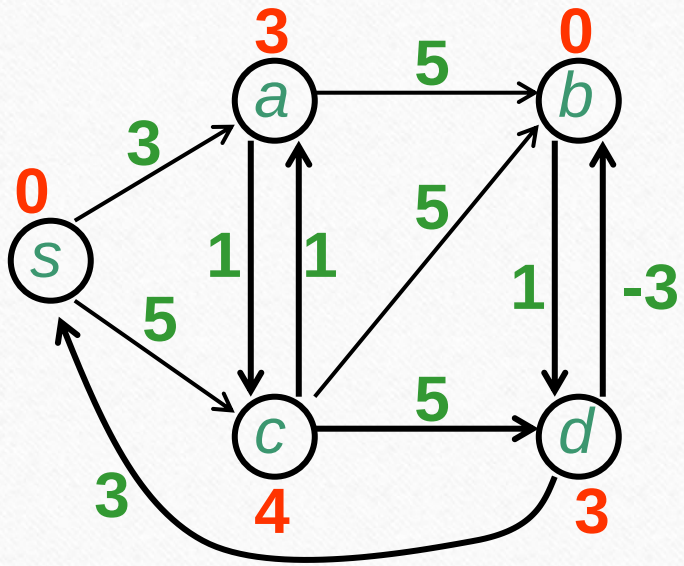


Étape 3 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Principe de l'algorithme de Bellman-Ford

Exemple 1 (suite)

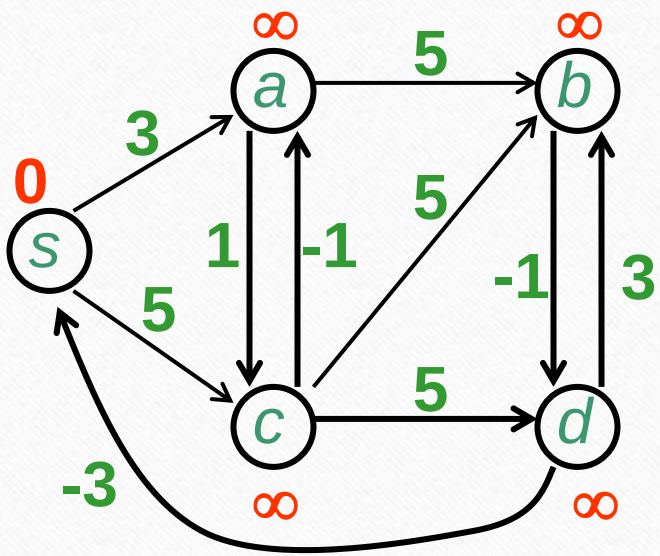


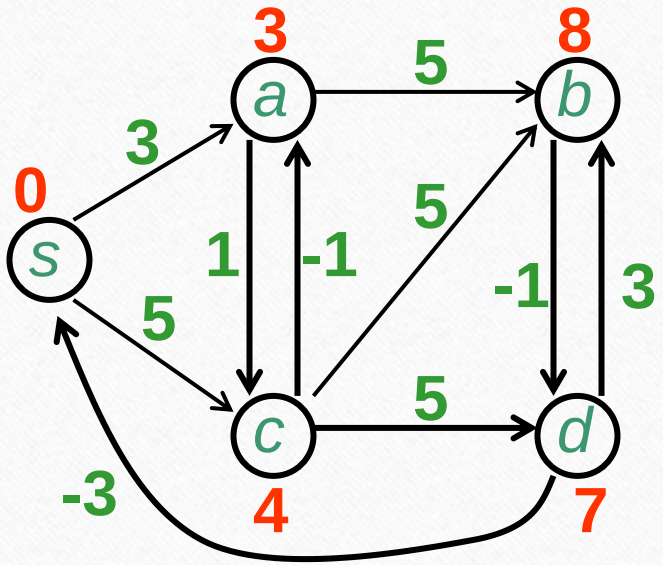


Étape 4 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)
reduction possible : cycle de coût négatif

Principe de l'algorithme de Bellman-Ford

Exemple 2

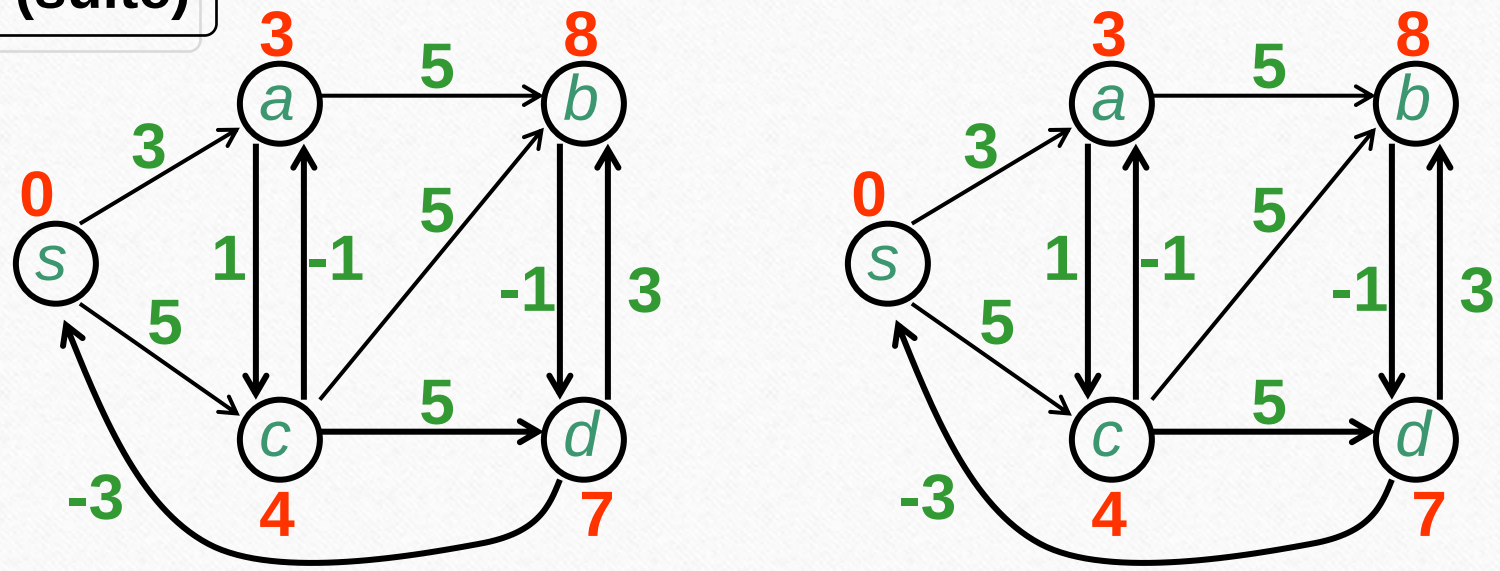




Étape 1 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Principe de l'algorithme de Bellman-Ford

Exemple 2 (suite)



Étape 2 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)
pas de réduction possible : coûts corrects

Principe de l'algorithme de Bellman-Ford

Remarque : En pratique, on pourra arrêter l'algorithme dès lors qu'aucune valeur de **d** n'a été modifiée pendant une itération complète. On pourra aussi mémoriser à chaque itération l'ensemble des sommets pour lesquels la valeur de **d** a changé, afin de ne relâcher lors de l'itération suivante que les arcs partant de ces sommets.

IV. Synthèse

En résumé, en fonction des caractéristiques du problème à résoudre il faudra choisir le bon algorithme :

- Si le graphe ne comporte pas de circuit alors, que l'on recherche un plus court chemin ou un plus long chemin, il suffit de trier les sommets topo-logiquement avec un parcours en profondeur d'abord, puis de considérer chaque sommet dans l'ordre ainsi défini et relâcher à chaque fois tous les arcs partant de ce sommet ;

IV. Synthèse

- Si le graphe comporte des circuits, alors
  - ✓ Si on recherche un plus court chemin, alors
    - ❑ Si la fonction coût est monotone croissante (le coût d'un chemin ne peut qu'augmenter lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont positifs et que le coût d'un chemin est égal à la somme des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
    - ❑ Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants),

IV. Synthèse

- Si le graphe comporte des circuits, alors
  - ✓ Si on recherche un plus long chemin, alors
    - ❑ Si la fonction coût est monotone décroissante (le coût d'un chemin ne peut que diminuer lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont compris entre 0 et 1 et que le coût d'un chemin est égal au produit des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
    - ❑ Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants).

V. Arbres couvrants minimaux

Vous êtes chargés de l'installation du câble dans la région M’sila-Alger. Vous disposez pour cela d'une carte de l'ensemble du réseau routier (le câble est généralement disposé le long des routes). On vous demande de définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale et qu'un certain nombre de lieux soient desservis.

V. Arbres couvrants minimaux

On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe **G = (S, A)**, où **S** associe un sommet à chaque lieu devant être desservi, et **A** contient une arête pour chaque portion de route entre **2** lieux. Ce graphe est valué par une fonction de coûts qui spécifie pour chaque arête $\{s_i; s_j\}$ la longueur de câble nécessaire pour connecter s_i à s_j.

Il s'agit alors de trouver un sous-graphe connexe et sans cycle de ce graphe (autrement dit, un arbre) qui recouvre l'ensemble des sommets du graphe. Ce graphe est appelé arbre couvrant. On cherche à minimiser le poids total des arêtes de l'arbre. On dira qu'on cherche l'arbre couvrant minimal **(ACM).**

V. Arbres couvrants minimaux

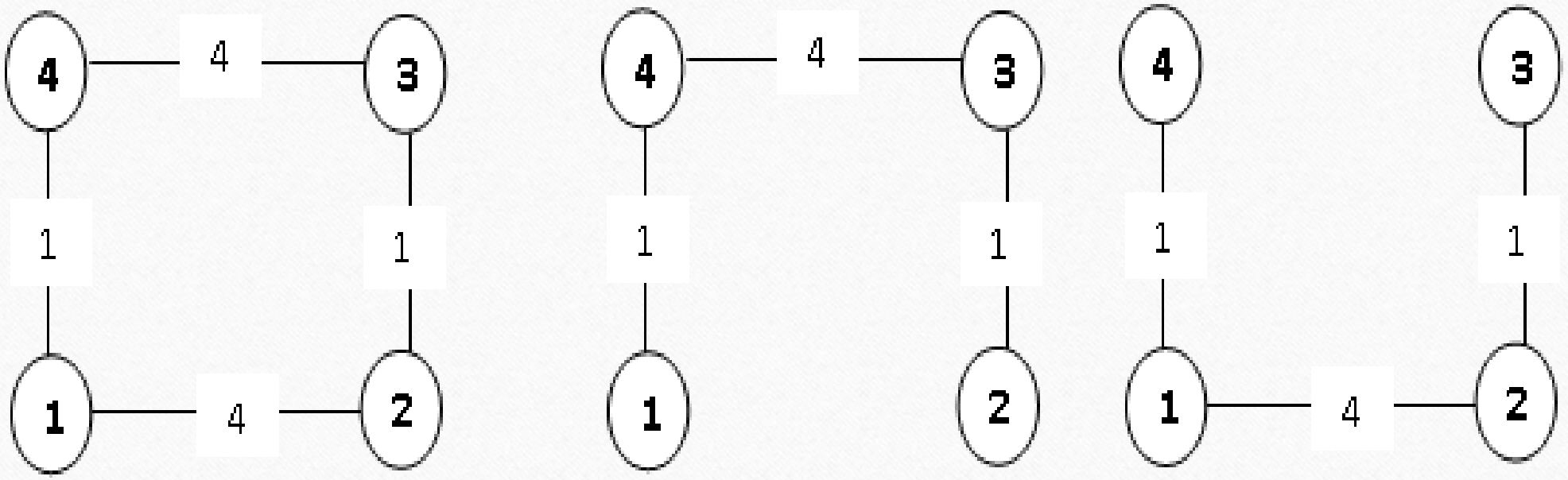
De façon plus formelle, un arbre couvrant minimal (ACM) (Minimal Spanning Tree / MST) d'un graphe **G = (S, A)** est un graphe partiel **G′= (S, A′)** de **G** tel que **G′** est connexe et sans cycle (**G′** est un arbre), et la somme des coûts des arêtes de **A′** est minimale.

Remarque : il peut exister plusieurs ACM, de même coût, associés à un même graphe.

V. Arbres couvrants minimaux

Exemple

2 ACMs



V. Arbres couvrants minimaux

Pour construire un ACM, on adopte une stratégie locale "gloutonne" qui consiste à sélectionner, de pas en pas, une arête devant faire partie de l'ACM. À chaque fois, on choisira la "meilleure" arête selon un certain critère. Il existe deux algorithmes différents suivant une telle stratégie gloutonne et permettant de calculer un ACM à partir d'un graphe : l'algorithme de Kruskal et l'algorithme de Prim. Ces deux algorithmes ont des complexités équivalentes.

V. Arbres couvrants minimaux

Principe générique

Dans ce chapitre, nous allons étudier deux algorithmes permettant de calculer des MST. Les deux algorithmes fonctionnent selon un principe glouton décrit dans **l'algorithme 6** (principe glouton générique pour calculer un MST) : l'idée est de sélectionner, à chaque itération, une arête de coût minimal traversant une coupure.

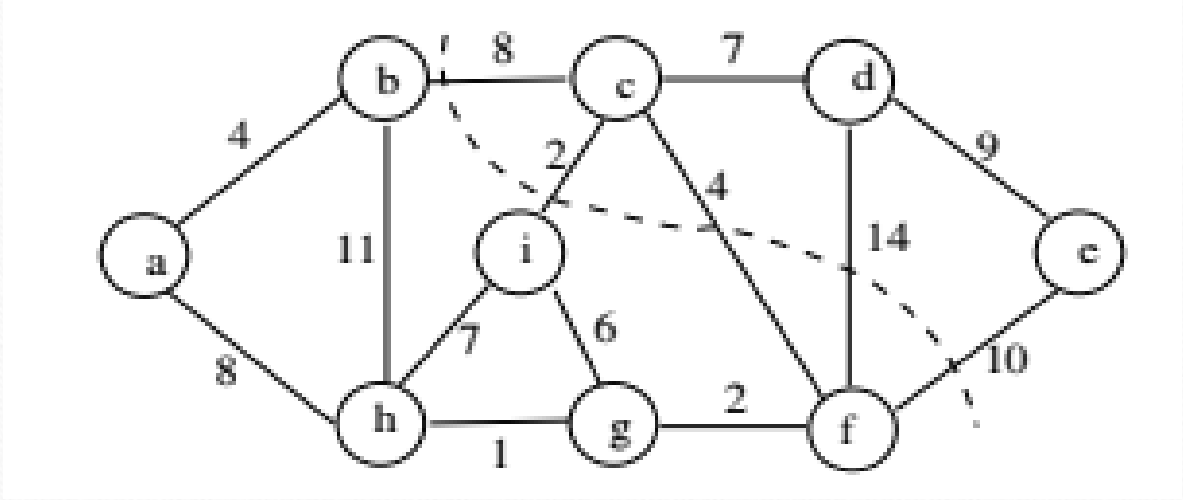
Une **coupure** d'un graphe **G = (S, A)** est une partition de l'ensemble des sommets en deux parties **(P, S \ P)**. Une arête **(s_i, s_j) traverse** une coupure **(P, S \ P)** si chaque extrémité de l'arête appartient à une partie différente, i.e., **$s_i \in P$ et $s_j \in S \setminus P$, ou $s_j \in P$ et $s_i \in S \setminus P$**. Une coupure **respecte** un ensemble d'arêtes **E** si aucune arête de **E** n'est traversée par la coupure.

V. Arbres couvrants minimaux

Principe générique

Considérons par exemple le graphe suivant :

La coupure **({a, b, f, g, h, i}, {c, d, e})** est représentée en pointillés et traverse les arêtes **{b, c}, {i, c}, {c, f}, {d, f} et {e, f}**. L'arête de coût minimal traversant cette coupure est **{i, c}.**



V. Arbres couvrants minimaux

Algorithme 6 : Principe glouton générique pour calculer un MST

Fonction MSTgénérique(g, cout)

Entrées : Un graphe **g = (S, A)** et une fonction cout **: A → ℜ**

Postcondition : Retourne un ensemble d'arêtes **E ⊆ A tel que (S, E)** est un MST de **g**

1 : E ← ∅
2 : **Tant que** |E|<|S|-1 **Faire**
3 : Soit **(P, S \ P)** une coupure (quelconque) qui respecte **E**
4 : Ajouter dans **E** une arête de coût minimal traversant la coupure **(P, S \ P)**
5 : **Fin Tant que**
6 : **Retourne** E
7 : **Fin**

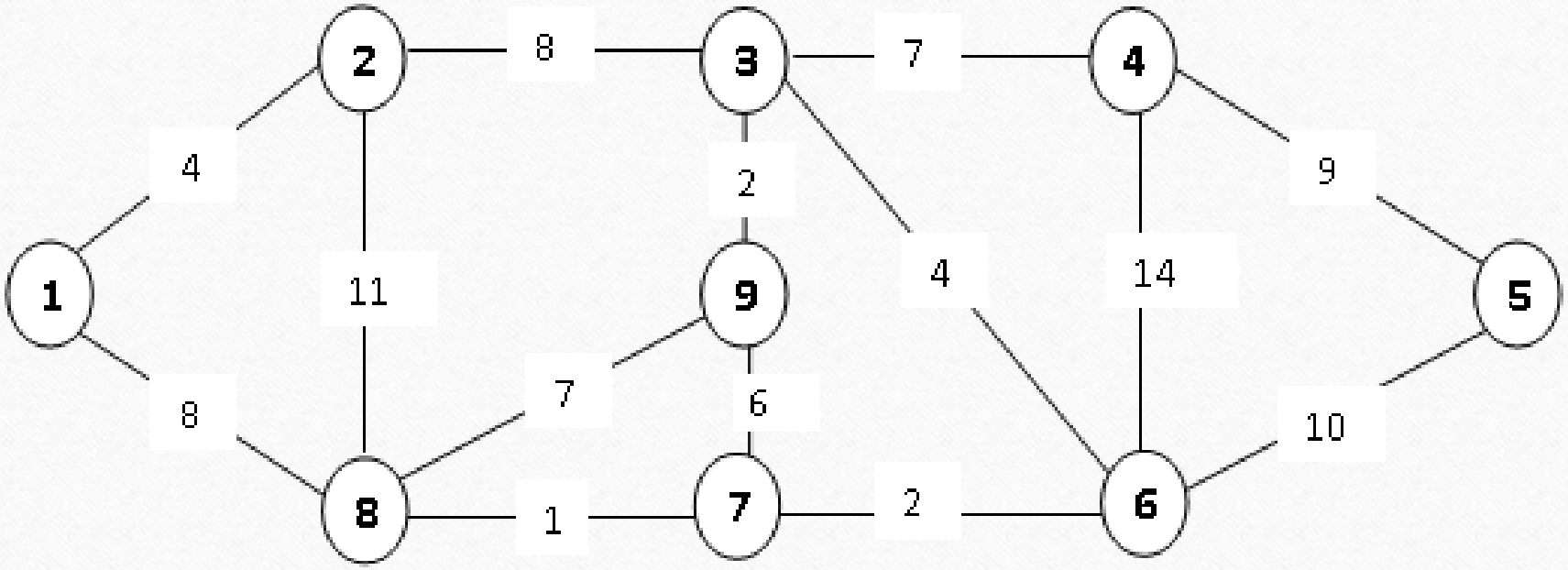
Algorithme de Kruskal

Principe :

- On commence par trier l'ensemble des arêtes du graphe par ordre de coût croissant.
- On va sélectionner de proche en proche les arêtes devant faire partie de l'ACM. Au début, cet ensemble est vide.
- On considère ensuite chacune des arêtes du graphe selon l'ordre que l'on vient d'établir (de l'arête de plus faible coût jusqu'à l'arête de plus fort coût).
- À chaque fois, si l'arête que l'on est en train de considérer peut être ajoutée à l'ensemble des arêtes déjà sélectionnées pour l'ACM sans générer de cycle, alors on la sélectionne, sinon on l'abandonne.

Algorithme de Kruskal

Exemple

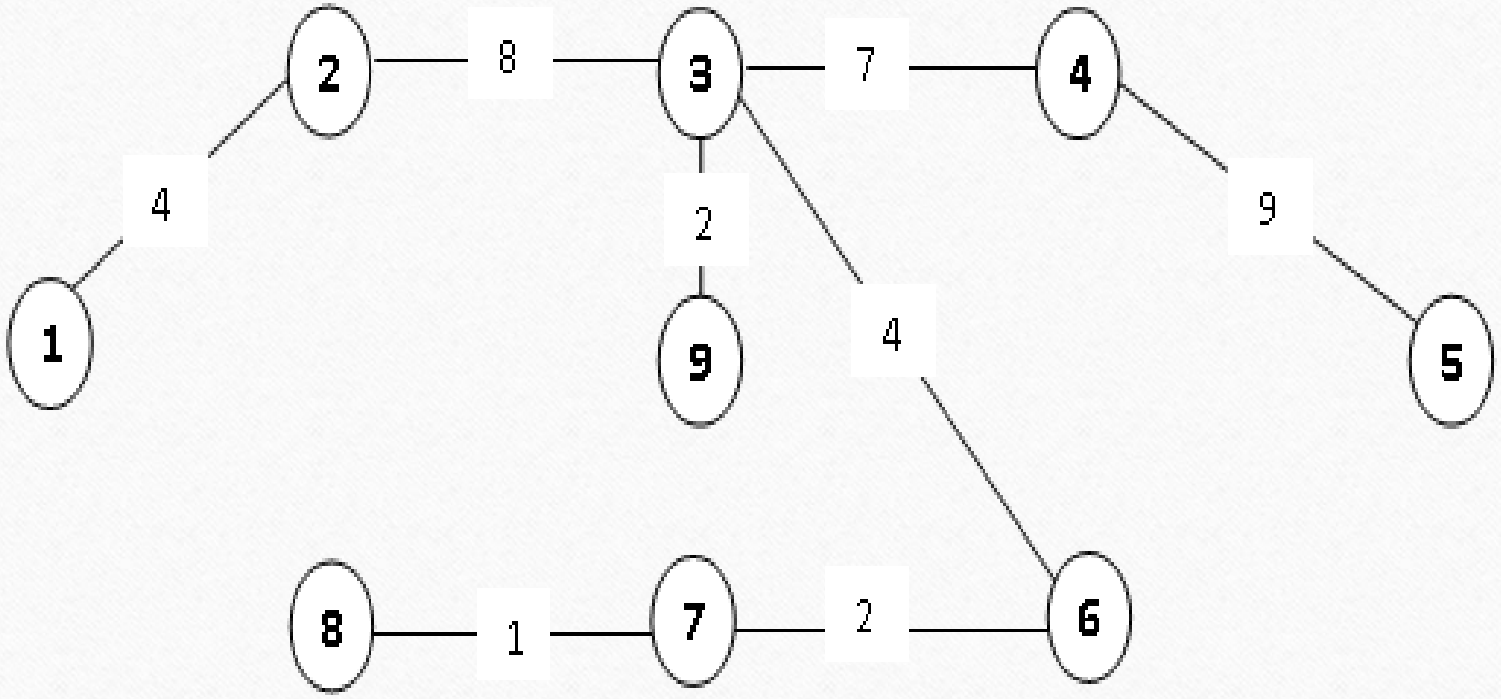


On trie les arêtes du graphe. On obtient l'ordre suivant :

$\{7, 8\} < \{3, 9\} = \{6, 7\} < \{1, 2\} = \{3, 6\} < \{7, 9\} < \{8, 9\} = \{3, 4\} < \{2, 3\} = \{1, 8\} < \{4, 5\} < \{5, 6\} < \{2, 8\} < \{4, 6\}.$

Algorithme de Kruskal

Exemple



On ajoute alors successivement dans l'ACM les arêtes :

{7, 8}, {3, 9}, {6, 7}, {1, 2}, {3, 6}, {3, 4}, {2, 3}, {4, 5}

Algorithme de Kruskal

Mise en œuvre : La difficulté majeure pour implémenter l'algorithme de Kruskal réside dans la façon de déterminer si l'arête en cours d'examen doit ou non être sélectionnée. Il s'agit de savoir si, en rajoutant l'arête { $\mathbf{s_i}$, $\mathbf{s_j}$ }, on crée un cycle ou non, autrement dit, il s'agit de savoir s'il existe déjà une chaîne entre $\mathbf{s_i}$ et $\mathbf{s_j}$. Afin de pouvoir répondre à cette question, on va partitionner l'ensemble des sommets du graphe en composantes connexes. Pour savoir si on peut sélectionner une arête {$\mathbf{s_i}$, $\mathbf{s_j}$}, il suffira de vérifier que $\mathbf{s_i}$ et $\mathbf{s_j}$ appartiennent à deux composantes connexes différentes. À chaque fois qu'on sélectionnera une arête {$\mathbf{s_i}$, $\mathbf{s_j}$}, on fusionnera les deux composantes connexes correspondantes en une seule.

Algorithme 7 : Kruskal (S , A , ν , K)

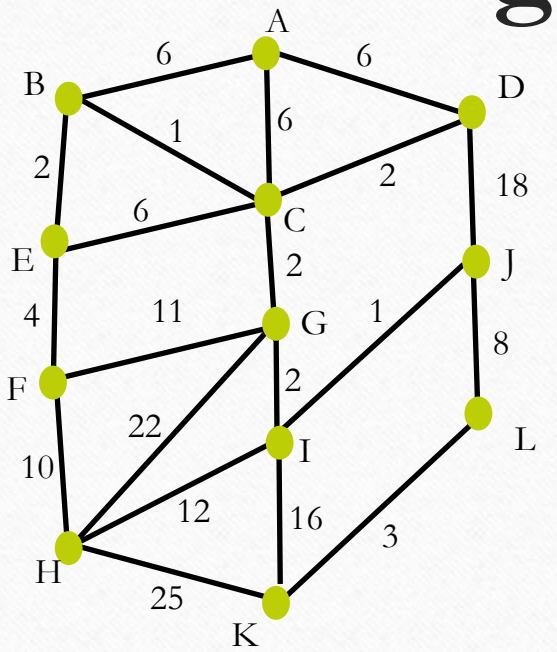
Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs,
Sorties : **K** ensemble des arêtes de l'ACM (**K ⊂ A**)

```
1 :  Pour chaque sommet s_i ∈ S Faire
2 :     Π[s_i] ← nil
3 :  Fin Pour
4 :  trier les arêtes de A par ordre de valeurs de ν croissant
5 :  K ← ∅
6 :  Tant que |K| < |S| - 1 Faire
7 :     soit {s_i, s_j} la (|K| + 1)^ieme plus petite arête de A
8 :     /* recherche de la racine r_i de la composante connexe de s_i */
9 :     r_i ← s_i
10 :    Tant que Π[r_i] ≠ nil Faire
11 :          r_i ← Π[r_i]
12 :    Fin Tant que
13 :    /* recherche de la racine r_j de la composante connexe de s_j */
14 :    r_j ← s_j
15 :    Tant que Π[r_j] ≠ nil Faire
16 :          r_j ← Π[r_j]
17 :    Fin Tant que
18 :    Si r_i ≠ r_j Alors
19 :          /* on ajoute {s_i, s_j} à l'ACM */
20 :          K ← K ∪ {{s_i, s_j}}
21 :          /* on fusionne les deux composantes connexes */
22 :          Π[r_i] ← r_j
23 :    Fin Si
24 :  Fin Tant que
25 : Fin
```

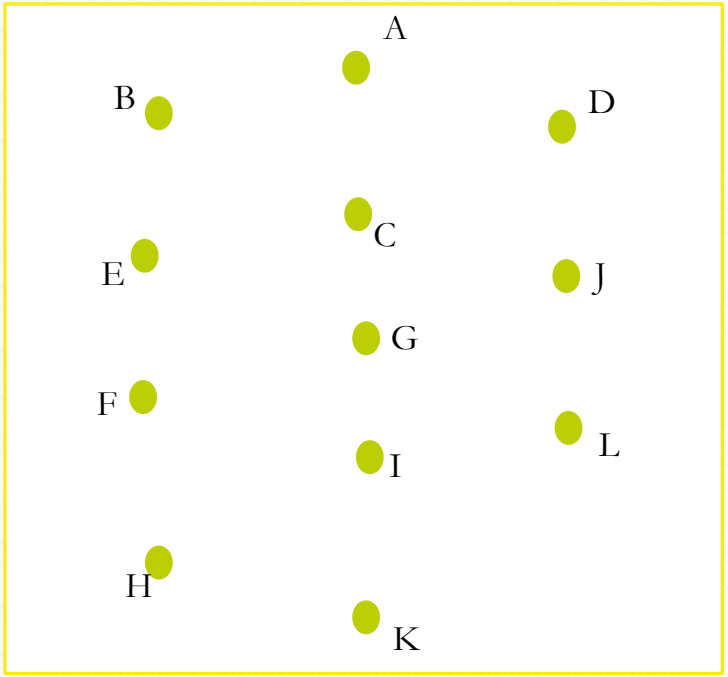
Algorithme de Kruskal

Complexité : On considère un graphe non orienté de **n** sommets et **m** arêtes. Pour trier l'ensemble des arêtes, avec une procédure de tri efficace (e.g., quicksort, mergesort ou heapsort), il faut exécuter de l'ordre de **m*log(m)** opérations. On passe ensuite, dans le pire des cas, **m** fois dans la boucle "tant que **|K| < |S| - 1**" (une fois pour chaque arête $\{s_i; s_j\}$). À chaque fois, il faut remonter des sommets s_i et s_j jusqu'aux racines des arbres correspondants. En gérant astucieusement la représentation des arbres par **Π**, on a vu que cette opération pouvait être faite en **O(log(n)).** Par conséquent, on a une complexité totale en **O(m * log(m))** (sous réserve d'utiliser une représentation par listes d'adjacence).

Algorithme de Kruskal



Next



Order of the arcs is.
BC(1), IJ(1), GI(2), CG(2), BE(2), CD(2), KL(3), EF(4), AB(6), AD(6), AC(6), EC(6), JL(8), FH(10),FG(11), IH(12), IK(16), DJ(18), GH (22), HK(25)

Start with BC

Algorithme de Prim

L'algorithme de Prim, décrit dans l'algorithme 7 est également un cas particulier de l'algorithme 8.

Cette fois ci, **K** est un ensemble d'arêtes connexes formant un arbre dont la racine est un sommet $\mathbf{s_0}$ choisi arbitrairement au début de l'algorithme. La coupure considérée à chaque itération est **(P, S \ P)** où **P** est l'ensemble des sommets qui appartiennent à l'arbre de racine $\mathbf{s_0}$ et contenant les arêtes de K. Cette coupure respecte les arêtes de **K** puisque tous les sommets appartenant à une arête de **K** appartiennent à **P** . Pour trouver efficacement l'arête de coût minimal traversant la coupure, l'algorithme maintient deux tableaux **c** et **π** tels que, pour chaque sommet $\mathbf{s_i \in P}$ pour lequel il existe une arête $\mathbf{\{s_i , s_j\}}$ traversant la coupure,

- $\mathbf{\{s_i; \pi[s_i]\}}$ est la plus petite arête partant de $\mathbf{s_i}$ et traversant la coupure **(P, S \ P)**;
- $\mathbf{c[s_i] = cout(s_i; \pi[s_i])}$.

Algorithme de Prim

Les éléments du tableau **c** sont initialisés à **1**, sauf pour les sommets $\mathbf{s_i}$ adjacents à $\mathbf{s_0}$ pour lesquels $\mathbf{c[s_i]}$ est initialisé à $\mathbf{cout(s_0 , s_i)}$. À chaque fois qu'un sommet $\mathbf{s_i}$ est ajouté à **P**, chaque arête connectant $\mathbf{s_i}$ à un sommet $\mathbf{s_j}$ de $\mathbf{S \setminus P}$ est utilisée pour mettre à jour $\mathbf{c[s_j]}$ et $\boldsymbol{\pi}\mathbf{[s_j]}$ dans le cas où le coût de $\mathbf{\{ s_i , s_j \}}$ est inférieur à la valeur courante de $\mathbf{c[s_j]}$.

Algorithme 8 : Prim (S , A , ν , K)

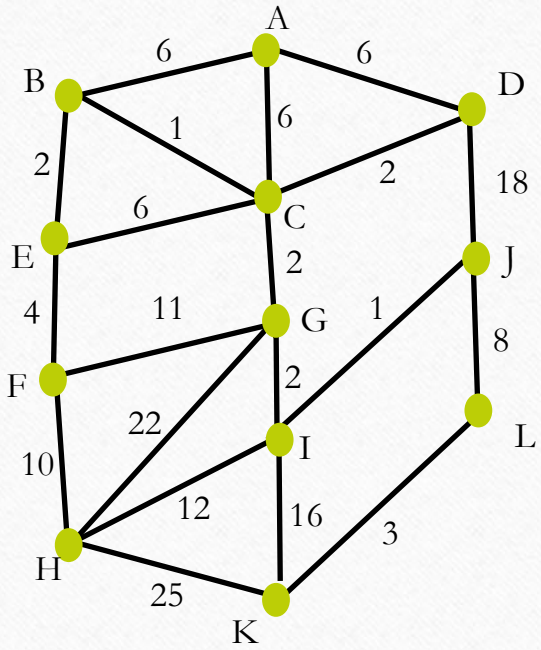
Entrées : **S** ensemble des sommets, **A** ensemble des arcs, **ν** valuations des arcs,
Sorties : **K** ensemble des arêtes de l'ACM (**K ⊂ A**)

1 : Soit s_0 un sommet de **S** choisi arbitrairement
2 : $P \leftarrow \{s_0\}$
3 : $K \leftarrow \emptyset$
4 : **Pour** chaque sommet $s_i \in S$ **Faire**
5 : **Si** $s_i \in adj(s_0)$ **Alors**
6 : $\Pi[s_i] \leftarrow s_0$
7 : $c[s_i] \leftarrow \nu(s_0, s_i)$
8 : **Sinon**
9 : $\Pi[s_i] \leftarrow$ **nil**
10 : $c[s_i] \leftarrow \infty$
11 : **Fin Si**
12 : **Fin Pour**
13 : **Tant que** $P \neq S$ **Faire**
14 : Ajouter dans **P** le sommet $s_i \in S \setminus P$ ayant la plus petite valeur de **c**
15 : Ajouter $\{ s_i , \Pi [s_i]\}$ à **K**
16 : **Pour** chaque sommet $s_j \in adj(s_i)$ **Faire**
17 : **Si** $(s_j \notin P)$ et $(\nu(s_i, s_j) < c[s_j])$ **Alors**
18 : $\Pi[s_j] \leftarrow s_i$
19 : $c[s_j] \leftarrow \nu(s_i, s_j)$
20 : **Fin Si**
21 : **Fin Pour**
22 : **Fin Tant que**
23 : **Retourne** K
24 : **Fin**

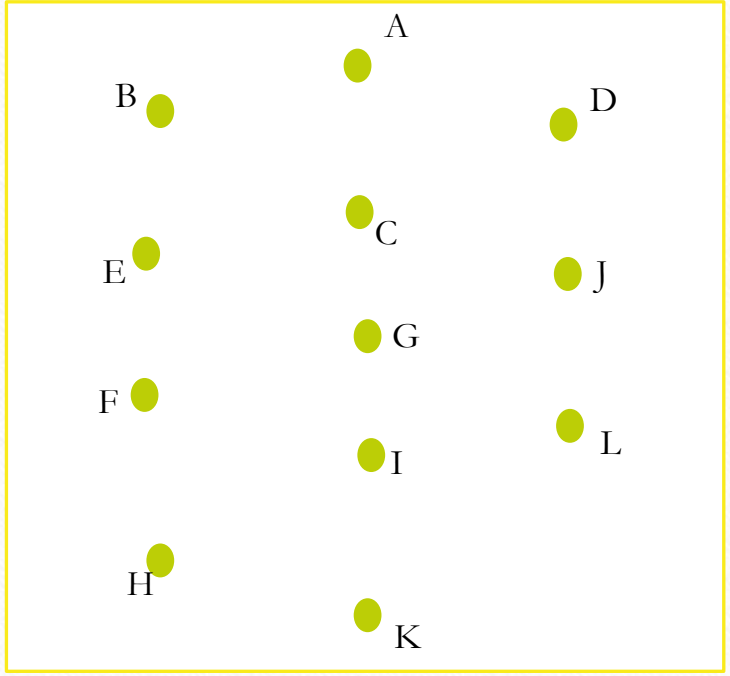
Algorithme de Prim

Complexité : Soient n et p le nombre de sommets et arêtes, respectivement. L'algorithme passe **n - 1** fois dans la boucle **lignes 13 à 22** (initialement **P** = $\{s_0\}$, un sommet est ajouté à **P** à chaque passage, et l'itération s'arrête lorsque **P = S**). À chaque passage, il faut chercher le sommet de **S\P** ayant la plus petite valeur de c puis parcourir toutes les arêtes adjacentes à ce sommet. Si les sommets de **S\P** sont mémorisés dans un tableau ou une liste, la complexité est $\mathbf{O(n^2)}$**.** Cette complexité peut être améliorée en utilisant une file de priorité, implémentée par un tas binaire, pour gérer l'ensemble **S\P**. Dans ce cas, la recherche du plus petit élément de **S\P** est faite en temps constant. En revanche, à chaque fois qu'une valeur du tableau **c** est diminuée, la mise à jour du tas est en O(log n). Comme il y a au plus **p** mises à jour de **c** (une par arête), la complexité de Prim devient **O(p log n).**

Algorithme de Prim



Next



VI.Quelques problèmes courants de théorie des graphes

Définition 2.11 (coloriage d'un graphe)

Soit **G = (S, A)** un graphe simple non-orienté, colorier le graphe **G** consiste à assigner une couleur (ou un nombre) à chaque sommet du graphe de telle sorte que deux sommets reliées par un arc/arrêté aient des couleurs différentes en utilisant le moins de couleurs possibles. Le nombre minimal de couleur est appelé **γ(G)** =nombre chromatique du graphe **G**.

De même colorier les arêtes du graphe **G** consiste à assigner une couleur (ou un nombre) à chaque arête du graphe de telle sorte que deux arêtes reliées à un même sommet aient des couleurs différentes en utilisant le moins de couleurs possibles. Le nombre minimal de couleur est appelé **γ'(G)** =indice chromatique du graphe **G**.

VI.Quelques problèmes courants de théorie des graphes

```
1 : Fonction G = Coloriage(G)
2 :     couleur courante = 1
3 :     Pour tout x sommet de G Faire
4 :         V = liste des voisins de x
5 :         couleur = plus petite couleur non encore utilisée dans V
6 :         Si couleur ≤ couleur courante Alors
7 :             colorier x avec cette couleur
8 :         Sinon
9 :             incrémenter la couleur courante et colorier x avec cette couleur
10 :        Fin Si
11 :    Fin Pour
12 : Fin
```

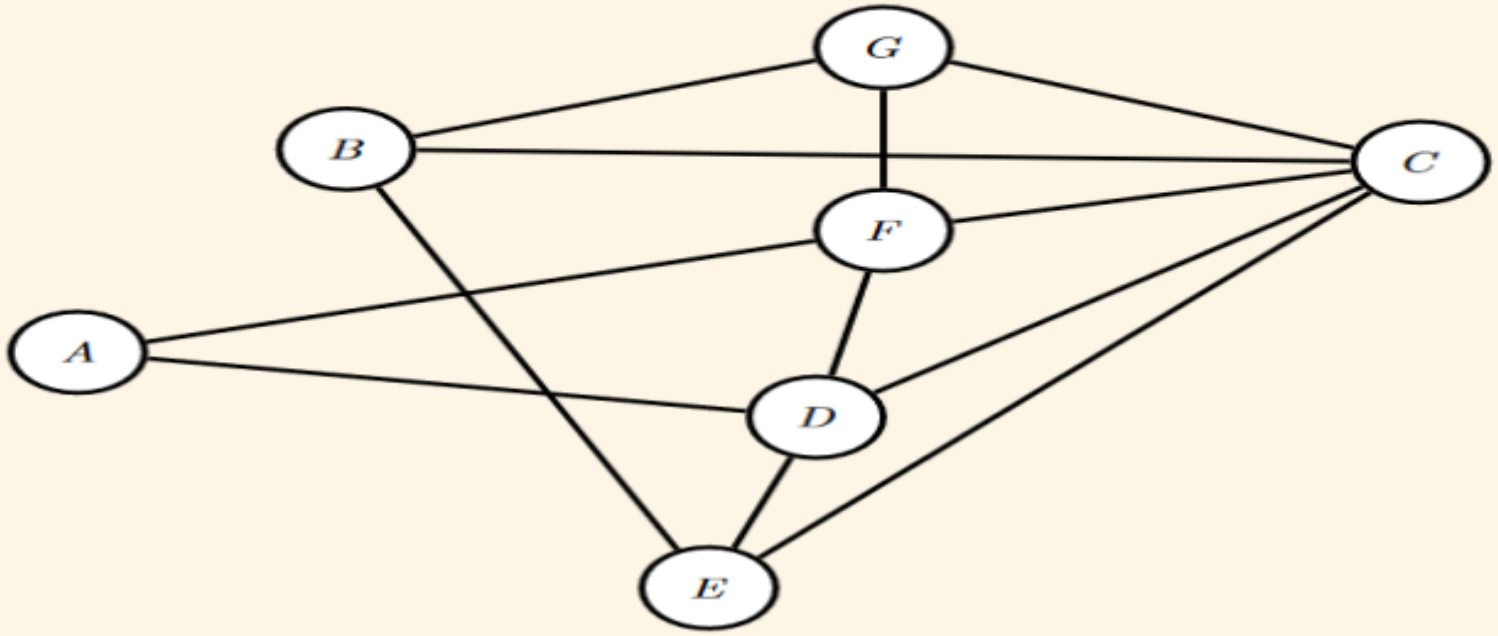
```
1 : Fonction G = Welsh(G)
2 :     L = liste des sommets classés dans l'ordre décroissant de leur degré
3 :     couleur courante = 0
4 :     Tant que L ≠ ∅ Faire
5 :         incrémenter la couleur courante
6 :         Colorier s le premier sommet de L avec la couleur courante
7 :         éliminer s de L
8 :         V = liste des voisins de s
9 :         Pour tout x dans L Faire
10 :            Si x ∉ V Alors 4
11 :                colorier x avec la couleur courante
12 :                ajouter les voisins de x à V
13 :            Fin Si
14 :        Fin Pour
15 :        éliminer les sommets coloriés de L
16 :    Fin Tant que
17 : Fin
```

L'algorithme de Welsh-Powell

Exemple

On peut classer les sommets par degré décroissant, puis les colorer :

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

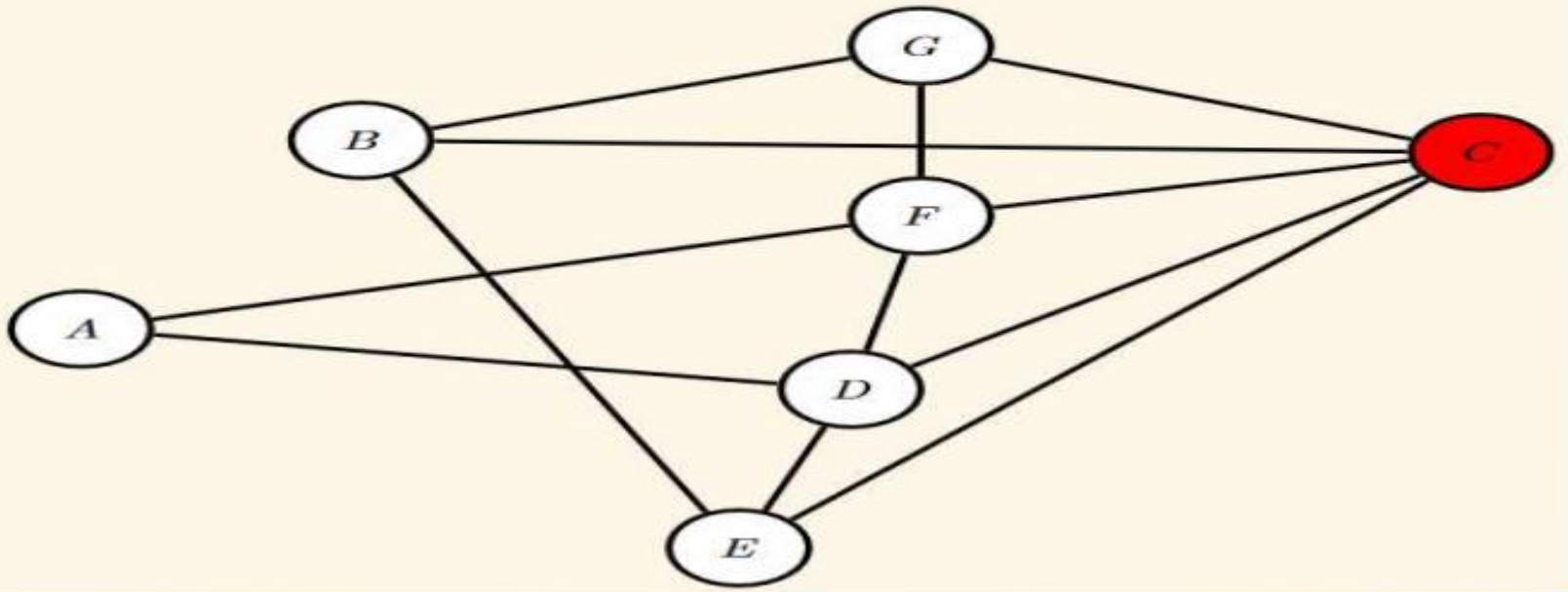


L'algorithme de Welsh-Powell

Exemple

Une coloration pourrait être :
on colorie C en rouge; puis dans l'ordre décroissant...

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

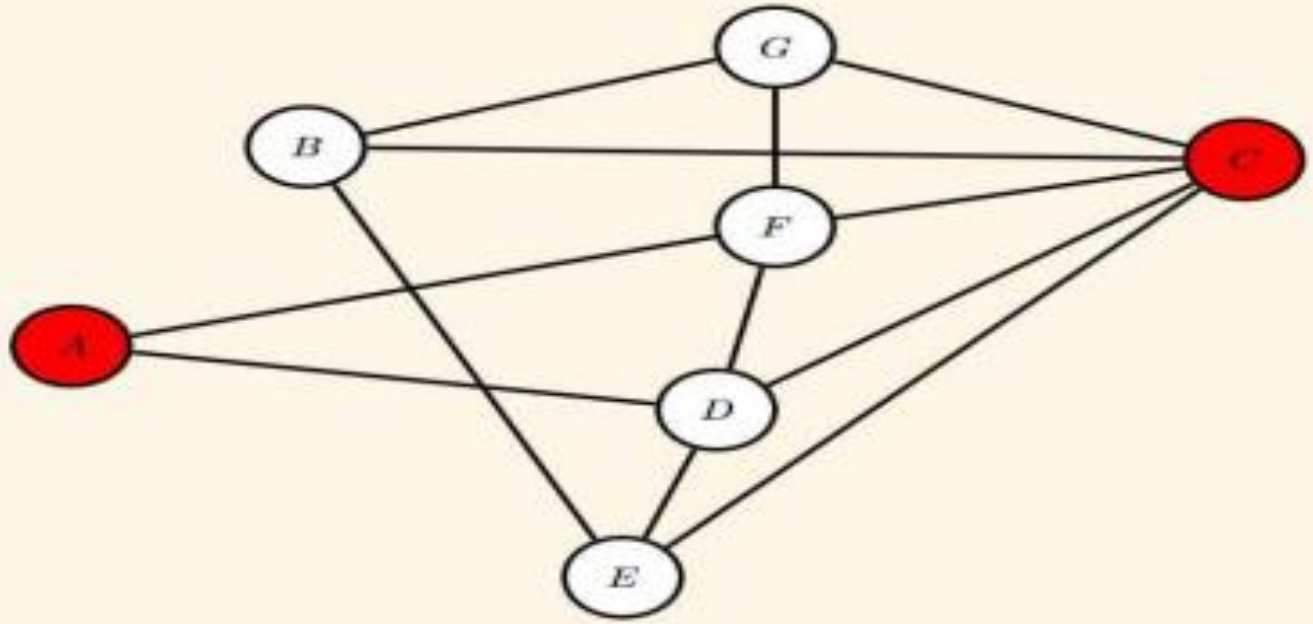


L'algorithme de Welsh-Powell

Exemple

Une coloration pourrait être :
on colorie C en rouge; puis dans l'ordre décroissant, le sommet non adjacent A en rouge.

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

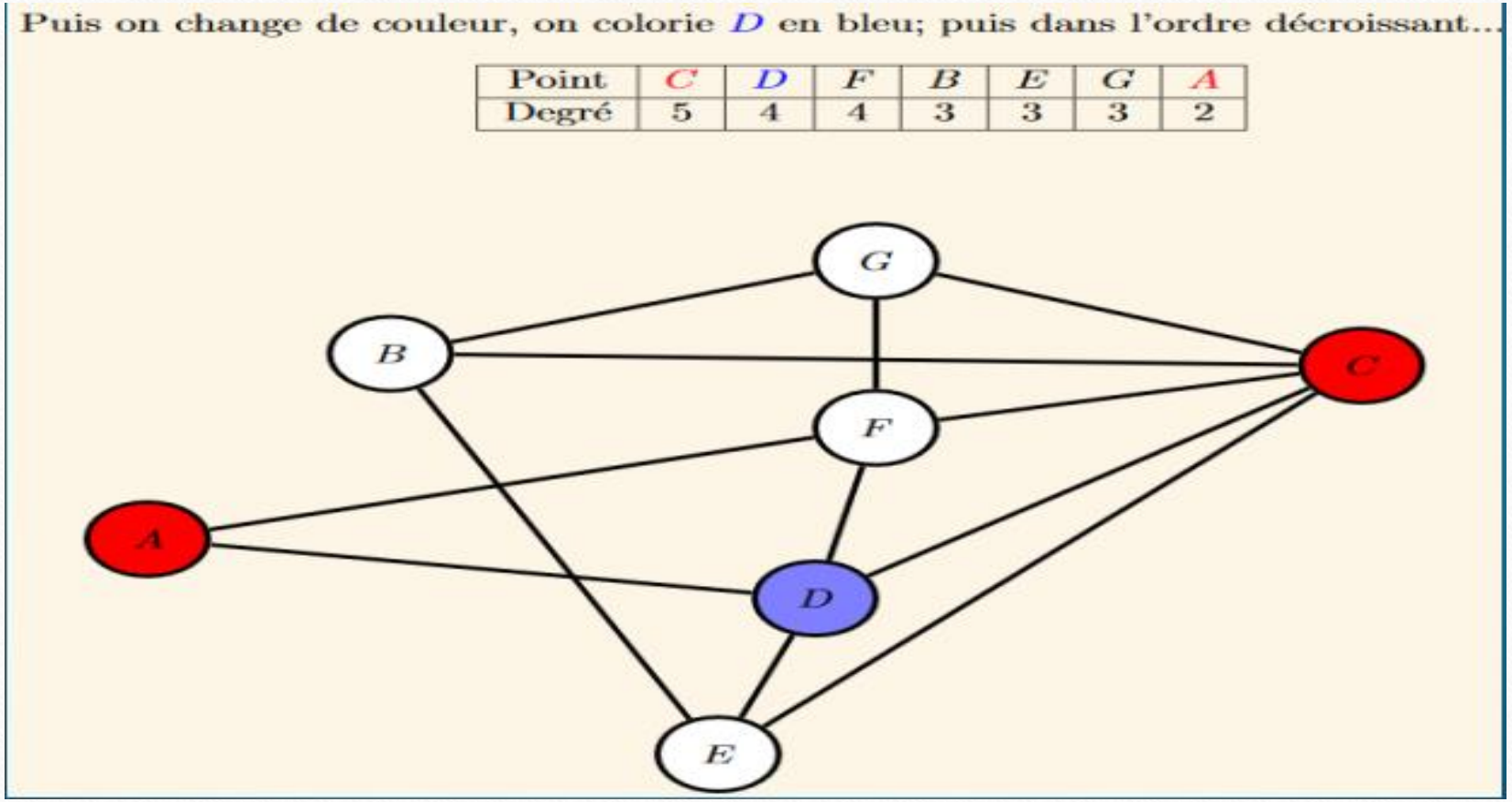


L'algorithme de Welsh-Powell

Exemple

Puis on change de couleur, on colorie D en bleu; puis dans l'ordre décroissant...

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

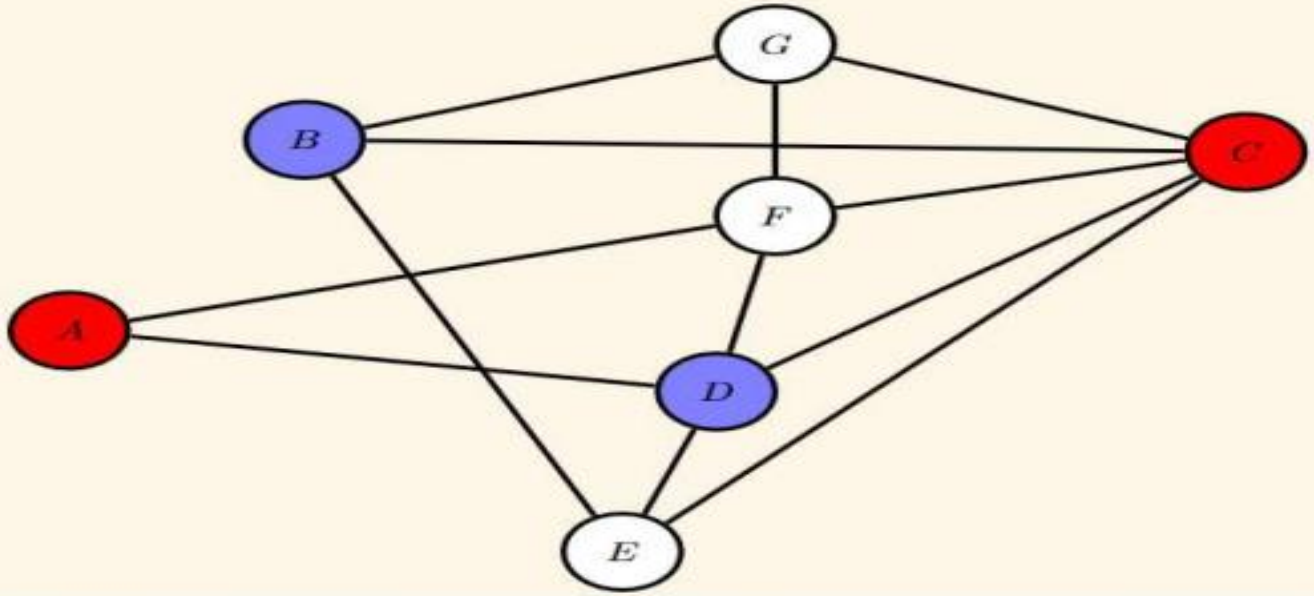


L'algorithme de Welsh-Powell

Exemple

Puis on change de couleur, on colorie D en bleu; puis dans l'ordre décroissant, le sommet non adjacent B en bleu.

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

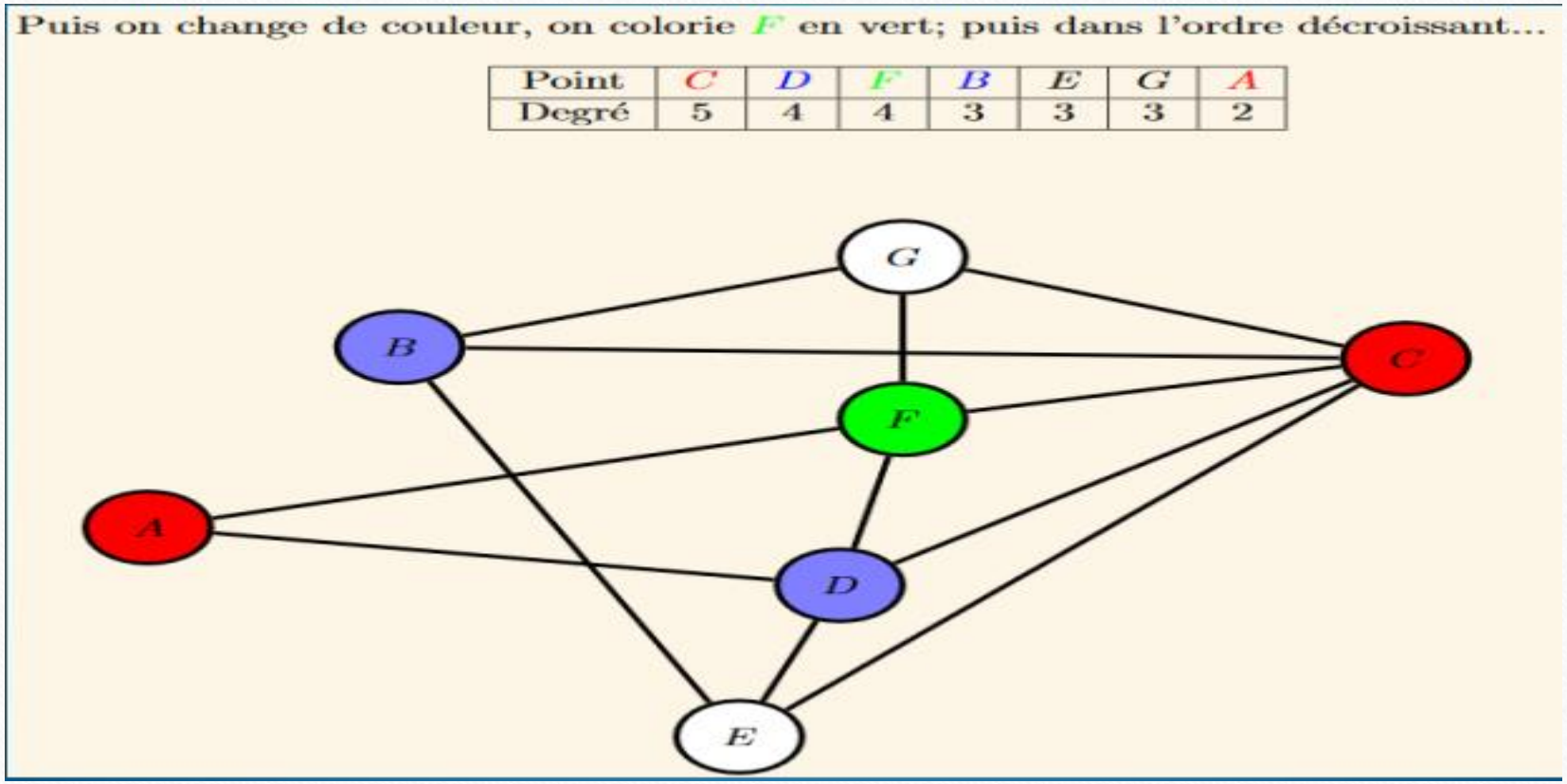


L'algorithme de Welsh-Powell

Exemple

Puis on change de couleur, on colorie F en vert; puis dans l'ordre décroissant...

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

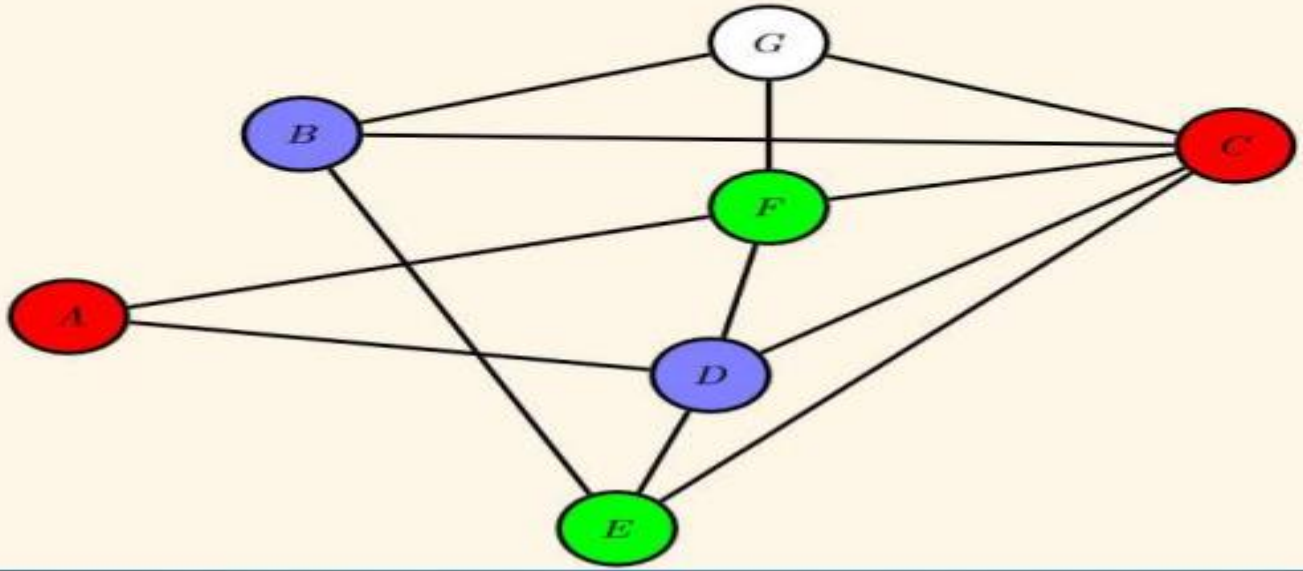


L'algorithme de Welsh-Powell

Exemple

Puis on change de couleur, on colorie F en vert; puis dans l'ordre décroissant, le sommet non adjacent E en vert.

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2

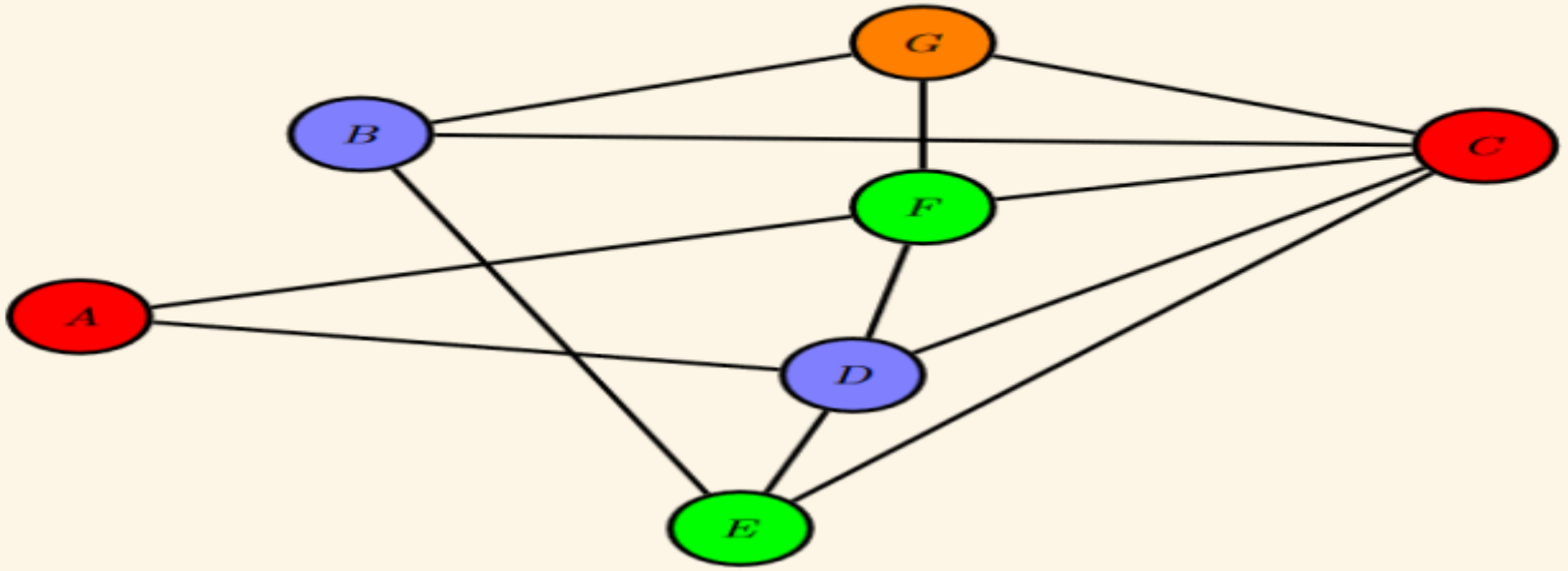


L'algorithme de Welsh-Powell

Exemple

Enfin, on change de couleur, on colorie G en orange.

Point	C	D	F	B	E	G	A
Degré	5	4	4	3	3	3	2



Attention, les deux algorithmes ne donne pas toujours le nombre minimal de couleurs !

VI.Quelques problèmes courants de théorie des graphes

L'algorithme Glouton peut être amélioré en traitant les sommets dans l'ordre décroissant de leur degré (comme dans l'algorithme de Welsh-Powell). Enfin notons que la structure du graphe impose certaines contraintes sur le nombre chromatique :

- les sommets d'une même clique doivent être coloriés d'une couleur différente
- les sommets d'un même stable peuvent tous être coloriés de la même couleur
- cela permet d'encadrer le nombre Chromatique de G :
- obtenir un coloriage à **k** couleurs permet d'affirmer que $\gamma(G) \leq k$
- trouver une clique à **k** sommets permet d'affirmer que $\gamma(G) \geq k$
- Le coloriage d'un graphe permet de résoudre de nombreux problèmes d'incompatibilité.

Questions ?